Exploring NVIDIA-CUDA for Video Coding

Aleksandar Colic
Dept. of Electrical and Computer Engineering and Computer Science
Florida Atlantic University
Boca Raton, FL 33431
+1 561 287 3885
acolic@fau.edu

Hari Kalva
Dept. of Electrical and Computer Engineering and Computer Science
Florida Atlantic University
Boca Raton, FL 33431
+1 561 287 3885
hkalva@fau.edu

Borko Furht
Dept. of Electrical and Computer Engineering and Computer Science
Florida Atlantic University
Boca Raton, FL 33431
+1 561 287 3885
bfurht@fau.edu

ABSTRACT
Today, world is rapidly turning to high definition multimedia. From engineering and programming point of view, this usually means more computation is needed and more memory space is required to achieve these higher qualities. In this paper we explore the use of parallelization opportunities in graphics processors to accelerate video encoding. We evaluate the NVIDIA CUDA[1] toolkit and evaluate the performance of motion estimation in video encoding. The main goal of this paper is to evaluate the capabilities of NVIDIA/CUDA and develop a process for implementing video/multimedia applications. We have discovered that the difference in performance when CUDA is not used properly can be over 100x. We show how we were able to use CUDA capabilities to reduce the motion estimation time from 7000 milli seconds to 70 milli seconds.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming.* B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids.

General Terms
Algorithms, Management, Measurement, Documentation, Performance, Experimentation.

Keywords
Compute Unified Device Architecture (CUDA), Graphics Processing Unit (GPU), Video coding, motion estimation, Parallel Processing.

MMSys'10, February 22–23, 2010, Phoenix, Arizona, USA.

1. INTRODUCTION
In this paper we present the application of NVIDIA CUDA toolkit for video coding applications [1]. The CUDA toolkit makes develop parallel programs on the NVIDIA GPUs easier and enables access to the large scale parallelism offered by the GPU. Video processing applications are especially suited for the NVIDIA platform because of the data parallel nature of the problem. In this paper we explore the tools available in CUDA and apply them tot he motion estimation problem. Motion estimation is widely studied and is the computationally expensive part of video coding. Working on a well understood domain allows us to explore the CUDA features better and allows readers to understand the intricacies of CUDA better.

1.1 Motion Estimation
Motion estimation is a widely known process. It is explained briefly in this section. Motion estimation is a computationally expensive process and is a key component for video processing operations such as encoding, segmentation and edge detection. The motion estimation engine that we are developing is part of the larger project for optimizing encoder implementation for High Efficiency Video encoding projects that are currently under way in MPEG.

Many fast motion estimation algorithms have been proposed in [2], but they may not be suitable for optimal used in CUDA. Thus, in this paper we focus on exploring and optimizing basic integer full search and exhaustive full search motion estimation algorithms. Motion estimation engine developed implements full search algorithm that exhaustively searches for a block match in a previous frame. The best match is selected based on the minimum sum of absolute differences. A half-pixel search algorithm is also implemented using the H.264 6-tap filter for half-pixel interpolation. The goal of this paper is not to develop a better motion estimation algorithm but to understand CUDA in the context of the well understood motion estimation problem.

1.2 CUDA architecture
Although a few GPU-based motion estimation methods have been proposed [3-6], CUDA architecture needs a new algorithm to fully utilize its features [7]. In the quest for maximum speed, NVIDIA's GPUs (Graphics Processing Units) have evolved far beyond single processors [8]. Modern NVIDIA GPUs are not single processors but rather are parallel supercomputers on a chip that consist of very many, very fast processors. Contemporary NVIDIA GPUs range from 16 to 256 stream processors per card, delivering incredibly powerful computing bandwidth. Those GPU

boards have become so powerful that the scientific computing community has begun using them for general purpose computing. It turns out that many mathematical computations, such as matrix multiplication and transposition, which are required for complex visual and physics simulations in games are also exactly the same computations that must be performed in a wide variety of scientific computing applications.

Because of this, CUDA has been developed - to allow application developers to write code that can be uploaded into an NVIDIA-based card for execution by NVIDIA's massively parallel GPUs. This allows applications developers to plug in a 500 gigaflop, 256-processor, NVIDIA-based card and upload applications to run within the NVIDIA GPU at far greater speed than possible on even the fastest general purpose CPU on the motherboard. A comparison of computing capabilities between GPUs and CPUs are given in the figure 1, taken from [1].

