2015

Design And Implementation Of Fast Motion Estimation In Modern Video Compression On Gpu

Zhaohua Yi
University of Mississippi

DESIGN AND IMPLEMENTATION OF FAST MOTION ESTIMATION IN MODERN VIDEO COMPRESSION ON GPU

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

ZHAOHUA YI

May 9, 2015

ABSTRACT

Motion estimation is the most compute expensive part of high definition video compression. It accounts for more than 50% of overall execution. Therefore, improving the performance of motion estimation can make significant impact on the overall performance of video compression. The performance of motion estimation can be improved in two aspects: algorithm and implementation. This thesis touches both aspects. We first propose an innovative motion estimation algorithm by replacing the traditional block matching method which comparing blocks pixel by pixel with a brand new method which based on LBP (Local Binary Pattern) code. Our new method first encodes the original video frames into LBP code and then compares the blocks only using the LBP code. Our algorithm reduces the amount of computation significantly by avoiding many pixel by pixel comparisons present in traditional block matching approaches. Using public benchmarks our experiments show our proposed motion estimation algorithm runs 5 times faster than a traditional algorithm. Furthermore, we accelerate our proposed algorithm on GPUs. Motion estimation processes of all blocks are offloaded to GPU and accelerated in parallel. Our GPU implementation runs 9 times faster than CPU implementation.

ACKNOWLEDGEMENTS

I would like to express the deepest appreciation to my advisor, Dr. Byunghyun Jang, who has the attitude and substance of genius. Without his guidance and persistent help this dissertation would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

While High Definition (HD) video brings us better user experience, it also brings us more challenges on video transferring and storage. In 1080p video, for example, one static image (one single frame) is 6M bytes ($1920 \times 1080 \times 3$) in size, and the size of one hour video can reach 648G with frame rate of 30 frames per second ($6M \times 30 \times 60 \times 60 = 648G$). Such huge size makes high definition video difficult to store in current storage and transmit through network system. Therefore, compression and decompression is a necessity.

A recently emergent video compression standard, H.265 (HEVC) [13], employs a hybrid compression architecture containing intra-frame compression for single video frame compression and inter-frame compression for frame sequence compression. Inter-frame compression is a bigger challenge because it typically involves a large number of video frames and requires huge computing power, accounting for up to 80% computing cost of whole H.265 compression processes. A key part of inter-frame compression is motion estimation which consumes more than 50% of total execution time [5]. Therefore, this thesis focuses on motion estimation part of video compression.

1.1 Introduction to Motion Estimation

When it comes to video frames, adjacent frames have similar pixel values with slight movement from one frame to next frame. Therefore the purpose of motion estimation is to find these slight movement. Typically each video frame is divided into smaller blocks with pixel number in width and height of BLOCKSIZE which is configurable. More specifically, the purpose of motion estimation is to find the motion vector of each blocks in one frame by comparing the frame with the next one. Since the motion vectors record the movement of each block, the next frame can be recovered from current frame and motion vectors. Therefore it is not required to store and transfer every frame, but only store and transfer certain frames and the associated motion vectors. And during video decompressing phase, all next frames are recovered. Because motion vector is much smaller in size than original raw images, it significantly reduces the amount of data for storage and transmission which means the original image is compressed.

1.2 Challenges of Motion Estimation

Motion estimation can reduces the video data significantly, however, it also significantly increases the computing cost for both video compression and decompression. The main challenge of motion estimation is to search a block of reference frame in next frame. More specifically, there are generally four challenges in Motion estimation, the first challenge is to determine where the search start from, the second challenge is to determine the best search path, the third challenge is to efficiently compare two blocks, and the fourth challenge is to determine when the search processes should stop.

1.3 Thesis Objectives and Contributions

Various factors can limit the performance of motion estimation. Those include determining good start point, efficient block matching, smart search path and smart stop policy. Among those, this thesis focuses on the block matching because it is the most computing expensive part of motion estimation process. A better block matching algorithm will improve the whole motion estimation process significantly.The first contribution of this thesis is we propose a new algorithm of block matching. The second contribution of this thesis is we accelerate the motion estimation by using the emerging GPU computing technology.

1.4 Literature Review

In 1997, a naive motion estimation method called *Full Search (FS)* [10] was proposed. It searches blocks by using a brute force approach. The basic idea of FS is to search all possible candidates one by one in a searching range, and find the best matching blocks. Fig. 1.1 illustrates the method of FS. FS algorithm has two important configurable parameters, block size and search range. The block size refers to how many pixels are in a block which can be $8 \times 8, 16 \times 16, 32 \times 32$ or some other values if required. The search range means the largest distance (in the number of pixels) that the algorithm will search a reference block. For example, 8 pixels means the algorithm searches the reference block within 8 pixels in left, right, top, and bottom sides. Because FS uses a brute force strategy, it is considered as the most accurate motion estimation algorithm, but the main problem is a very slow speed. For 1080p frames in video, if the block size is 16×16, search range is 8, total 289 possible blocks in the next frame need to be searched for each block.

In order to improve the performance of FS algorithm, several improved algorithms are proposed. The most well known algorithm is *Three Step Search (TSS)* algorithm [7] which was published in 1998. TSS improves FS algorithm by changing the search strategy. In FS, the search strategy is full search, which means search all possible candidates one by

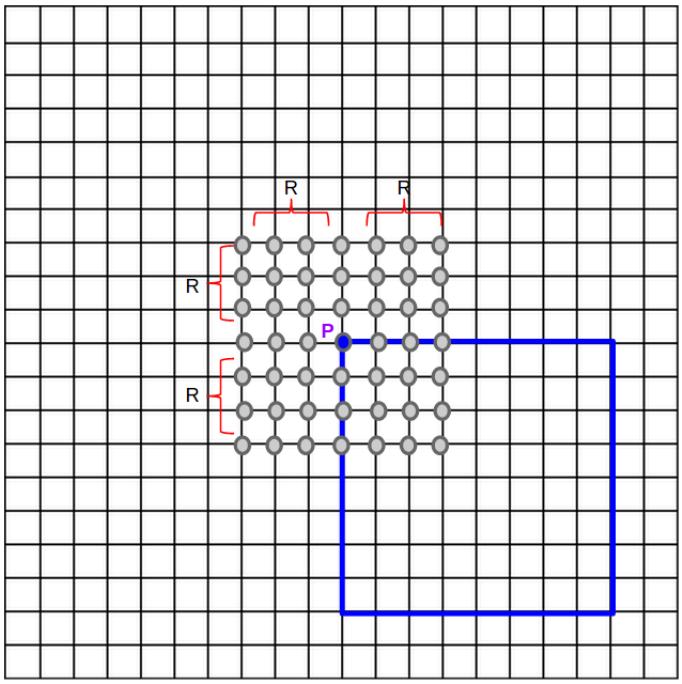


Figure 1.1. FS algorithm.

one. TSS, however, uses a selective search. First, it searchs 8 points whose distance to the original point is R, if target block is found, search stops, if not, choose the most matching points and then change the search range to R/2 and search the 8 points whose distance to new original point is R/2. If target block is found in this search step, the search stops, otherwise, by using the same strategy to choose a new original point and change the search range to R/4 and search again. As shown in fig. 1.2, in the first step, these circle points are searched, in second step, the square points are searched and in the third step the triangle points are searched.

The TSS algorithm improves FS algorithm significantly. In FS, totally 289 target possible blocks are matched for each block, but in TSS, there is only 29 points. This reduces computations by about 10 times. But its accuracy is affected. The main reason is that, after the first step, if the target block is not matched, it is hard to choose the best block as the start of next step. And then the second step maybe start from wrong points.

Because of the accuracy problem of TSS, a *New Three Step Search (NTSS)* algorithm

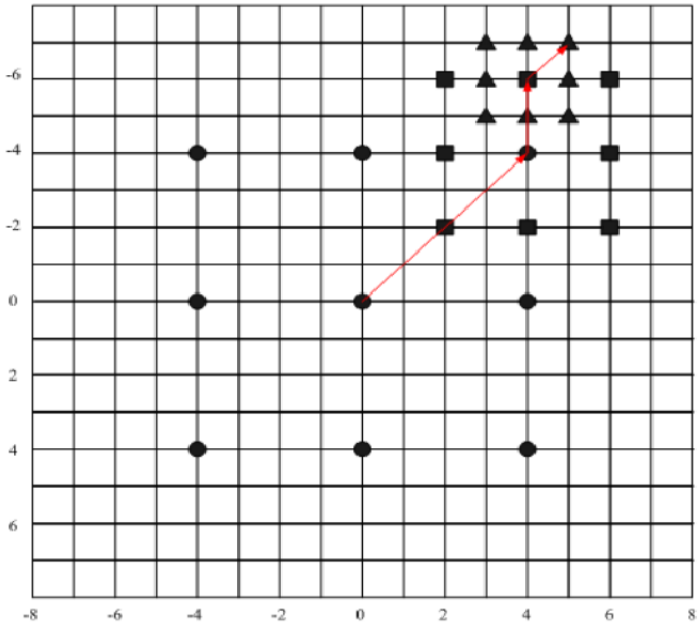


Figure 1.2. TSS algorithm.

[9] was proposed to improve TSS algorithm in 2002. The basic idea of NTSS is to add more search points to improve the accuracy. As shown in Fig. 1.3, there are more circle points in first step. These points are newly added search points in NTSS. Except these newly added search points, the search strategy is same as TSS.

