A high-performance parallel implementation of Sum of Absolute Differences algorithm for motion estimation using CUDA

Sanyam Mehta, Arindam Misra, Ayush Singhal
Indian Institute of Technology, Roorkee, Uttarakhand, INDIA
{ san01uec, ari07uce, ayushuec,}@iitr.ernet.in

Praveen Kumar
Dept. of Computer Science, GRIET, Hyderabad.
praveen.kverma@gmail.com

Ankush Mittal
College of Engineering Roorkee
Roorkee, Uttarakhand, INDIA
dr.ankush.mittal@gmail.com

ABSTRACT
The Sum of Absolute Differences (SAD) algorithm for motion estimation constitutes one of the most time consuming and compute intensive part of image registration in Automated Video Surveillance (AVS) using non-stationary cameras and H.264 video compression. Many present solutions, implemented by use of dedicated hardware for video surveillance are incapable of delivering image registration for larger sized video streams in real-time. In this paper, we present a novel, high performance implementation of SAD on the general purpose GPU architecture, which is available in many laptops and PCs, using NVIDIA's CUDA. The proposed implementation contains innovative and efficient optimizations to overcome the bottlenecks that limit the achievable speedup and outperforms the other parallel implementations not only in terms of speed but also accuracy. The use of threshold in the SAD computation prevents detection of false motion caused due to noise. The proper choice of macro block size for SAD computation provides more accurate results with significant reduction in execution times. This makes the motion estimation real-time which scales well for larger image sizes. The wide range of memories offered by the GPU architecture has been exploited to a large extent and a significant speedup of 204X for an image size of 1024×768 was achieved for SAD on the GeForce GTX 280 as compared to the serial implementation. The use of texture memory for accessing the periphery of the reference frame prevents the out of bound references and contributes to the speedup achieved.

Keywords
GPU, Motion estimation, Sum of Absolute Difference, Image Registration.

1. INTRODUCTION
The unprecedented growth in the amount of video content from various video capturing devices and the Internet has aroused a fresh interest in the video processing technology[1][2].The tracking of moving objects using non-stationary cameras requires video frame registration[3][4], which uses motion estimation. It plays a crucial role in many situations, like airborne and other on-the-move surveillance scenarios. Block-based motion estimation uses block matching, which employs the SAD as one of the most frequently used criteria [5][6]. The high complexity of these state-of-the-art algorithms makes them very difficult to be implemented on an ordinary CPU in real-time, which poses a major challenge to the developers. This calls for a High Performance Computational (HPC) solution to this problem that should be able to handle large amounts of data and achieve real-time object tracking and scales well for different input sizes.

As a major part of motion estimation is pure SAD computation, [5] improvement in its performance would substantially reduce the time taken for block-based motion estimation.

Fast and robust object tracking in video sequences is required by many applications. These methods are used by Video Surveillance systems to track objects, robots rely on them to perform navigation tasks or interacting with human beings, augmented reality systems depend on the position data acquired by visual tracking to place the virtual objects in their world, video-games or assisted devices can be controlled by hand or face tracking softwares that use motion estimation.

Most of multimedia streaming applications need motion picture encoding and decoding. Motion estimation is one of the most important processes in motion picture coding, as the achievement of a high compression ratio depends on how well the motion estimation is accomplished [7].

The recent developments in the GPU architecture have provided an effective tool to handle the workload. The GPUs are massively parallel unified shader designs that have a much higher computational capability than the CPUs and the video bandwidth between GPU and the video memory is about five times faster than that between CPU and system memory, the NVIDIA GeForce GTX 280 for example, has a single precision floating point capability of 933 GFlops and a memory bandwidth of 141.7 Gb/sec. CUDA enables new applications with a standard platform for extracting valuable information from vast quantities of raw data. It enables HPC on normal enterprise workstations and server environments for data-intensive applications [10]. Moreover, the GPUs are quite ubiquitous as many laptops and desktops have GPUs, which can be used for running the application. Previously, some GPU based motion estimation methods have been proposed, [8][11] but the results have not been able to achieve real-time motion estimation by achieving a frame processing rate of 24-30 frames per second (fps) for large video sizes. In this paper we present the parallel implementation of the SAD algorithm for motion estimation using block matching, exploiting the thread configurations and different memory types offered by the GPU architecture to a large extent. The use of threshold in SAD computation adds to the accuracy of motion estimation. It outperforms other implementations not only in terms of speed but also accuracy. A significant speedup of 204X for an image size of 1024×768 was achieved for SAD on the GeForce GTX 280 as compared to the serial implementation. The use of texture memory for accessing the periphery of the reference frame prevents the out of bound references and contributes to the speedup achieved. The scalability was tested by executing different frame sizes on both the GPUs for macro block sizes of 8×8 as well as 16×16.