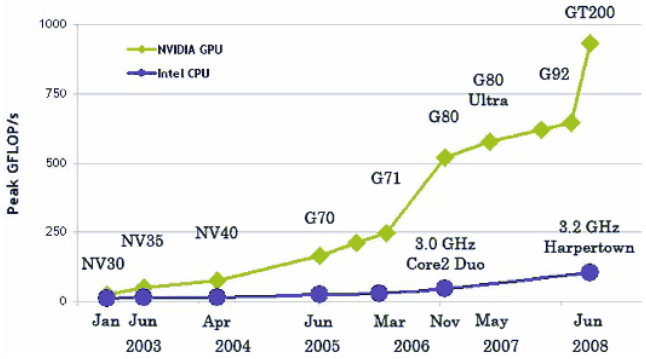

Figure 1.Compared compute capability measured in GFLOPs

CUDA allows developers to use C as a high-level programming language. Using already well established high level language and subtly adding additional, CUDA specific features (implemented in a way that they follow C programmer's way of thinking as close as possible) provides extremely user friendly environment.

GPU is specialized for compute-intensive, highly parallel computation because of the sole purpose of the GPU in the first place – graphics rendering. Because of that GPU and CPU differs in a way that more GPU transistors are devoted to data processing rather than data cashing and flow control. This makes them excellent candidates to be used to solve highly arithmetically intensive problems. And when we take into consideration that CUDA enabled GPU's are many core systems, conclusion arises that the CUDA architecture is best suited for solving highly arithmetic problems that can be done on many data elements in parallel – data parallelism. Or differently said, CUDA architecture provides best performance when every available core performs same set of arithmetic computation on a different data element. If there are no data dependencies between data elements, true parallelism can be accomplished – every core doing its work independently of one another just on a different set of data elements.

2. CUDA ARCHITECTURE

CUDA comes with user friendly environment, and one of the most important features is ability to write code that can be executed on a device mainly focusing on what the program is doing not how hardware is organized. Of course there are restrictions and rules needed to be followed by software because of the hardware limitations. Because of that, both software and hardware organization will be described shortly.

2.1 Software implementation

When a CUDA programmer starts developing CUDA software he needs to understand what is different from regular C code and what is similar [9]. Biggest difference is that CUDA code is divided into two parts, host code and device code. Host code is a code that runs on CPU, and it is in charge of everything, including when to run a device code. Device code is code that will run on GPU. Device code is organized as a function with a special name – kernel and special way of declaring. That is about all the difference between regular C function and CUDA function. As long as developer follows these special rules for declaring CUDA functions, he can simply regard them as a regular C function.

CUDA function or kernel represents portion of code that will be executed on GPU device. Since idea is to perform some calculations represented in kernel on different, mutually independent data elements basically that kernel will be executed as many times as there are data elements needed to be processed.

So far this seems similar as running functions on CPU using simple **for** loop for example. But since there are many cores available on the GPU, every available core will do all calculations for different data element, and since calculations are on different data element they are done in parallel. For example, if there is a need to process K data elements, K kernel executions have to happen to process all of them. If GPU has N cores that means that N out of K kernels can be executed in parallel. So kernels become independent threads assigned to some core to be executed. Developer has to know how many threads he needs to process all the data before calling his kernel. Because number of threads needs to be specified there are special arguments assigned to every kernel, and there lies the difference between normal C function and kernel, in those additional arguments needed to be provided.

Every thread has to know on what data element it has to work on. Usually data elements that need to be processed in these situations are arranged in one dimensional arrays or more dimensional arrays. Since that is the case, simplest and most practical approach would be to organize threads in a way to closely match organization of data elements. That way thread would fetch appropriate data element, depending on its position among other threads. Every thread is provided with group of variables that tells that thread its ID.(position in thread organization)If both data element ID and thread ID matches, or closely matches(meaning that it is easy to compute data element ID using thread ID) than choosing right data element for thread becomes easy or in some cases trivial.

General organization of the treads is: threads are grouped in a thread blocks, thread blocks are grouped in a grid of thread blocks. For user convenience threads within a block can be organized in a one, two or three dimensional way. Tread blocks can be also organized in such a way, although three-dimensional organization of grid is mentioned as existing by NVIDIA, but still not possible to implement.(as discovered while exploring different CUDA options for these paper) We are quite sure that in later improvements that dimension will be available because adding one more dimension can open completely new development horizons to a developer and make his life much easier. For now,

because of the hardware organization there are some limitations concerning number of threads per block – 512 maximum threads in one block, and 65535 blocks in a block grid. Calculating maximum number of threads that GPU can manage per one kernel call gives incredible number of 33553920 threads. One simple example how threads can be organized is given on the figure 2 [1].