With the emergence of GPGPU, the recent motion estimation research has employed GPGPU accelerations. Chen et al [4] introduced a CUDA implementation. However they did not present any new algorithm and just simply rewrote the motion estimation algorithms in CUDA.

When it comes to block matching algorithm research, there are two popular methods proposed: mean difference (1.1) [8] and mean square error (1.2) [3]. The main problem of both methods is that they compare the block pixel by pixel. In this thesis we propose a new algorithm that compare blocks not by pixels.

$$MeanDiff_{b1}^{b2} = \frac{1}{row * col} \sum_{j=1}^{row} \sum_{i=1}^{col} (|P_{ij}^{b1} - P_{ij}^{b2}|) \tag{1.1}$$

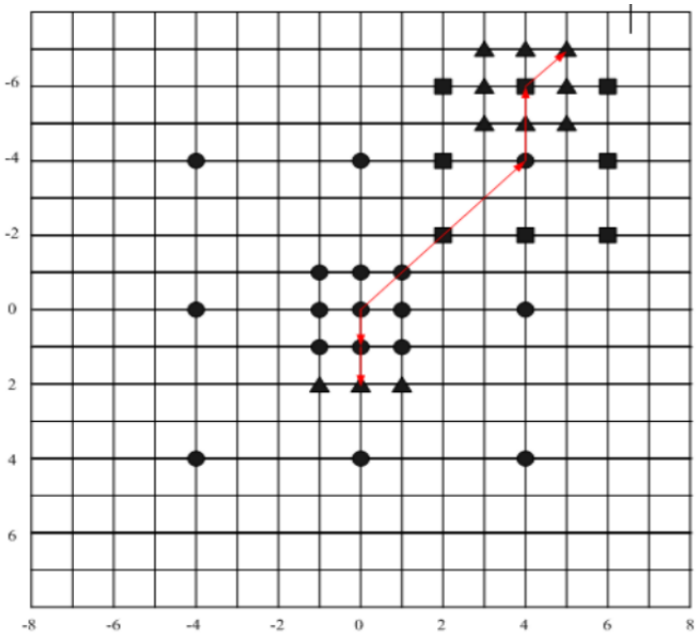


Figure 1.3. NTSS algorithm.

$$SqureError_{b1}^{b2} = \frac{1}{row * col}\sqrt{\sum_{j=1}^{row}\sum_{i=1}^{col}(P_{ij}^{b1} - P_{ij}^{b2})^2} \tag{1.2}$$

1.5 Terminologies

Motion Vector The change of location of each block between two frames. For example, the location of one block in frame f is (x, y) and in frame $f+1$ is $(x+vx, y+vy)$, the (vx, vy) is the motion vector of this block between frame f and $f+1$.

Motion Estimation The process of computing or searching the motion vector.

Reference Frame The first frame in motion estimation.

Prediction Frame The second frame in motion estimation.

Reference Block The block in reference frame.

Candidate Block The block in prediction frame.

Searching Range The largest range to search the candidate block in each reference block.

CHAPTER 2

GPGPU FOR VIDEO COMPRESSION

In this chapter, several basic skills that used in this thesis will be discussed. We first discuss why software video compression provides benefit to certain platforms and then discuss why software video compression can be accelerated by GPU computing technologies.

2.1 Hardware vs. Software Video Compression

Video compression is usually implemented in hardware on most platforms, however, there are several platforms that prefer software implementation for several reasons [6]. The first reason is that hardware implementation is expensive in cost. For some platforms, especially mobile devices, video is not a high priority, therefore additional transistor and more power is not preferred. The other reason is that the hardware implementation is not flexible to adopt new algorithms, while software approach is much more flexible. Therefore, software approach provides unique benefit to certain platforms.

2.2 GPU Computing

GPU (Graphic Computing Unit) computing is an emerging software approach for accelerating general purpose tasks which is a perfect solution to accelerate video compresion. Originally used for computing-intensive graphic rendering tasks, GPU is specially designed as many-core architecture to support many parallel threads or tasks that makes it a potential parallel computing platform for general parallel tasks which is well known as GPGPU (General Purpose GPU). The key reason that GPU can perform parallel task benefit from SIMD (Single Instruction, Multiple Data) architecture which using consist of an array of processors with a control processor and a main memory. On each clock cycle, the control processor issues an instruction to each processor using the control bus. Each processor performs that instruction and (optionally) returns a result to the memory via the data bus. With SIMD architecture, GPU execute same instructions on multiple data elements simultaneously which improves the performance significantly.

2.3 OpenCL

OpenCL [12] and CUDA [11] are two GPGPU programming frameworks that enable the GPU for general computing tasks. OpenCL and CUDA are very similar in their core programming model yet differ in available platforms: OpenCL is industry standard supported by most hardware vendors while CUDA is available only on NVIDIA hardware. This thesis uses OpenCL for experiments. In OpenCL computing model, parallel computing tasks are mapped into a N dimension thread structure called NDRange where N can be configured into one, two or three dimension according to the actual task structure. For efficient organization and scheduling, NDRange is further divided into multiple work-groups where each work-group contains multiple work-items. Work-item is a basic computing unit that processes an independent data element and all work-items in the same work-group can process data elements simultaneously. The program of work-item that runs on GPU device is called

kernel.

OpenCL provides many benefits in the field of high-performance computing, and one of the most important is portability. OpenCL kernels can execute on GPUs and CPUs from such popular manufacturers as Intel, AMD and Nvidia. New OpenCL-capable devices appear regularly, and efforts are underway to port OpenCL to embedded devices, digital signal processors, and even FPGA (field-programmable gate arrays). Not only can OpenCL kernels run on different types of devices, but a single application can dispatch kernels to multiple devices at once. For example, if the computer contains an AMD CPU and an AMD GPU card, applications can synchronize kernels running on both devices and share data between them. These features make OpenCL an ideal GPU computing programming choice.

2.4 GPU Computing for Motion Estimation

Several features of Motion Estimation task make it be accelerated by GPU computing easily. The first and most important feature is that the motion estimation task is a per-block task, which means every block will be processed with the same way. As discussed above, SIMD computing model can efficiently accelerate these kind of tasks. Every core in the GPU device can process one block and all blocks can be processed simultaneously. The second feature of motion estimation is that the block number is large, therefore additional effort of offloading computing task to GPU can be hidden by beneficial of large parallel computing throughputs. In this thesis, the detailed implementation of using GPU computing to accelerate motion estimation will be discussed in Chapter 4.

CHAPTER 3

THE PROPOSED ALGORITHM

In this chapter, the detailed algorithm will be discussed. First, we will discuss the original FS algorithm with the complexity analysis. And then we will introduce our proposed algorithm FS-EBME with the complexity analysis. Finally the complexity of these two algorithms will be compared.

3.1 Traditional FS (Full Search) Algorithm

A naive brute force algorithm of motion estimation algorithm is known as *Full Search (FS)* algorithm. As the name indicates, this algorithm searches every possible candidate blocks in the searching range as shown in Fig. 3.1 where P is the top-left point of reference block in reference frame, blue block is the reference block and gray points are the top-left points of all candidate blocks in prediction frame and R shows the searching range.

For each block at the position of P, the FS algorithm will search all possible positions in a search range R. The algorithm 1 illustrates the FS algorithm.

Algorithm 1 Full Search Motion Estimation.

```
for each Block b at position of ref in CURRENT image do
  P_0 = the same position of ref in NEXT image
  Range = 1
  MinDiff = BlockMatching(ref, p_0)
  MV_b = P_0 − ref
  if MinDiff <= MinDiffThreshold then
    Return
  end if
  while Range <= SearchingRange do
    for each point p in this range in NEXT image do
      Diff = BlockMatching(ref, p)
      if Diff < MinDiff then
        MinDiff = Diff
        MV_b = p − ref
      end if
      if MinDiff <= MinDiffThreshold then
        Return
      end if
    end for
    Range++
  end while
end for
```

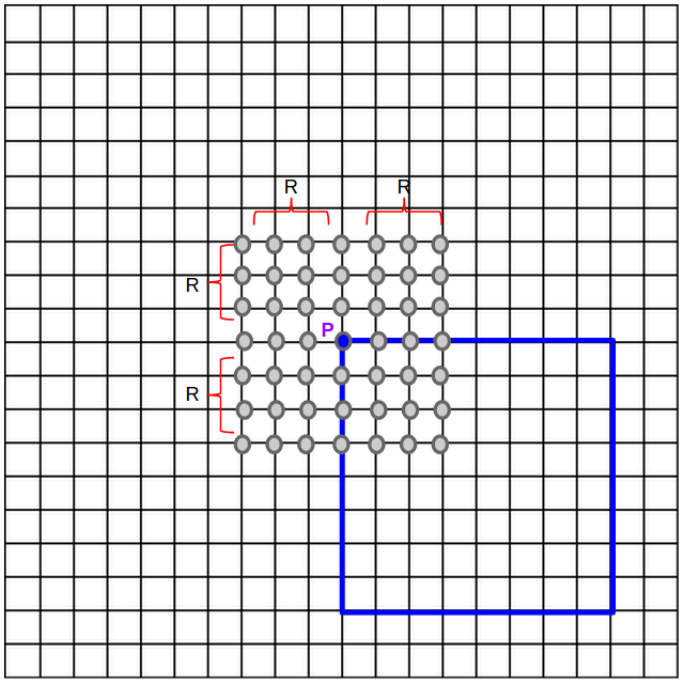


Figure 3.1. FS algorithm.