2. GPU Architecture and Related Work

NVIDIA’s CUDA is a general purpose parallel computing architecture that leverages the parallel compute engine in NVIDIA Graphic Processor Units (GPU) to solve many complex computational problems. The programmable GPU is a highly parallel, multithreaded, many core co-processor specialized for compute intensive highly parallel computation. Since CUDA is geared towards fine-grained parallelism, it often works well for the highly data parallel applications which we often find in video processing.

The three key abstractions of CUDA are the thread hierarchy, shared memories and barrier synchronization, which render it as only an extension of C. All the GPU threads run the same code and, are very light weight and have a low creation overhead. A kernel can be executed by a one dimensional or two dimensional grid of multiple equally-shaped thread blocks. A thread block is a 3, 2 or 1-dimensional group of threads. Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. Threads in different blocks cannot cooperate and each block can execute in any order relative to other blocks. The number of threads per block is therefore restricted by the limited memory resources of a processor core. On current GPUs, a thread block may contain up to 512 threads. The multiprocessor SIMT (Single Instruction Multiple Threads) unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.

The device memory space consists of various types of memories, as shown in Figure. 1, each thread has a local memory and registers; also each block has its shared memory. Besides this the constant and the texture memory allocated for a grid is read only. The local memory resides in global memory allocated by the compiler and delivers the same performance as any other global memory region. The shared memory is on chip and the accesses are 100x-150x faster than accesses to local and global memory. The shared memory, for high bandwidth, is divided into equal sized memory modules called banks. But, if two memory requests fall in the same bank, then the access is serialized, thus reducing the bandwidth. For devices of compute capability 1.x, the warp size is 32 and the number of banks is 16, the shared memory requests are split into two halves for a warp and hence there can be no bank conflicts between threads belonging to two different halves of the warp.

The problem of motion estimation using SAD on parallel architectures has been addressed by many earlier works like [8][11]. Chen et.al in [11] divide the macroblock into sub blocks, which cannot optimally use shared memory, and processing each pixel on each thread doesn’t give much speed up as the number of blocks are increased. Their approach also involves merging of the SAD values obtained. Lee et.al in [8] proposed a multipass motion estimation algorithm for GPU, which generates local and global SAD values in the first and the second passes respectively. In other works, the implementations are not suited to the optimal utilization of CUDA memories. Thus, our implementation takes care of all these factors.

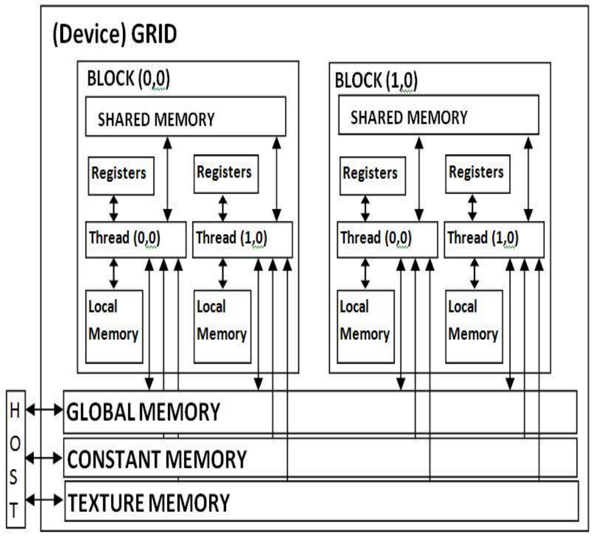

Figure 1 The device memory space in CUDA.

3. The Sum of Absolute Difference Algorithm

The SAD algorithm is widely used method for motion estimation, which finds widespread application in areas like image registration for Automated Video Surveillance and video compression using H.264[4][5], it forms the most computationally intensive part of image registration as well as video compression. In the process of video coding, the similarities between video frames could be used to achieve higher compression ratios. The usual coding techniques applied to moving objects within a video scene lower the compression efficiency as they only consider the pixels at the same position in the video frames. Motion estimation and the SAD algorithm are used to capture such movements more accurately for better compression efficiency. In video surveillance using moving cameras, a popular way to handle translation problems on images, using template matching is to compare the intensities of the pixels, using the SAD measure.