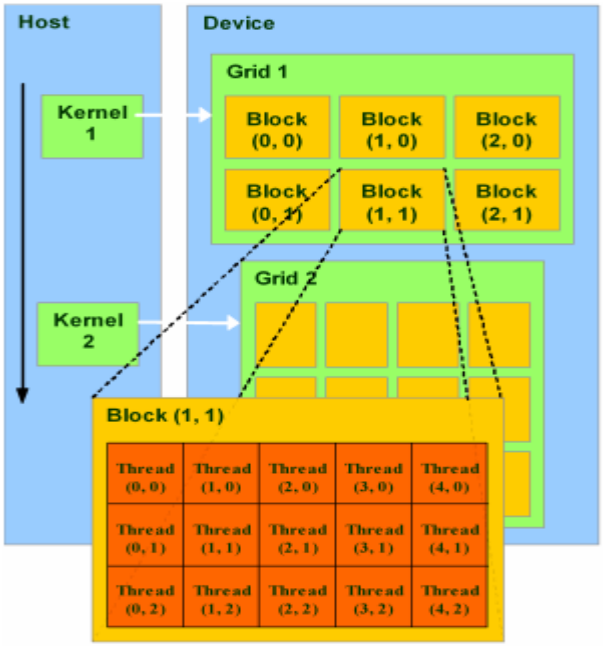

Figure 2.Example of thread organization

The way developer would like to organize his threads needs to be provided to a kernel, and it is done by passing two arguments to a kernel, one for thread block organization, other one for grid organization. Third argument that can be passed to kernel is optional while previous two are required, and it tells kernel how much of a shared memory every thread is going to use. Shared memory will be discussed in hardware implementation.

2.2 Hardware implementation

Main parts of graphics card are GPU and device memory. There are more parts that provide synchronization and scheduling among cores and memory but our focus of explanation will be on this two parts.

GPU consists of certain number of multiprocessors, which number depends of graphics card generation and price of course. What is important to understand is that every multiprocessor consists of 8 single processors who actually execute threads on them. When execution of a kernel starts every multiprocessor is assigned with a thread block to execute and every thread in that thread block will be executed on one of the .Thread blocks waits in a queue for available multiprocessor to execute on. Special scheduler is assigning thread blocks to appropriate multiprocessor and it is making sure that multiprocessors are constantly fed and work among multiprocessors is evenly distributed. Figure 3 [1] shows two different block distributions, when GPU has two or four multiprocessors.

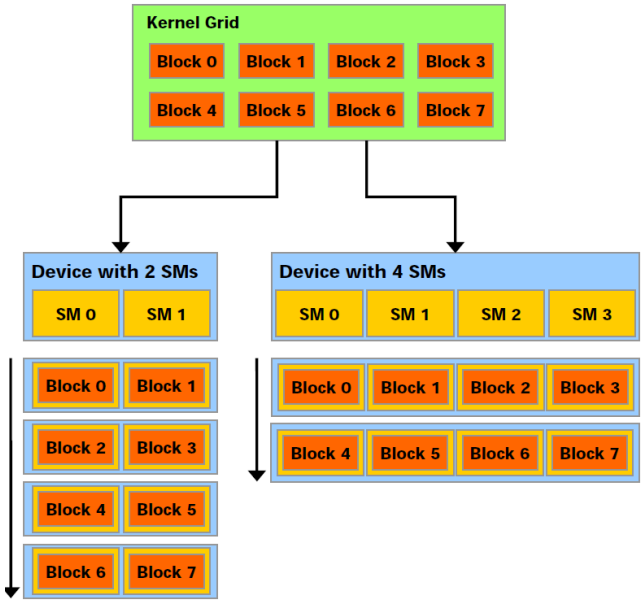

Figure 3.Example of block distribution among GPUs with different number of multiprocessors

A device with more multiprocessors will automatically execute a kernel grid in less time than a device with fewer multiprocessors.

Graphic cards have various memories on GPU's disposal to use when necessary. A thread that executes on the device has access to global memory and the on-chip memory through the memory types. Memory available is:

- One set of local registers per thread. On a chip.
- A parallel data cache or shared memory that is shared by all the threads executed on one multiprocessor and implements the shared memory space.
- A read-only constant cache that is shared by all the threads and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory
- A read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory.
- Local memory, assigned to a thread to use if needed. It is not cashed; basically it is part of the global memory allocated to a thread to serve as a helping, local memory.

Of these different memory spaces, global and texture memory are the most plentiful. There is a 16 KB per thread limit on local memory, a total of 64 KB of constant memory, and a limit of 16 KB of shared memory, and either 8,192 or 16,384 32-bit registers per multiprocessor. Global, local, and texture memory have the greatest access latency (although texture is cached), followed by constant memory, registers, and shared memory. Figure 4 [1] describes memory organization.