3.1.1 Block Matching Method

Algorithm 1 tells us that the block matching is a key function and the most expensive in motion estimation. A traditional block matching method is simply to compare individual pixels in the block one by one. The difference of whole block is represented as the square error of all pixels in the block. Therefore, the purpose of block matching is to search all candidate blocks in prediction frame to find the block who has the minimum block difference with reference block.

Algorithm 2 illustrates the process of matching two blocks. Assume the block has *col* columns and *row* rows, for each pixel at the location of column = i and row = j, the deference between two pixels is $P_{ij}^{b1} - P_{ij}^{b2}$, that is, the difference between these two blocks is euclidean distance as shown in the following equation.

$$Diff_{b1}^{b2} = \sqrt{\sum_{j=1}^{row}\sum_{i=1}^{col}(P_{ij}^{b1} - P_{ij}^{b2})^2} \tag{3.1}$$

Algorithm 2 Block Matching.

$j = 1$
$i = 1$
$Diff = 0$
while $j < row$ **do**
  while $i < col$ **do**
    $Diff+ = (PIXEL_{ref-ij}^{curent-image} - PIXEL_{p-ij}^{next-image})^2$
  end while
end while
$Return\ sqrt(Diff)$

3.2 The Complexity of Traditional FS Algorithm

There are several parameters used to evaluate the complexity of FS algorithm:

- Image Size

Image Size presents the total pixel number in the image, and it is denoted by two

parameters width W and height H, therefore the total pixel number is $W \times H$. The Block Size impact the complexity significantly because more pixels require more computing time. Therefore, the complexity of video compression for 1080p video (width is 1920 and height is 1080) is much higher than 720p video (width is 1280 and the height is 720).

• Block Size

Block Size presents the width and height of each block. In this thesis, block is a square where width equals height that is denoted by B. With Block Size B, the total block number in one video frame is $(W/B) \times (H/B)$. Choosing an appropriate Block Size is a trade-off among variant considerations. Larger block size results in less block number, but the block matching time for individual block is increased. On the other hand, smaller block size cause more block number while the block matching time is decreased. The Block Size also impact the accuracy of motion estimation. In this thesis, we did not test different Block Size, but just used the industry recommended Block Size $B = 16$.

• Search Range

Search Range is a parameter illustrates how many candidate blocks will be matched with reference block, and it is denoted by R. R presents the distance in pixels to search the blocks. If the search range is R, the total number of blocks need to be compared to compute the motion vector is $(2 \times R + 1) \times (2 \times R + 1)$

3.2.1 Complexity Evaluation

We evaluate the complexity by counting the total mathematical operations for motion estimation by using the equation 3.2:

$$C = C_1 \times C_2 \times C_3 \tag{3.2}$$

Where total complexity C is determined by three sub-complexity: the number of blocks C_1 , the number of candidate blocks that be compared with the reference block C_2

and the total mathematical operations of one block matching C_3.

As discussed above, the total number of blocks is determined by the Image Size (W and H) and Block Size (B), and C_1 can be computed by the equation 3.3;

$$C_1 = (W/B) \times (H/B) \tag{3.3}$$

The total number of candidate blocks is determined by the Search Range R, and the complexity C_2 can be computed by the equation 3.4.

$$C_2 = (2 \times R + 1) \times (2 \times R + 1) \tag{3.4}$$

The complexity of block matching C_3 is determined by the block size and the block matching algorithm. In this block matching algorithm, for comparing one pixel, there are three operations, one minus, one square and one addition. Therefore the complexity of C_3 can be computed by equation 3.5;

$$C_3 = 3 \times B \times B \tag{3.5}$$

By merging the three sub-complexity equations, the total complexity C can be computed by the equation 3.6;

$$C = C_1 \times C_2 \times C_3 = (W/B) \times (H/B) \times (2 \times R + 1) \times (2 \times R + 1) \times 3 \times B \times B \tag{3.6}$$

For example, for a 1080p HD image where $W = 1920$, H=1080. If $B = 16$, $R = 8$. The total complexity of motion estimation of one image is:

$(1920/16) \times (1080/16) \times 17 \times 17 \times 3 \times 16 \times 16 = 1,811,128,320$

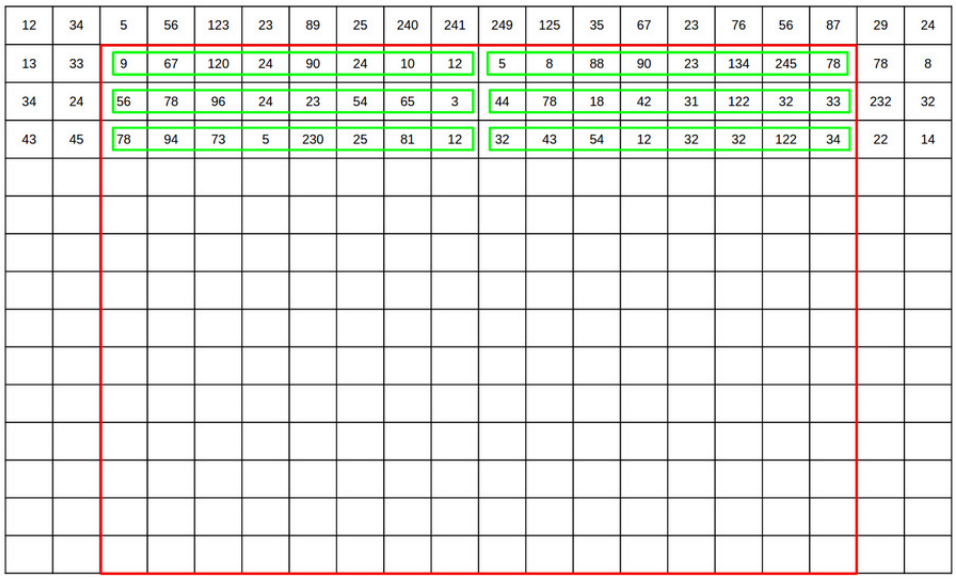


Figure 3.2. Transfer pixels to long word.

3.3 Our Proposed Algorithm (FS-EBME)

We proposes a new block matching algorithm that compares blocks by not comparing pixels but comparing combined pixels after encoding. The basic idea is to combine adjacent 8 bytes as a long word. Therefore, when we compare 8 bytes only one time as a long integer, we don't need to compare the bytes one by one for eight times. Theoretically this approach speeds up motion estimation by 8 times. An example is shown in Fig. 3.2. One block contains 16×16 pixels and we need to compare total 256 pixels with 256 comparisons. If we combine 8 bytes to a long word, we just need to compare only 32 words for entire block.

However, it is impossible to combine the pixel value directly into a long word in real video. In adjacent frames of same video, even for same object, the pixel values are impossible to be exactly same in two frames because of noise. Some pixel values may have very slight change between frames. For example the pixel value may be changed from 100 to 101. It is tolerable if we compare pixels, but if we combine bytes to a word, there is likely to be different.

Therefore, we need transform raw pixel values to a stable format so that we can combine bytes to long integers to reduce the comparisons. In this thesis, we present an

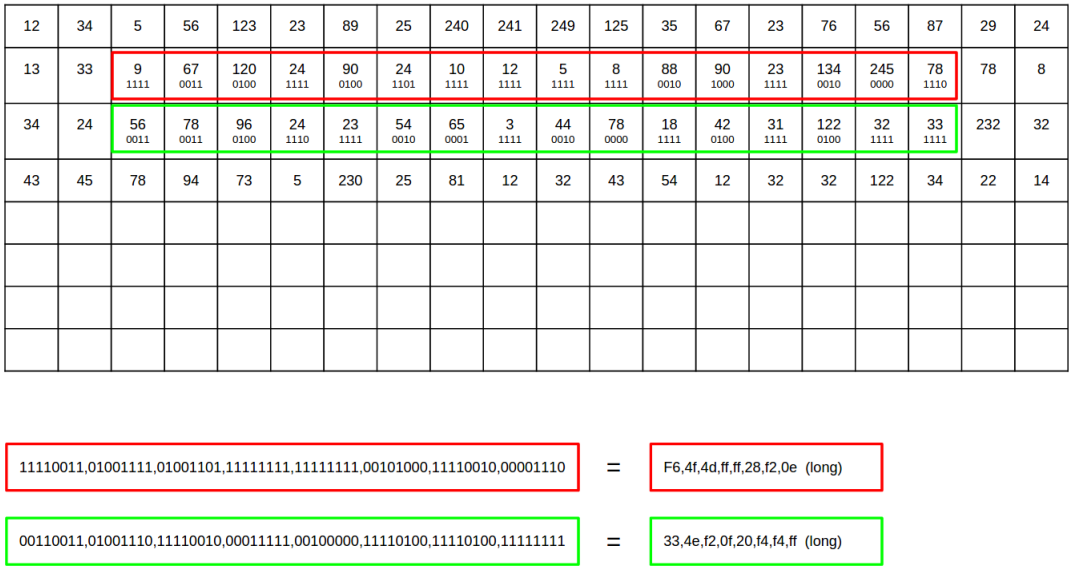


Figure 3.3. Example of our LBP encoding.

innovative encoding method by borrowing the idea of LBP (Local Binary Pattern) [1]. The basic idea of LBP is transforming each pixel value to a relationship among the pixel value of its neighbors. For each pixel, there are eight neighbors (left, top, right, bottom and four diagonal positions). If the value of this pixel is larger than the value of its neighbor, the corresponding bit is set to 1, otherwise set to 0. Therefore, for each pixel, the LBP code is 8 bits that is one byte. In order to reduce the LBP code size, in this thesis, only four neighbors are used for LBP code (only left, right, bottom, top), that means for each pixel, the LBP code is 4 bits, and the LBP code of two adjacent bytes will make one byte. Therefore, 16 adjacent pixels will generate 8 bytes, in other words, 16 adjacent pixels will generate one LBP long word. By using this encoding method, only one comparison is required to compare 16 pixels in block matching function, this improves the block matching and motion estimation significantly. We call our block matching algorithm an Enhanced Block Matching by Encoding (EBME).