The motion estimation on a video sequence using SAD uses the current video frame and a previous frame as the target frame. The two frames are compared pixel by pixel, summing up the absolute values of the differences of each of the two corresponding pixels. The result is a positive number that is used as the score. SAD reacts very sensitively to even minor changes within a scene.

The block based motion estimation is performed on a set of pixels, every frame is divided into equally sized blocks, and for each block in the current frame, a search for most resembling block is done in the reference frame, searching the whole reference frame for each block in the current frame makes this task computationally intensive [5]. After the best matches are found for the current block the motion vectors are stored along with the SAD values.

The basic unit of this method is a macro block of size 16×16 or 8×8 on the previous frame (*reference frame*), which is taken as the reference to gauge the direction of motion. SAD algorithm is usually implemented in a nested loop with conditional branch.

Student Authors: Sanyam Mehta, Arindam Misra, Ayush Singhal. B.Tech Final Year, CSE, IIT Roorkee.

```
for(rows of macro blocks)
  {  for(columns of macro blocks)
       {  for(rows of template)
            {  for(columns of template)
                    {
                    SAD computation;
                    SAD comparison; (>threshold)
                    }
            }
```

Figure 2 The Sum of Absolute Differences algorithm

Besides the calculation of SAD there are four loops in the algorithm as depicted in Figure. 2. It seeks the best matched macro block in the search region of the reference image frame to the current macro block in the current image frame. The best matched macro block is the one which has the minimum SAD value with the current macro block.

The algorithm depicted in Figure 2 is computationally intensive as very large number of calculations and comparisons are required. However, as it is a block based algorithm, it is a good candidate for being parallelized and implemented on parallel architectures like the GPU architecture.

Let $I_{ref}(x_r, y_r)$denote the pixel intensity of the pixels in the reference image and $I_{curr}(x_c, y_c)$denote the pixel intensities in a macro block of the current frame then the SAD value of the pixel intensities is given by (1).

$$SAD\ (x,y) = \sum_{i=0}^{T_{rows}} \sum_{j=0}^{T_{cols}} \left| I_{ref}(x_r + i, y_r + j) - I_{curr}(i,j) \right| \quad (1)$$

Since several hundred repeated calculations of SAD are necessary in order to find the best matched macro block, it is important to seek an efficient implementation of SAD algorithm.

3.1 Our approach to SAD

In our approach to the SAD, the image is divided into macro blocks, with the SAD computation for each macro block being handled by a thread. Thus for an image size of 1024×768 the thread configuration for a macro block size of 8×8 was 128×96 threads, and 64×48 for a 16×16 macro block, with each thread performing the computation for its respective macro block. In the reference frame a 32×32 (or 16×16) region called the window, is selected and the sum of absolute differences (SAD) is computed in this window frame, with the macro block of the current frame i.e. one by one all the possible 16×16 (or 8×8) blocks on the 32×32 (or 16×16) window frame are taken for SAD computation on each pixel. The frame in which the SAD turns out to be the minimum is selected and determines the motion of the object from the reference frame to the current frame.

In the following section we describe the approach taken for the image size of 256×191. For this image size we have used 8×8 as the macro block size so shared memory could be used effectively, and portions of both the frames that were required for the SAD computation could be accessed from the shared memory which is limited to 64 kilobytes per block. As the computation of SAD on each pixel refrains from accessing memory locations within the same banks, there were lesser bank conflicts. As the shared memory is limited, the portions of the current and the reference frames required for SAD computation of the macro blocks could not be accommodated in the shared memory, to overcome this problem the remaining portion was accommodated in the texture memory, similar to the way shown in Figure 3 for an image size of 1024×768. The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together achieve best performance. This helped us to remove the memory latency involved in SAD calculation. Also, as the out of bound memory accesses are clamped by the texture memory, therefore we were able to effectively perform SAD computation on the peripheral pixels of the reference frame, which added to the accuracy and speedup achievable. In our approach only the SAD values above a threshold were considered, so that motion caused due to noise is ignored. Moreover, the choice of the macro block size led to the added accuracy in the results, which gave more precise motion vector estimates for the image size of 256×191 as shown in Figure 4.