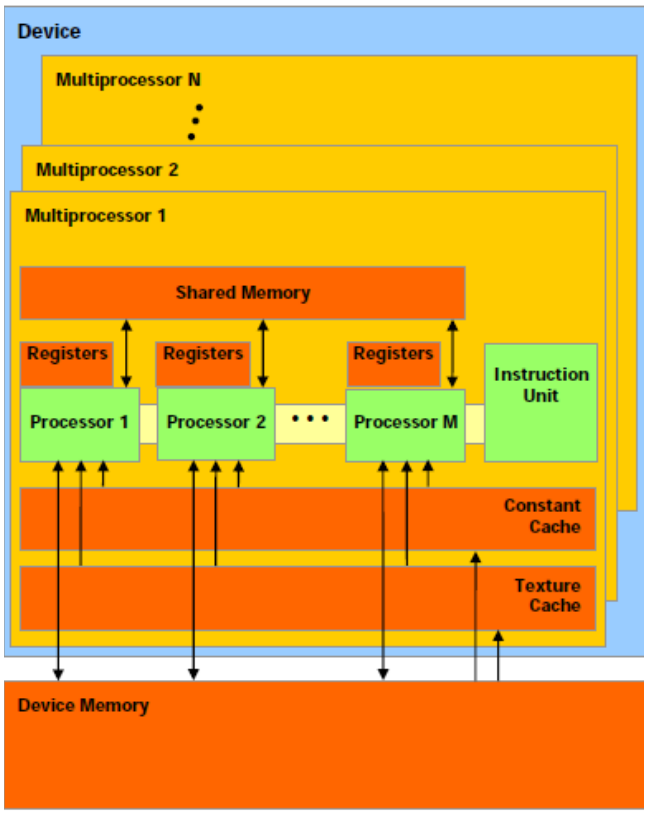

Figure 4.Memory organization on the CUDA enabled graphic cards

Since GPU processors are not best for flow control and data cashing CUDA developers came with smart solution to hide latencies caused by this. Their solution is to group threads of one thread block, that are executed on one multiprocessor, in a groups of 32 threads called warps. Basic idea behind this is to hide latencies by swapping groups of threads currently executing on cores of that multiprocessor. Rough and simplified example would be one when a group of threads want to access global memory multiprocessors initiates loading from global memory for that group of threads and in the same time swaps currently executing (waiting for data from memory) group of threads with other group of threads that can start their execution so that cores are doing some work instead of waiting for data to arrive. With this more efficient use of cores is achieved. Warp is divided into two equal parts of 16 threads called half warps. When swapping happens it happens between half warps or warps.

What is important for developers to understand is that because of this warp – half warp organization and swapping, although there are 8 cores per multiprocessor, developer has to remember that groups of 16 threads are being executed in parallel on one multiprocessor. Although this do sound confusing since there are only 8 cores that can do actual work on the physical level, developer should focus on the software level where this seems to be possible. Experiments will prove this to be important fact to know from optimization point of view.

3. MOTION ESTIMATION IMPLEMENTATION USING CUDA

CUDA allows flexibility in choosing different approaches to solve the same problem. Question is what approach to choose. Of course, right choice would be one that provides biggest boost in speed. This paper will show that that choice is not easy to be found. Every different approach usually performs significantly different and every different approach has to deal with tradeoffs because of software or hardware limitations.

Before trying to solve any problem any developer should first consider if his problem can be parallelized. Motion Estimation problem can be divided into two parts that can be parallelized. One of the parts are computing minimum sum of absolute differences (SAD) [10] between pixel blocks in reference and current frames. And second part is to find the minimum SAD values and motion vectors for pixel blocks. So two kernels were created, one calculating SAD values for all candidate blocks, and second one finding the minimum SAD values and motion vectors.

Both kernels can be implemented in different ways. Some of the ways for calculating SAD values are discussed below:

1. *Assigning every thread to calculate only one SAD value between two pixels*. Thread blocks would have same dimensions as a block size declared for the current and reference macro blocks of pixels [10]. (block Size*block Size of threads per thread block) Limitations of this approach are as follows. The first problem is that a block size has to be smaller than 22x22 in order not to exceed the 512 threads allowed per thread block. Other problem is that additional kernel for accumulation of all separate SAD calculations for each candidate block in the search range would be needed. Third issue is practically the biggest one, to calculate all SAD values for one macro block we will need (2*search Range)*(2*search Range) thread blocks per one macro block. Problem becomes apparent when search range increases or if number of macro blocks increases (by lowering block size or performing motion estimation on bigger video).

2. *Assigning every thread to calculate SAD for one candidate block in the search range*. Thread block is a (2*search Range)*(2*search Range) dimension. Advantage of this approach is that everything for a macro block is calculated in one thread block, so problem of a big number of macro blocks do not present a problem anymore. Biggest issue here is that the biggest search range possible would be 11 in order to have less than 512 threads per thread block; such a small search range is will not work for general purpose motion estimation.

3. *Assigning every thread to calculate SAD for one candidate block in a search range and use multiple thread blocks for each macro block*. This is similar to the previous method; the difference is that one thread block calculates one quarter of a search area, so four thread blocks are needed per macro block. In this case maximal search range possible would be 22, which is better but still not flexible enough for motion estimation.