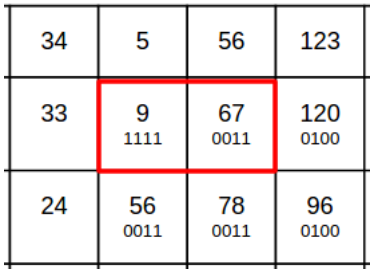


Figure 3.4. Details of LBP encoding.

3.3.1 Encoding Process

Fig. 3.3 illustrates the detailed process of our LBP encoding method. In our LBP encoding method, each pixel will be compared with its four neighbors, left, right, bottom and top. Four bits are used to store the comparing relationship between this pixel and its four neighbors. The left is the first bit (the most left one), top is the second bit, right is the third bit and bottom one is the fourth bit (the most right one). For each pixel, if the pixel value is less than the value of the left pixel, the first LBP bit is set as zero (0), otherwise one (1). By using the same rule, the remain three bits are set by comparing with its other neighbors.

Fig. 3.4 shows an example. The pixel 67 is encoded to 0011, because the left value 9 and top value 56 is less than 67, therefore the first and second LBP bits are set to 0, the right pixel 120 and bottom pixel 78 is larger than 67, therefor the third and forth LBP bits are set to 1. Because each pixel has 4 bits LBP code, two adjacent pixels have eight bits LBP code which is one byte. In this example, pixel 9 and pixel 67 have one byte of LBP code F3 (11110011). By using this LBP encoding method, an image of $W \times H$ pixels will be encoded into $(W/2) \times H$ LBP bytes.

3.3.2 Our Enhanced FS Algorithm (FS-EBME)

By adopting our new block matching algorithm EBME, the original FS algorithm is promoted to the *FS-EBME* algorithm. FS-EBME is changed into two steps, the first step is to encode the two input frames (reference frame and prediction frame) into LBP code, and then the

second step is to use full search strategy to search the motion vector of each block by using EBME algorithm.

The LBP encoding step is to encode every pixel of input image. For those pixels that locate in the border of the image, we assume the pixel value is zero if it is out of the image.

3.4 The Complexity of the FS-EBME Algorithm

Our FS-EBME algorithm can be evaluated by using the same evaluation method of counting the total mathematical operations. The difference is that there is additional computing cost of encoding, therefore the total complexity C of our FS-EBME algorithm is denoted by the equation 3.7:

$$C = C_1 \times C_2 \times C_3 + C_4 \tag{3.7}$$

Where total complexity C is determined by three sub-complexity: the number of blocks C_1 , the number of candidate blocks that be compared with the reference block C_2, the total mathematical operations of one block matching C_3 and the complexity of encoding C_4.

As discussed above, the total number of blocks is determined by image size (W and H) and block size (B), and C_1 can be computed by the equation 3.8;

$$C_1 = (W/B) \times (H/B) \tag{3.8}$$

The total number of candidate blocks is determined by the Search Range R, and the complexity C_2 can be computed by the equation 3.9.

$$C_2 = (2 \times R + 1) \times (2 \times R + 1) \tag{3.9}$$

The complexity of block matching C_3 is determined by the block size and the block matching algorithm. In this block matching algorithm, every line is combined as a long word and requires only one comparing. Therefore the complexity of matching one block only is same as the number of lines in the block. The complexity of C_3 can be computed by equation 3.10;

$$C_3 = B \tag{3.10}$$

The complexity of encoding is 8 operations, because four neighbors will be compared and every comparison requires two operations (one minus and one shift operation). Therefore the complexity of encoding for whole image is

$$C_4 = 8 \times W \times H \tag{3.11}$$

By merging the four sub-complexity equations, the total complexity C can be computed by the equation 3.12;

$$C = C_1 \times C_2 \times C_3 + C_4 = (W/B) \times (H/B) \times (2 \times R + 1) \times (2 \times R + 1) \times B + 8 \times W \times H \tag{3.12}$$

For example, for a 1080p HD image of $W = 1920$, H=1080. If $B = 16$, $R = 8$, the total complexity of computing motion vector of one image is:

$1920 \times 1080 \times 4 + (1920/16) \times (1080/16) \times 17 \times 17 \times 16 = 16,588,880 + 37,454,440 = 54,043,320$.

3.5 Discussion

Theoretically, our algorithm FS-EBME in the same complexity level with traditional FS algorithm, because our algorithm is constant times faster than traditional FS algorithm. However, practically, constant times improvement is significant in real application. By comparing the complexity of our new method with the traditional one, the theoretical complexity

is improved by 33 times.

CHAPTER 4

EXPERIMENTS

4.1 Experiment Configuration

Our proposed algorithm is evaluated by the following four experiments: First, we implemented a basic FS algorithm running on CPU as our baseline. This is a naive brute force method, we assume this is the most accurate method. We then implemented the following three configurations to compare its accuracy and performance. Second, we improved the baseline by using GPU to accelerate the traditional FS algorithm with several optimization methods. Third, we improved the baseline from another view by using our proposed algorithm but not using GPU. Last, we further improved the third experiment by using GPU to do acceleration. The following subsections will discuss the details of each experiment and compare the result.

4.2 Original FS Algorithm on CPU (Experiment 1.1)

High Definition (HD) image contains thousands of blocks, and each block needs to be processed independently for motion estimation. However, because CPU can only execute tasks serially, these blocks have to be processed one by one. Fig. 4.1 illustrates the implementation

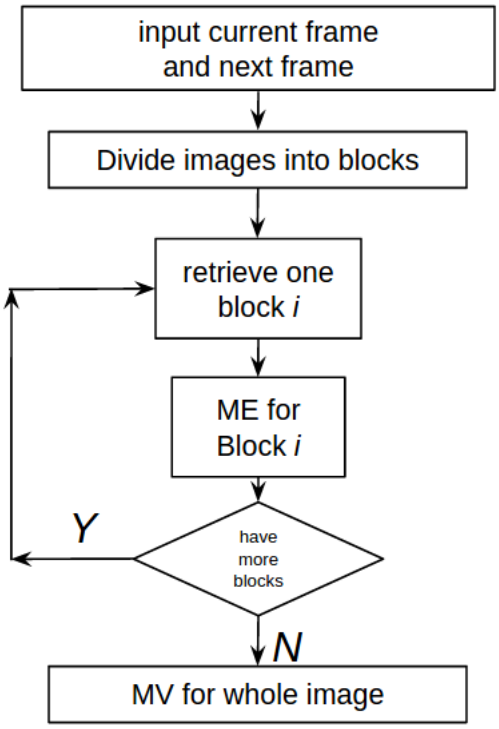


Figure 4.1. Baseline of FS implementation.

of our baseline. The first step is dividing the reference frame into blocks, and then retrieving blocks one by one to feed to motion estimation function for computing the motion vector.

4.3 Original FS Algorithm on GPU (Experiment 2.x)

4.3.1 Implementation Details (Experiment-2.1)

In this experiments, we use GPU to accelerate the FS motion estimation algorithm. As discussed above, a major computational challenge of the baseline experiment is that the blocks in an image have to be processed one by one serially, which is inefficient. This problem can be improved by using GPU's SIMD executing model where blocks are offloaded to GPU and processed in parallel as Fig. 4.2 illustrates. In this execution model, each GPU core process one block independently and simultaneously which is illustrated by Fig. 4.3 where images (reference image and prediction image) are uploaded into the GPU, and GPU

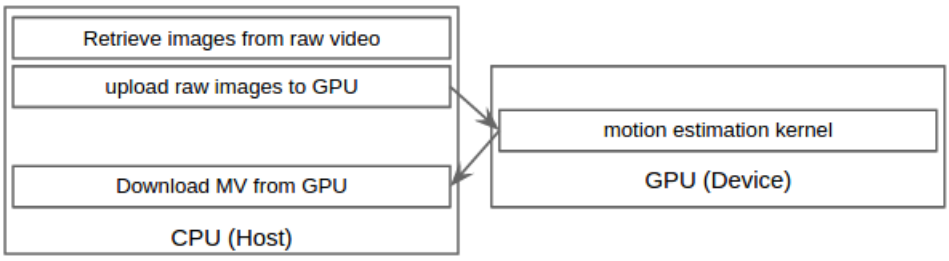


Figure 4.2. Offloading motion estimation to GPU.

will schedule many-core to perform the motion estimation for all blocks simultaneously.

For implementation, we have chosen the cross-platform, industry standard OpenCL programming framework to implement our motion estimation algorithms on CPU-GPU heterogeneous systems. OpenCL targets to exploit both Task-Level Parallelism (TLP) and Data-Level Parallelism (DLP) present in application. We first perform a task-flow analysis to identify the data-parallel portions of the application, each of which becomes a candidate for GPU kernel. GPU kernels are executed on manycore GPU hardware in a SIMT (Single Instruction Multiple Threads) and SIMD (Single Instruction Multiple Data) fashion. Fig. 4.4 illustrates the mapping between the image blocks and OpenCL threads. In Fig. 4.4, each blue rectangle illustrates one block in the reference frame and it is processed by one GPU core.