Thus, through the above mentioned approach we were able to make the SAD calculations faster by 204 times on the GTX 280 for an image size of 1024×768, as compared to the serial implementation. A reasonable speedup of 11X was achieved on the GTX 8400GS.

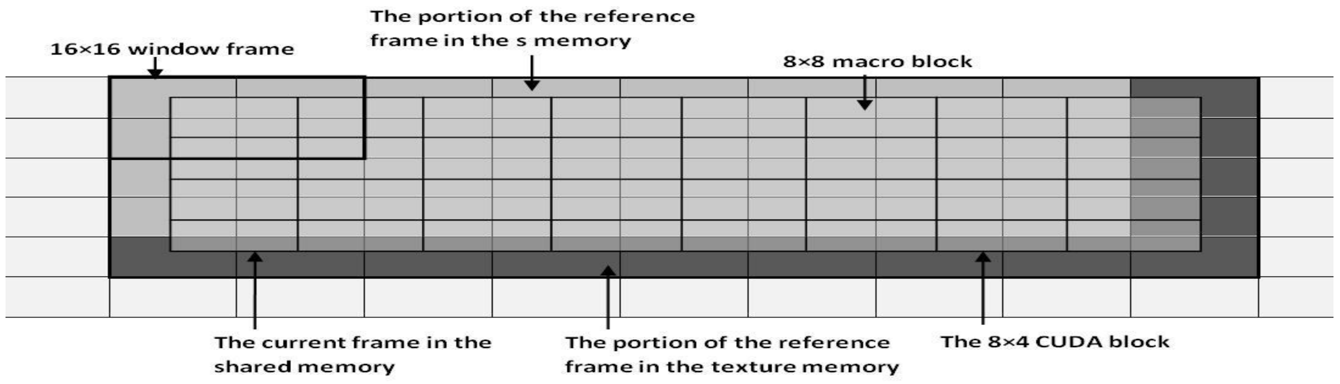
16×16 window frame
The portion of the reference frame in the s memory
8×8 macro block
The current frame in the shared memory
The portion of the reference frame in the texture memory
The 8×4 CUDA block

Figure 3 Our Approach to SAD computation

Macro block size	Occupancy	Grid Size	Block Size	Total Branches	Divergent Branches	Shared Memory (bytes)	Total Global memory Loads	Total Global Load Request
8×8	0.094	16×24	8×4	49224	7788	4124	23680	768
16×16	0.031	8×12	8×4	14865	4	8220	39168	768

Table 1 CUDA Profiler output for 1024×768 image on the GTX 280

4. Experimental Results

The parallel version of SAD was executed on the NVIDIA GeForce GTX 280 and the NVIDIA GeForce 8400GS GPUs. The GTX 280, having a single precision floating point capability of 933 GFlops and 1GB dedicated DDR3 video memory, was on board a 3.2 GHz Intel Xeon machine. The GTX 280 has 30 streaming multiprocessors with 8 cores each, hence at total of 240 stream processors. It belongs to the compute capability 1.3 which supports advanced features like page-locked host memory and those which take care of the alignment and synchronization issues.The 8400 GS has two streaming multiprocessors with 8 cores each, i.e. 16 stream processors, with a single precision floating point capability of 28.8 GFlops and 128 MB of dedicated video memory. An 8400GS on board a 2.0GHz Intel Centrino Duo machine was used. It belongs to the compute capability 1.2. The development environment used was Visual Studio 2005 and the CUDA profiler version 2.2 was used for profiling the CUDA implementation. The image sizes that have been used are 1024×768, 640×480 and 256×191.