4. *A mixed approach*; this approach removes limitations regarding possible search range size by dynamically assigning more thread blocks to cover whole search area. After experimenting we determined that solutions 1, 2, and 3 above were ruled out and the mixed approach is selected as a starting point for our experiments.

Idea of this paper is not to focus in detail how motion estimation algorithm can be implemented as presented in [3] and [11], rather to explore CUDA architecture on familiar problem. Because of that all

the above mentioned approaches are described briefly mainly focusing on differences between approaches from CUDA perspective.

Figure 5 describes how search area is organized for every current MB. Corresponding pairs of pixels are used to calculate SAD values for one reference MB in search area. Corresponding pairs of pixels are paired by same position in reference and current MB. Number of pairs per MB matches number of pixels in MB. All SAD values from every pair for one MB are accumulated and that accumulated SAD value represents SAD value for that reference MB in search area at the end there are search area number of SAD values for every current MB in a frame.

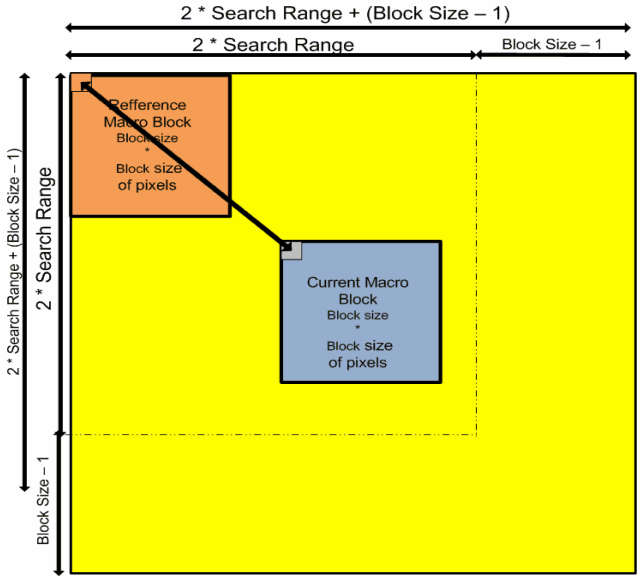

Figure 5.Search area for one current block that has to be covered using full search in motion estimation

4. EXPERIMENTS

For the purpose of this paper, experimentations done on one of the kernels is enough to show impacts of different approaches developing using CUDA. So our focus will be on a kernel for calculation SAD values for macro blocks, specifically approaches 3 and 4 in previous section as a starting point.

Idea is to modify SAD kernel to see how much different approaches would impact overall performance. Most significant improvements in performance can be achieved by focusing on the following:

1. Removing un-coalesced loads and stores from global memory
2. Using shared memory instead of global memory.
3. Removing shared memory bank conflicts to avoid serialization of thread execution.
4. Thread block size to be multiple of 16(e.g. 16xn or 32xn etc.).

4.1 Experiment I

This experiment is devised to show quantitatively how different implementations of the same code can impact performance. Five different codes will be run and compared, every one of them building upon the previous one. In that way it can easily be seen how specific optimizations improve execution speed.

1. First implementation is a baseline code written in C and completely run on the CPU. It is a baseline code in a sense that both speedup and accuracy of GPU implementation are checked against the baseline results. For a GPU implementation to be valid it has to provide the same output for the same set of inputs as a baseline code.

2. Second implementation is representation of approach 3 mentioned at the beginning of this section. This is not fully optimized code, meaning all of the four recommendations for improving performance are ignored. This implementation is direct port of the CPU code, with a biggest difference being additional calculation for fetching and storing appropriate data elements from global memory.

3. Third implementation differs from the second one only in one part. Additional temporary variable is being created in every thread to serve as an accumulator for intermediate SAD calculations. That way global memory stores are reduced somewhat per thread. Storing happens only once, at the end of the thread.

4. Fourth implementation introduces using of shared memory. Since shared memory is available to every thread of a thread block, idea is to use all the threads of a thread block to preload portion of a reference and current frame which is needed for calculating SAD values. Here global memory accesses are reduced to minimum. (every thread loads in couple of pixels which other threads can use).

5. Fifth implementation represents completely optimized code. Every recommendation is taken into consideration, such as thread block size is multiple of 16,un coalesced loads and stores are removed, there is no warp serialization because of shared memory bank conflicts etc.

All implementation are executed on NVIDIA 8800 GT which has compute capability of 1.1 and 14 multiprocessors. All implementation are run for 3 video files with different resolutions. Since the algorithm tested is exhaustive full search motion estimation, there are no content dependencies.

4.2 Experiment II

This experiment will try to show scalability of performance using different graphics cards with CUDA capabilities. Idea is to show how code performs when more or less multi cores are available for executing threads or when GPUs with different compute capabilities are available. Also performance of codes for different search ranges can be interesting and important data.