4.3.2 Optimization Methods

Optimizing GPU kernel is a challenging task as it requires the deep understanding of underlying GPU hardware architecture and its thread execution model. The performance of GPU kernel is very sensitive to small changes in thread configuration and memory access patterns.

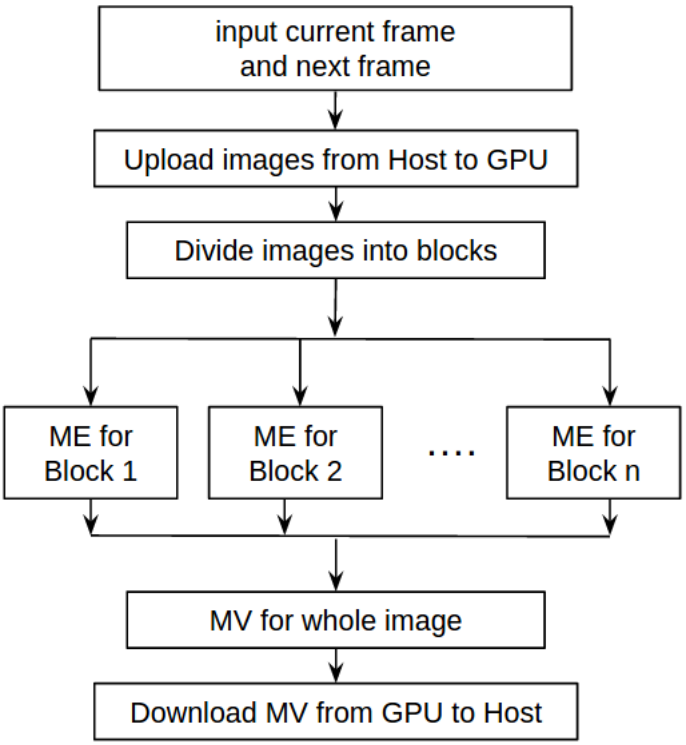


Figure 4.3. FS algorithm implementation with GPU acceleration.

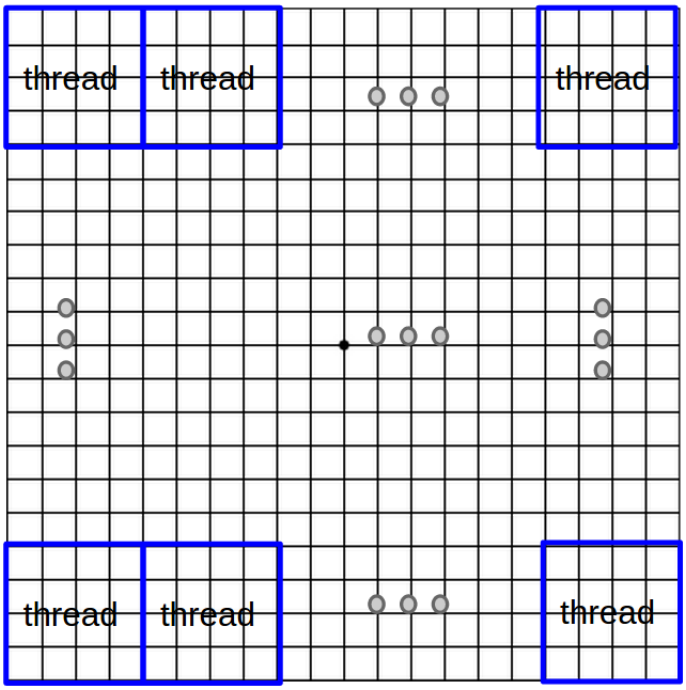


Figure 4.4. Mapping blocks to GPU threads.

```
//BLKSZ is constant 16
for (j=0;j<16;j++)
{
        for(i=0;i<BLKSZ;i++)
        {
                //compare pixel[j][i]
        }
}
```

```
//BLKSZ is constant 16
for (j=0;j<16;j++)
{
        //compare pixel[j][0]
        //compare pixel[j][1]
        ...
        //compare pixel[j][15]
}
```

Figure 4.5. Loop Unrolling.

4.3.2.1 Loop Unrolling (Experiment 1.2 and Experiment 2.2)

Designed for data intensive computing, control flow must be avoided to achieve high performance on GPUs. Therefore, if the number of control instructions can be reduced, the performance of GPU will be improved. Loop unrolling is an efficient way to reduce the control flow of the program. Our first optimization is to apply loop unrolling technique to reduce control instructions.

In order to compare two blocks pixel by pixel, two-level nested loop is used to iterate all pixels in the block. First loop is to iterate the block row by row and the second loop is to iterate column by column in the same row. The code on the left side in Fig. 4.5 illustrates this process. In this implementation, one control instruction is required for each pixel comparison. Since the iteration count is fixed to BLKSZ (e.g., 16 in our experiments), we can remove the second loop and replace it with 16 statements directly as in the right side of Fig. 4.5.

4.3.2.2 Local Memory (Experiment 2.3)

Multiple memory space is one of features of modern GPU architecture. On-chip memory is fast, however the size is small because it is expensive. Off-chip memory is larger, however it is slow because it requires significantly more cycles to access off-chip memory. One of our important optimizations is to utilize on-chip scratchpad local memory which is as fast as cache. Prefetching frequently used data from slow off-chip global memory to this local

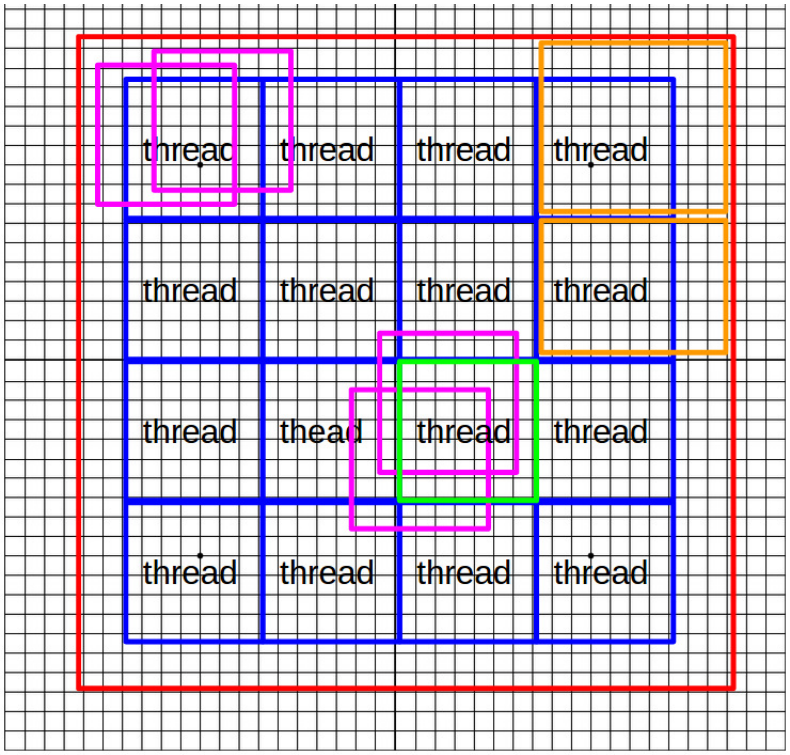


Figure 4.6. Using local memory in OpenCL kernel.

memory significantly improves the performance. The GPU kernel of motion estimation need to access neighboring pixels as Fig. 4.6 illustrates. Without prefetching those data to local memory, it would result multiple accesses to slow global off-chip memory. Our implementation loads a block of data simultaneously before performing pixel operations. This way, overall memory latencies are dramatically reduced.

GPU is efficient to run SIMD model instructions, however it is inefficient to perform tasks where tasks run different branches which is called branch divergence. In our implementation, there are divergence when the kernel load data from global memory to local memory. Because each block need data of the neighbor area, additional data is required to be loaded to local memory as Fig. 4.6 illustrated. In Fig. 4.6, every blue rectangle demonstrates a thread which runs motion estimation process for one block. For those thread in the middle area such as the green block, it can access the neighbor's data that have been loaded by its neighbor threads. However, for those thread in the border such as the orange threads, they

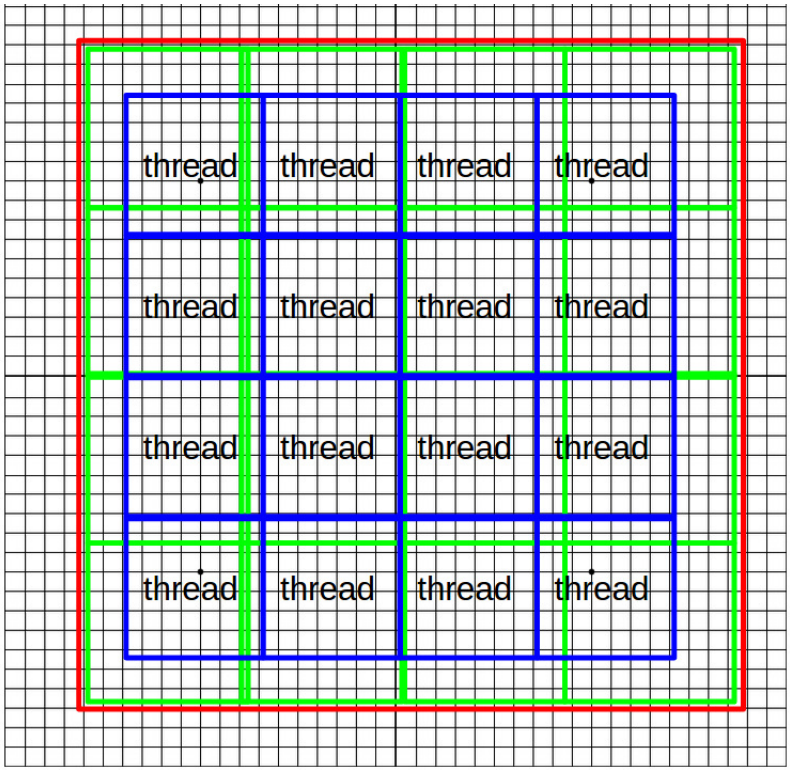


Figure 4.7. Divergence removal solution 1.

have no enough neighbors to help to load the data of neighbor area, therefore they need load these missing data by the thread itself. Which means the thread in the middle area and border area runs different instructions to load data from global memory to local memory. This will cause divergence.