Image Size	Macro block size	Execution time (ms)
1024×768	16×16	40.63
1024×768	8×8	14.23
640×480	16×16	19.06
640×480	8×8	5.11
256×191	8×8	2.28

Table 2. Execution times for GeForce GTX 280

Image Size	Macro block size	Execution time (ms)
1024×768	16×16	779.23
1024×768	8×8	344.78
640×480	16×16	292.64
640×480	8×8	136.37
256×191	8×8	24.53

Table 3. Execution times for GeForce 8400GS

The SAD algorithm is computationally intensive and hence was suitable for implementation on the GPU architecture. Besides this, being a block based algorithm, it offered a pixel level parallelism that was exploited in our implementation. The host of memories offered by the CUDA architecture were exploited to hide the memory latency. The use of texture memory allowed for computing SAD on the peripheral pixels of the reference frame due to the *cudaAddressModeClamp* addressing mode of the texture memory, which clamps the out of bound memory accesses to valid addresses. Consequently, we were able to achieve significant speedup in the GPU implementations, as compared to the serial implementation.The parallel codes for SAD were executed on both the GPUs for practical comparison. The execution times for SAD on the GTX 280 are shown in Table 2 and those for the 8400 GS are shown in Table 3. The execution times indicate that the implementations scaled well for larger image sizes on both the GPUs, for an image size of 1024×768 we were able to achieve an execution time of 40.63ms on the GTX 280 and 8309ms on serial execution, indicating a significant speedup of 204X. The image size of 256×191 is best for use in motion estimation and the proposed implementation promises improved utility in real-time motion estimation, even on the 8400GS. Figure 4 shows the motion vectors obtained by SAD computation on the two frames. The profiler outputs of the GTX 280 for the 1024×768 image with a macro block size of 8×8 as well as 16×16 is shown in Table 1.

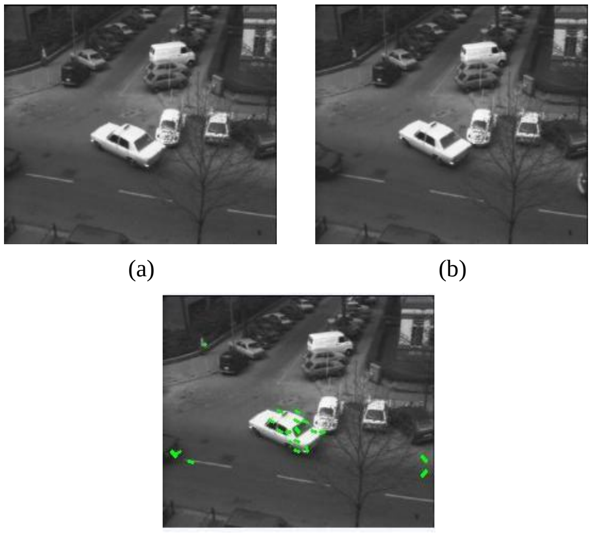
(a) (b) (c)

Figure 4 (a) Reference Frame (b) Current Frame (c) Output with motion vectors

As each thread performs the SAD computation for the entire macro block, the occupancy remains low, especially in the case of the size 16×16. However, in case of 8×8, as shown in Table 1 the occupancy became almost three times that of the 16×16 implementation, owing to larger grid size and hence better utilization of the streaming multiprocessors. The total number of branches that became divergent in the 16×16 implementation was a very small fraction of the total number of branches. In the 8×8 implementation although, the total number of divergent branches are higher, the execution time is lower as well as the occupancy is higher, as compared to the 16×16 implementation largely because of the better use of shared memory and larger grid size. The total number of warps serialized in the 8×8 implementation was less as compared to the 16×16 implementation, due to the lesser bank conflicts within the threads. Due to these factors we were able to achieve a significant speedup of 204X on the GTX 280 for 1024×768 image size, and a reasonable execution time of 24.53ms for the image size of 256×191 on the 8400GS, which meets the real time requirements.

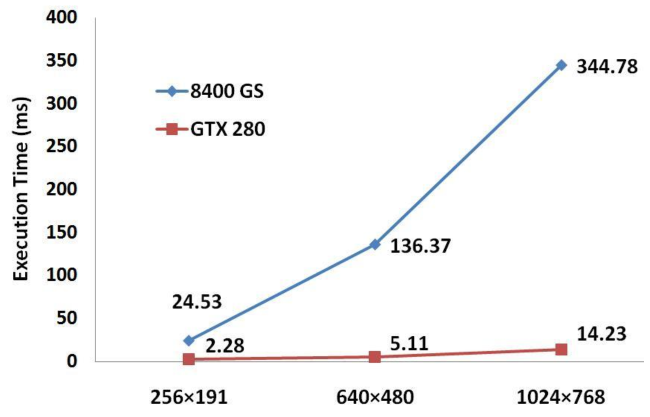

Figure 5 Comparison between the execution times on GTX 280 and 8400GS for various image sizes, using the 8×8

Figure 5 shows a graph indicating the comparison of execution times for various image sizes on the GTX 280 as well as the 8400GS, which shows the scalability of the proposed implementation.