Implementations used in this experiment are:

1. Same implementation used in previous experiment, mentioned as a number 1.

2. C code executing only on CPU with a difference between it and previously mentioned code is that it also calculates sub pixels using H.264 interpolation and performing full search on them also.

3. Same implementation as mentioned in experiment I under number 5.

4. implementation that performs motion estimation both on integer and sub pixels using code mentioned under 3 as a base.
5. Same implementation as mentioned in experiment I under number 3.

All implementation are run on different machines, with different graphics cards. Cards that are used are:

- 8800 GT - compute capability of 1.1 and has 14 multiprocessors
- 9800 GTX – compute capability of 1.1 and has 16 multiprocessors
- 285 GTX – compute capability of 1.3 and has 30 multiprocessors

All codes will be run for 3 video files with different resolutions.

5. RESULTS

5.1 Experiment I

Provided below there are 3 graphs, each representing times for different CUDA implementations performed on 3 different video files.

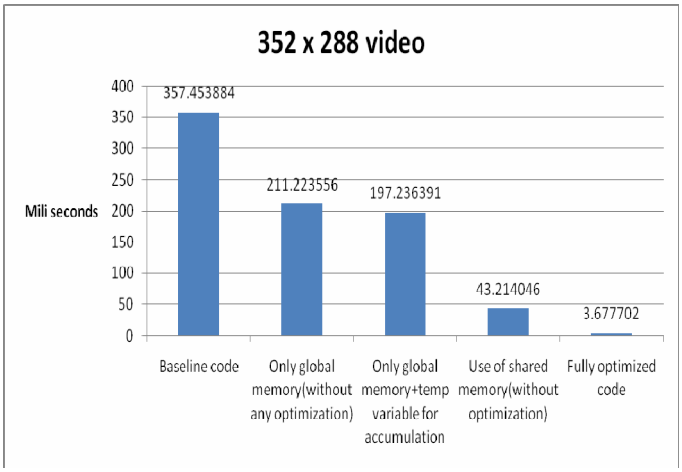

Figure 6.Results for the video file of 352x288 resolution

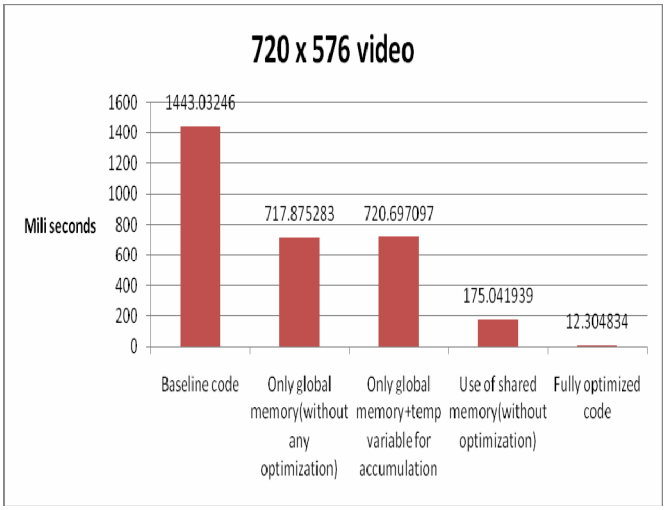

Figure 7.Results for the video file of 720x576 resolution

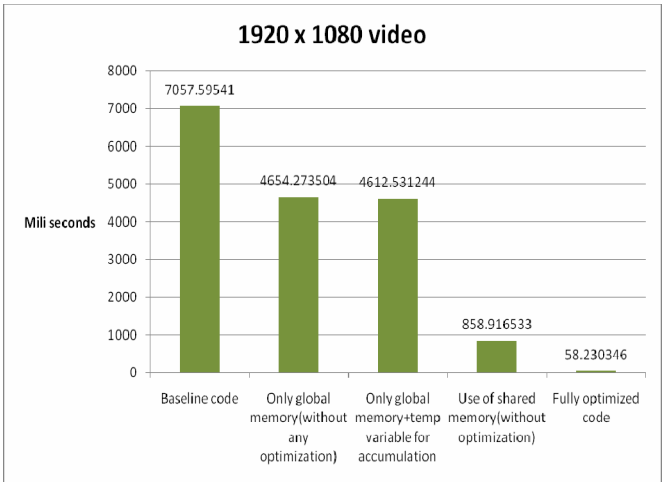

Figure 8.Results for the video file of 1920x1080 resolution

Comparing all three figures it is obvious that pattern emerges. Baseline code is slowest, followed by the next two implementations that differ a little in times in a favor of implementation with temp variable added. Speedup with these two implementations is less than 2, which is significant only for large videos with big resolutions. When shared memory is introduced we can see dramatic improvement. That is showing us how important using of shared memory even in its simplest non optimized mode is. Finally fully optimized code showed another dramatic boost in performance. When all the pieces of the puzzle are put to the right place speedup can be drastic.