In order to solve this divergence problem, each thread will load more data than a block. For example, if there are 4×4 threads in a group, the search range is 8, that means the data need to be loaded to local memory is $(16 \times 4 + 8) \times (16 \times 4 + 8)$, which means 80×80 pixels need to be loaded to local memory. If we want all threads to load same size data from global memory to local memory, each thread needs to load 20×20 pixels from global memory to local memory. This method is illustrated by Fig. 4.7.

Another way to solve this problem is to launch 5×5 thread to load data but only 4×4 thread to perform motion estimation tasks and the other 9 threads remains idle during the motion estimation process. As Fig. 4.8 illustrates, every thread of all 5×5 thread will

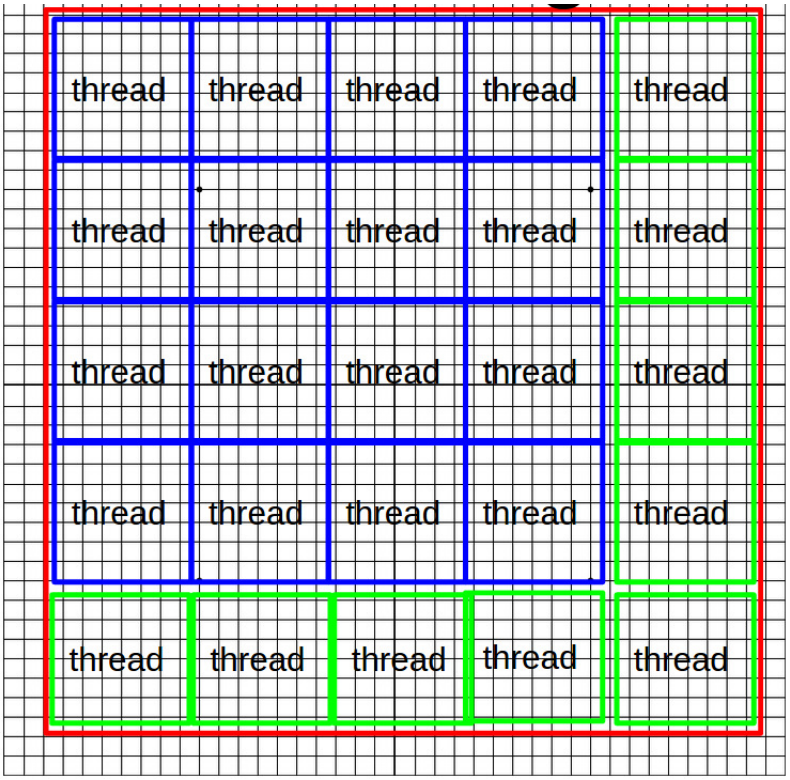


Figure 4.8. Divergence removal solution 2.

load a block data from local memory to local memory, but only blue threads will perform motion estimation and the green thread will be idle during the motion estimation process. This approach can solve the divergence problem but cause some GPU cores to be idle.

4.3.2.3 Bank Conflict Removal (Experiment 2.4)

Physical memory is implemented through a number of banks. If multiple threads access different banks, the accesses can be performed in parallel. However, if multiple threads access same bank, these accesses are serialized even if these threads are running in parallel. This is called *bank conflict*.

Bank conflict is another critical issue that cause inefficiency on GPU. When does bank conflict happen? Assume there are totally N banks in memory. These memory addresses of $x \times N + A$ where A is an address and x=0,1,2,3... are in the same bank. For example $2 \times N + A$ and $3 \times N + A$ are in the same bank. If one thread is accessing $2 \times N + A$ and

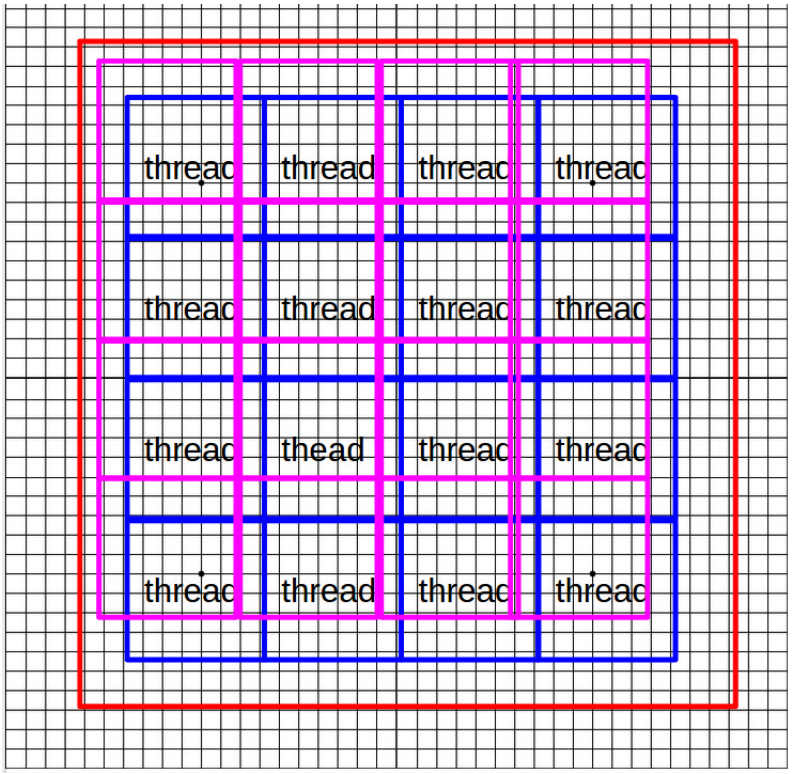


Figure 4.9. Bank conflict in OpenCL kernel.

other thread want to access $3 \times N + A$, the bank conflict happens.

If thread A is accessing bank X, and thread B also accesses bank X, these two accesses must be serialized, which means thread B must wait for the thread A to finish its memory accessing. If bank conflict happens, threads will perform memory accessing serially even if they are parallel threads and run simultaneously. This impacts the performance significantly.

In our experiments, bank conflict is a problem. Fig. 4.9 demonstrates that all thread in the same column accesses the same bank if the number of bank is 64. In order to remove the bank conflict, we change the search path for each row which is illustrated in Fig. 4.10. In the first row, the motion estimation algorithm will search the blocks from the most left points, the second row will shift the start search points one pixel different from the first row, and the third row and forth row use the same rule to shift the start points.

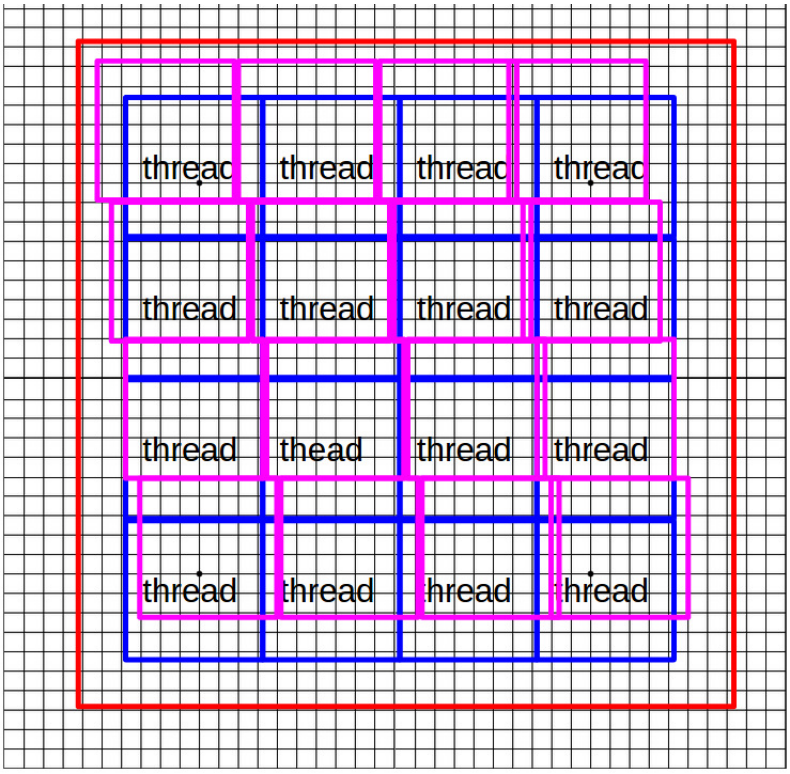


Figure 4.10. Reducing bank conflict in OpenCL kernel.