5. Conclusion and Future Work

SAD for motion estimation constitutes one of the most time consuming and compute intensive parts of image registration in Automated Video Surveillance (AVS) using non-stationary cameras as well as H.264 video compression. SAD is block based and offers pixel level parallelism that can be very well exploited in the GPU architecture.

Through this paper, we present a novel, high performance implementation of SAD on the general purpose GPU architecture, using CUDA which makes the motion estimation real-time and scales well for larger image sizes. The proposed implementation outperforms the other parallel implementations not only in terms of speed but also accuracy .The wide range of memories offered by the GPU architecture has been exploited to a large extent resulting in a significant speedup of 204X for SAD on the GeForce GTX 280. Efficient usage of the different kinds of memories offered by the CUDA architecture and subsequent experimental verification resulted in the most optimal implementations. As a result, significant speedup was achieved.

In the future we intend to explore other block based motion estimation techniques like the Fast Normalized Cross Correlation on the GPU architecture using CUDA and compare the portability of both the algorithms on the GPU architecture.

6. REFERENCES

[1] Lin, D.; Xiaohuang Huang; Quang Nguyen; Blackburn, J.; Rodrigues, C.; Huang, T.; Do, M.N.; Patel, S.J.; Hwu, W.-M.W.; , "The parallelization of video processing," Signal Processing Magazine, IEEE , vol.26, no.6, pp.103-112, November 2009.

[2] M.K. Bhuyan, Brian C. Lovell, Abbas Bigdeli, "Tracking with Multiple Cameras for Video Surveillance," dicta, pp.592-599, 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications (DICTA 2007), 2007

[3] Arvind Kandhalu, Anthony Rowe, Ragunathan Rajkumar, Chingchun Huang, Chao-Chun Yeh, "Real-Time Video Surveillance over IEEE 802.11 Mesh Networks," rtas, pp.205-214, 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium, 2009

[4] Praveen Kumar, Kannappan Palaniappan, Ankush Mittal and Guna Seetharaman. “Parallel Blob Extraction using Multicore Cell Processor”. Advanced Concepts for Intelligent Vision Systems (ACIVS) 2009. LNCS 5807, pp. 320–332, 2009.

[5] Vanne, J.; Aho, E.; Hamalainen, T.D.; Kuusilinna, K.; , "A High-Performance Sum of Absolute Difference Implementation for Motion Estimation," Circuits and Systems for Video Technology, IEEE Transactions on , vol.16, no.7, pp.876-883, July 2006

[6] Rehman, S.; Young, R.; Chatwin, C.; Birch, P.; , "An FPGA Based Generic Framework for High Speed Sum ofAbsolute Difference Implementation," European Journal of Scientific Research, vol.33, no.1, pp. 6-29, 2009

[7] N.A. Khan, S. Masud, A. Ahmad, “A variable block size motion estimation algorithm for real-time H.264 video encoding”, Signal Processing: Image Communication, vol.21, no.4, pp. 306-315, April 2006

[8] Chuan-Yiu Lee; Yu-Cheng Lin; Chi-Ling Wu; Chin-Hsiang Chang; You-Ming Tsao; Shao-Yi Chien; , "Multi-Pass and Frame Parallel Algorithms of Motion Estimation in H.264/AVC for Generic GPU," Multimedia and Expo, 2007 IEEE International Conference on , vol., no., pp.1603-1606, 2-5 July 2007

[9] Boyer, M.; Tarjan, D.; Acton, S.T.; Skadron, K.; , "Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors," Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on , vol., no., pp.1-12, 23-29 May 2009

[10] Kung, M.C.; Au, O.C.; Wong, P.H.W.; Chun Hung Liu; , "Block based parallel motion estimation using programmable graphics hardware," Audio, Language and Image Processing, 2008. ICALIP 2008. International Conference on , vol., no., pp.599-603, 7-9 July 2008

[11] Wei-Nien Chen and Hsueh-Ming Hang, 2008 "H.264/AVC motion estimation implementation on compute unified device architecture (CUDA)" in conf. ICME 2008. National Chiao-Tung University, Taiwan