Also one more important conclusion that can be taken looking from the graphs are that program running time increases for videos with a bigger resolution which was quite expected result. Bigger resolution means bigger number of current MBs which means more work.

5.2 Experiment II

Since we know how dramatic speedup can be from the previous experiment, experiment two is devised to built upon that knowledge and provide us with information regarding how code performs when search range varies and when more computing power is on codes disposal.Search range varied from 5 to 46 to cover mostly used search ranges in today's motion estimation algorithms.

5.2.1 8800 GT

8800 GT can be regarded as a middle class graphics card, although it has been outdated since newer generations have showed up. But since it is reliable and relatively cheap compared to its capabilities it can serve as a starting point for our experiment.

Provided bellow there are 3 graphs, each representing times for different CUDA or CPU implementations performed on 3 different video files.

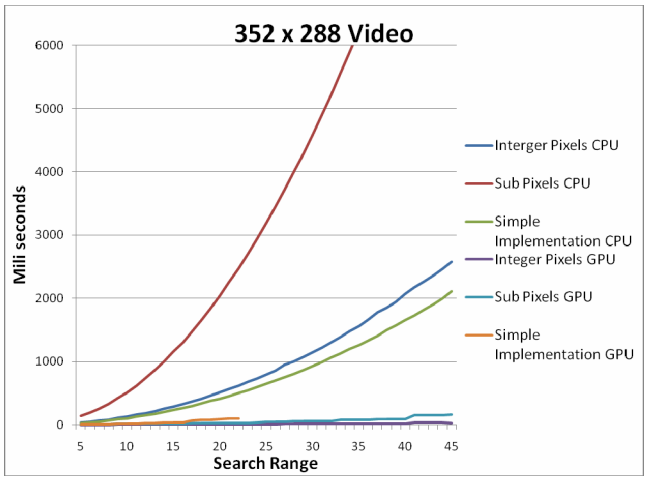

Figure 9.Results for the video file of 352x288 resolution

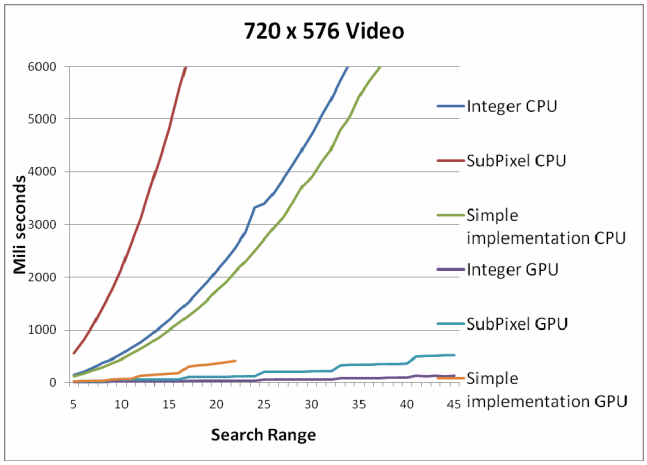

Figure 10.Results for the video file of 720x576 resolution

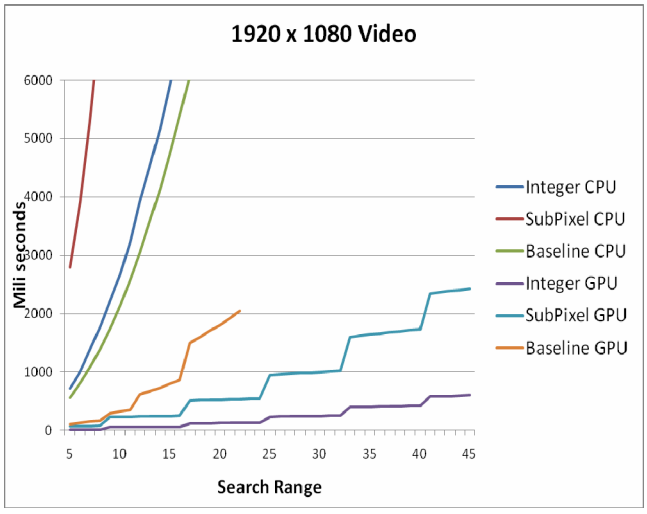

Figure 11.Results for the video file of 1920x1080 resolution

Again, comparing 3 tables we can derive a pattern. While time for CPU implementations increase exponentially, time on the GPU increases semi-linearly with search range increasing. Meaning that bigger speedup is going to be achieved for bigger search ranges.

Also worth noticing is that for videos with a bigger resolution CPU time is increasing exponentially faster than for the smaller resolution videos while although GPU times increase with resolution they increase linearly. That mutual relation leads to greater speedups for larger resolution videos which use bigger search ranges.