4.4 Our Proposed FS Algorithm on CPU (Experiment 3.1)

4.4.1 Implementation Details

Our new FS algorithm has the same framework as traditional FS algorithm. However, there are two key differences that make the algorithm more efficient. The first one is that our algorithm has the LBP encoding process and the second one is that our block matching is based on comparing the LBP code but not the pixel value. Fig. 4.11 illustrates the work flow of our new FS algorithm. Though this algorithm requires additional LBP encoding process, the LBP encoding process is executed for only one time even if the block matching process will be iterated for many times to match all possible candidate blocks. Therefore the amortized time consumption of motion estimation of each block is reduced.

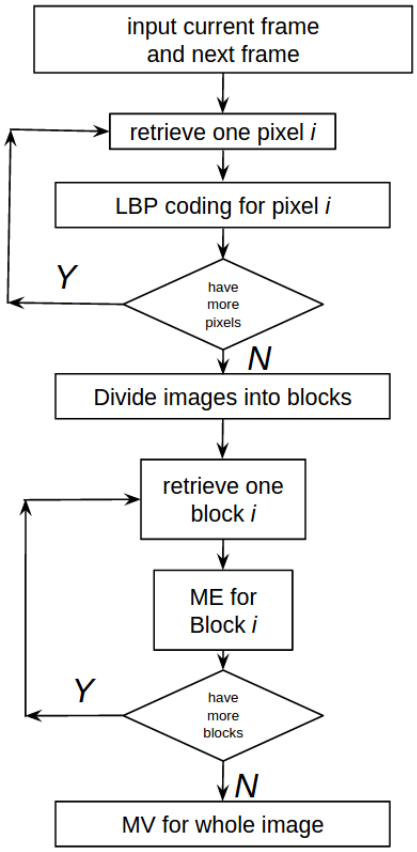


Figure 4.11. Implementation of FS algorithm with LBP encoding.

Fig. 4.12 demonstrates the implementation details of our LBP encoding process. Each pixel is compared with four neighbors including left, top, right and bottom neighbor, therefore each pixel has four LBP bits. Two adjacent pixels generate eight LBP bit code that is one byte long. Fig. 4.13 demonstrates the implementation details of LBP code based block matching method. In this implementation, every block matching just compares 16 long integers but not 256 pixels as in traditional methods.

```
void LBPencode(unsigned char * frame, unsigned char * encodedfrm,int framewidth, int frameheight)
{
        int row = 0,col = 0,x = 0,y = 0,evenflag = 0;
        unsigned char bitbyte = 0, prebitbyte = 0,bitflag = 0;

        for(row=0;row<frameheight;row++)
        {
                evenflag = 1;
                bitbyte = 0;
                prebitbyte = 0;
                bitflag = 0;
                for(col=0;col<framewidth;col+=2)
                {
                        bitbyte = 0;

                        y=row; x=col-1;//left
                        if(x< 0 || abs(frame[y*framewidth+x] - frame[row*framewidth+col])<10){
                                bitflag = 0;
                        }else{
                                bitflag = 1<<3;
                                bitbyte |= bitflag;
                        }
                        y=row-1;x=col;//top
                        if(y<0  || abs(frame[y*framewidth+x] - frame[row*framewidth+col])<10){
                                bitflag = 0;
                        }else{
                                bitflag = 1<<2;
                                bitbyte |= bitflag;

                        }
                        x=col+1;y=row;//right
                        if(x>=framewidth || abs(frame[y*framewidth+x] - frame[row*framewidth+col])<10){
                                bitflag = 0;
                        }else{
                                bitflag = 1<<1;
                                bitbyte |= bitflag;

                        }
                        x=col;y=row+1;//bottom
                        if(y>=frameheight || abs(frame[y*framewidth+x] - frame[row*framewidth+col])<10){
                                bitflag = 0;
                        }else{
                                bitflag = 1;
                                bitbyte |= bitflag;
                        }
                        if(evenflag%2 == 0){//hand the LBP code
                                bitbyte = bitbyte | prebitbyte;//set it to encoded
                                encodedfrm[(row*(framewidth/2))+(col/2)-1] = bitbyte;
                                prebitbyte = 0;
                        }else{
                                prebitbyte = (bitbyte <<= 4);//need the other half
                        }
                        evenflag++;
                }
        }
};
```

Figure 4.12. implementation of LBP encoding.

```
inline int compareTwoBlocksWithLBP(unsigned char * LBPcode1,unsigned char * LBPcode2,
                                   int framewidth, int frameheight,
                                   int b1x, int b1y, int b2x, int b2y)
{
        int i = 0;
        int diffcount = 0;
        for(i=0;i<16;i++){
                if((unsigned int)LBPcode1[(b1y+i)*(framewidth/2)+b1x/2]
                != (unsigned int)LBPcode2[(b2y+i)*(framewidth/2)+b2x/2]){
                        diffcount++;
                }
        }

        return diffcount;
};
```

Figure 4.13. Implementation of block matching based on LBP code.

4.5 Our Proposed FS Algorithm on GPU (Experiment 4.1)

4.5.1 Implementation Details

We further accelerate our new FS algorithm by offloading LBP encoding and block matching parts to GPU as Fig. 4.14 illustrates. For LBP encoding process, because each pixel is encoded independently, we can map pixel to OpenCL thread for LBP encoding. For motion estimation of each block, the reason is same as what we did in the experiment above by using GPU to accelerate original FS algorithm.

4.6 Comparison of Experiment Results

4.6.1 Evaluation Method

Accuracy and speed are evaluated in our experiments. For accuracy evaluation, we assume the baseline is the most accurate algorithm because of its brute force nature, and the second experiment, using GPU to accelerate the baseline, has the same accuracy as our baseline because the GPU computing does not change the algorithm but just changed the computing speed. However, the third and fourth experiments have different motion estimation results

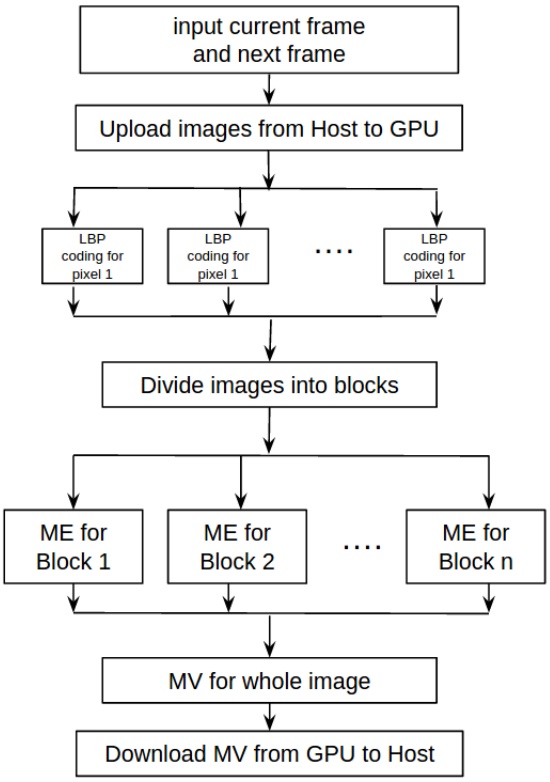


Figure 4.14. Implementation of FS algorithm with LBP encoding with GPU acceleration.

because they use our new proposed algorithm. Therefore, our accuracy evaluation is only used to evaluate our new algorithm by comparing the motion vectors with the baseline. For speed evaluation, we evaluate the performance by measuring the time of processing one motion estimation with two frames, reference frame and prediction frame.

We test images are obtained from Baker et al [2] which is a joint compute vision research group among Microsoft, Middlebury College, Brown University and TU Darmstadt. The test images contain 12 different clips each of which contains a sequence of video images. Our test evaluates all 12 clips and then computes the average time.

4.6.2 Evaluation Setup and Analysis

Our experiments are conducted on the platform consisting of Intel CPU i7-3770K and Nvidia GPU GTX680 with 16GB main memory. Table 4.1 illustrates our test results on speed of each experiments that discussed above. In table 4.1, there are four columns. The first

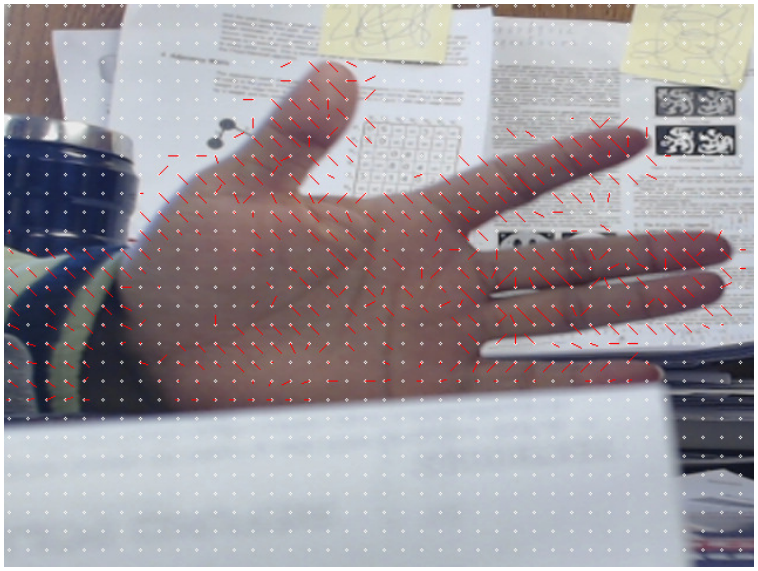
Table 4.1. Experimental results on speed (in millisecond (ms)).

Experiment Identity	Configuration explain	1080p	720p
Experiment 1.1	Baseline	240	101
Experiment 1.2	CPU with loop unrolling	208	87
Experiment 2.1	GPU with global memory	52	30
Experiment 2.2	GPU with loop unrolling	51	30
Experiment 2.3	GPU with local memory	26	13
Experiment 2.4	GPU bank conflict removal	25	12
Experiment 3.1	CPU with LBP	47	22
Experiment 4.1	GPU with LBP	5	3

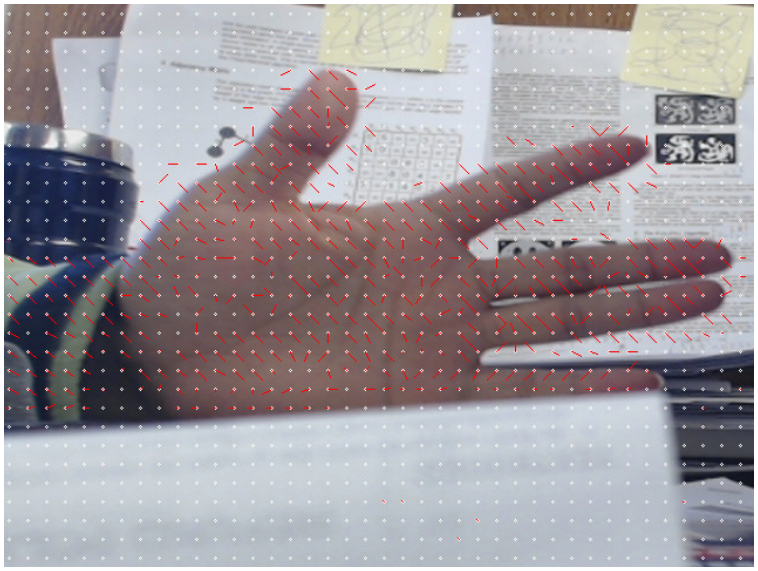
Table 4.2. Experimental results comparison.

Platform	Traditional Algorithm (ms)	Our Algorithm (ms)	Improvement (times)
CPU Only (1080p)	240	47	5.1
GPU-CPU (1080p)	25	5	5
CPU only (720p)	101	22	4.6
GPU-CPU (720p)	12	3	4

column is an experiment configuration identity, the second column is a simple description of experiment configuration. The third column is test result with 1080p images and the fourth column is test result with 720p images. The Fig. 4.15, 4.16, 4.17 and 4.18 demonstrate the visualization of motion estimation results. In terms of evaluation of accuracy, we compared the motion vectors that generated from baseline and our proposed algorithm, and 93 % motion vectors are same which means the accuracy is 93 %.



(a) Traditional Block Matching Algorithm.



(b) Our Block Matching Algorithm.

Figure 4.15. Visualization of our experiment examples on motion estimation - 1.

(a) Traditional Block Matching Algorithm.

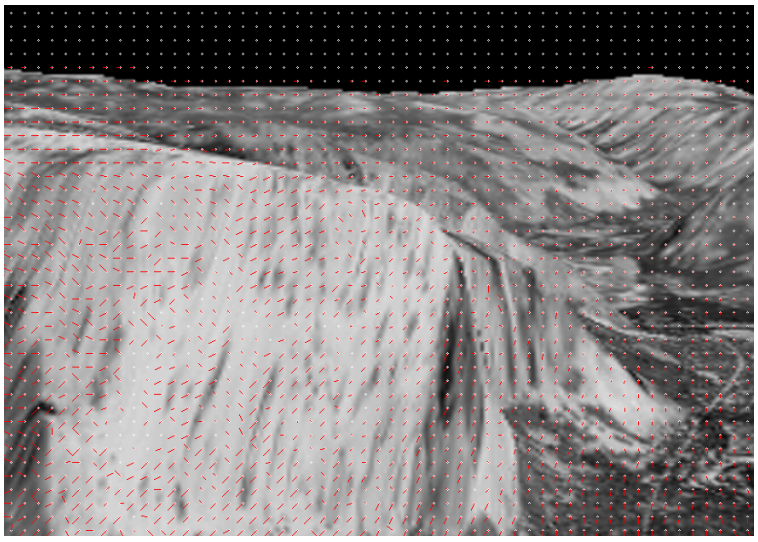
(b) Our Block Matching Algorithm.

Figure 4.16. Visualization of our experiment examples on motion estimation - 1.

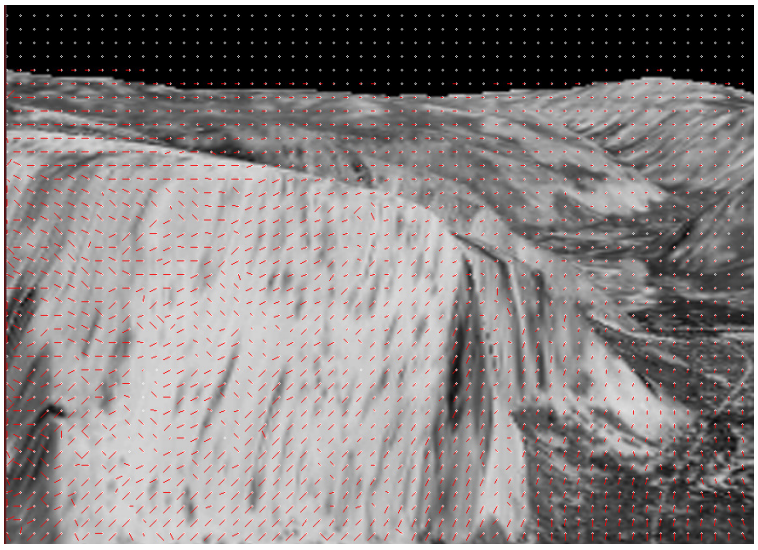
(a) Traditional Block Matching Algorithm.

(b) Our Block Matching Algorithm.

Figure 4.17. Visualization of our experiment examples on motion estimation - 1.



(a) Traditional Block Matching Algorithm.



(b) Our Block Matching Algorithm.

Figure 4.18. Visualization of our experiment examples on motion estimation - 1.

CHAPTER 5

CONCLUSION

This thesis summarizes our research efforts on designing and implementing a fast motion estimation. The thesis proposes an innovative LBP encoding based motion estimation algorithm which first encodes original raw video frames into LBP code and then executes block matching by using LBP code. The benefit of this approach is to avoid comparing pixels directly during block matching. One drawback of this algorithm is that the accuracy is impacted. Our experiments show that our new algorithm runs 5 times faster than traditional algorithm with same hardware configuration with 7% accuracy loss. However, our proposed algorithm opens up a new approach for motion estimation. We believe further improvement can address the accuracy issue in various ways. As for implementation, we implement the proposed algorithm on emerging GPU computing platforms where each GPU core processes the motion estimation processes for one block and all blocks are processed in parallel. Our research efforts also include GPU architecture aware optimizations. From the result of our experiments, the most optimized implementation runs 9 times faster than the original CPU based implementation.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. Face description with local binary patterns: Application to face recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(12):2037–2041, 2006.

[2] Simon Baker, Daniel Scharstein, JP Lewis, Stefan Roth, Michael J Black, and Richard Szeliski. A database and evaluation methodology for optical flow. *IJCV*, 92(1):1–31, 2011.

[3] Aroh Barjatya. Block matching algorithms for motion estimation. *IEEE Transactions Evolution Computation*, 8(3):225–239, 2004.

[4] Wei-Nien Chen and Hsueh-Ming Hang. H. 264/avc motion estimation implmentation on compute unified device architecture (cuda). *Multimedia and Expo, 2008 IEEE International Conference on*, pages 697–700, 2008.

[5] P Davis and Sangeetha Marikkannan. Implementation of motion estimation algorithm for h. 265/hevc. *IJAREEIEE*, 2014.

[6] Muhammad Usman Karim Khan, Muhammad Shafique, Mateus Grellert, and Jorg Henkel. Hardware-software collaborative complexity reduction scheme for the emerging hevc intra encoder. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 125–128, 2013.

[7] Jong-Nam Kim and Tae-Sun Choi. A fast three-step search algorithm with minimum checking points using unimodal error surface assumption. *Consumer Electronics, IEEE Transactions on*, 44(3):638–648, 1998.

[8] John P Lewis. Fast template matching. 95(120123):15–19, 1995.

[9] Renxiang Li, Bing Zeng, and Ming L Liou. A new three-step search algorithm for block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(4):438–442, 2002.

[10] Yih-Chuan Lin and Shen-Chuan Tai. Fast full-search block-matching algorithm for motion-compensated video compression. *Communications, IEEE Transactions on*, 45(5):527–531, 1997.

[11] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *Ieee Micro*, 28(2):39–55, 2008.

[12] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[13] Gary J Sullivan, Jens Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, 2012.

VITA

Zhaohua Yi was born in Yueyang City, Hunan Province, China, on Mar 24, 1980. He received Bachelor Degree of Electromechanical Engineer from Beijing Technology and Business University in 2002 and Master Degree of Software Engineering from Tsinghua University in 2005. Before join University of Mississippi for this second Master Degree, Zhaohua worked in Intel China for eight years from 2005 to 2013.