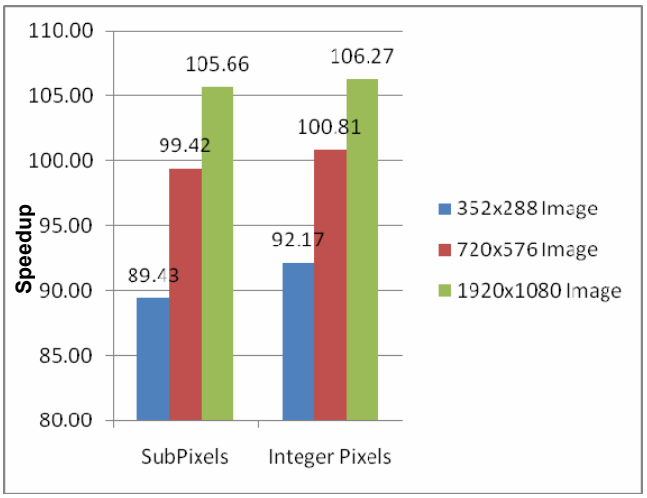

Figure 12.Comparison speedup with increased resolution for 8800 GT graphics card for search range of 16

5.2.2 9800 GTX

9800 GTX differs from 8800 GT just in the number of multiprocessors, it has two more. So results should show increase in performance purely based on a more number of computing units available since they do not differ in compute capabilities, they both have compute capability of 1.1.

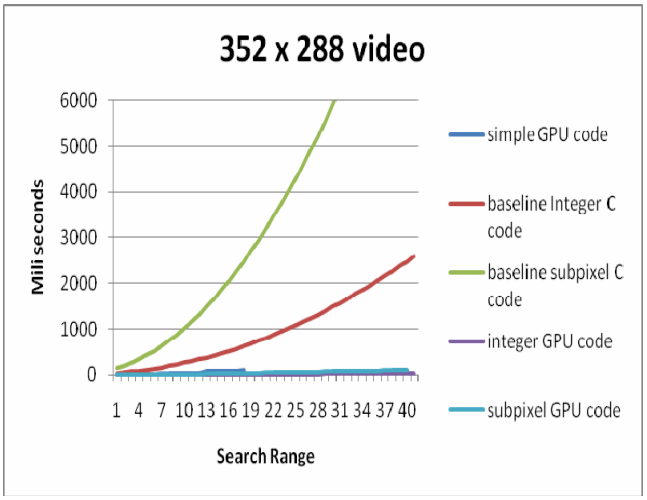

Figure 13.Results for the video file of 352x288 resolution

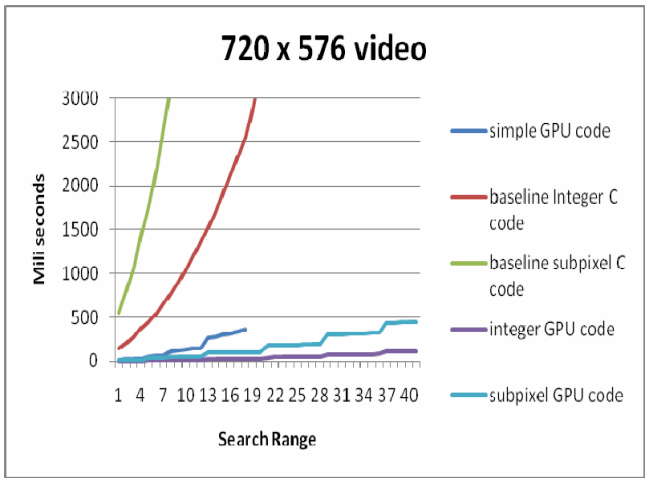

Figure 14.Results for the video file of 720x576 resolution

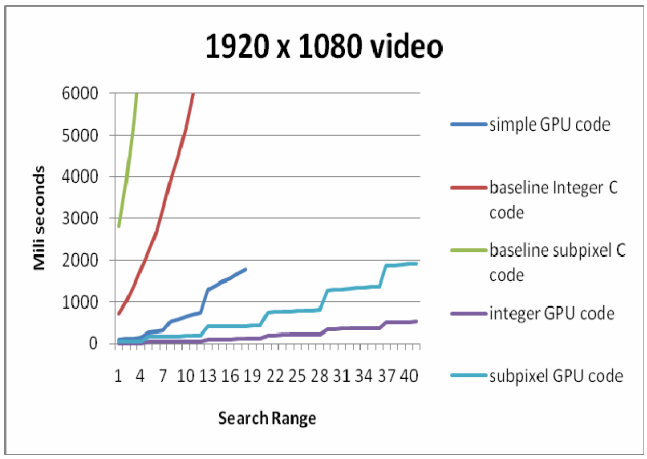

Figure 15.Results for the video file of 1920x1080 resolution

On the first look, compering these three figures with three equivalent figures from section 5.2.1 it is noticable that the same trends are being followed.Baseline C codes rise exponentially and GPU code rise semi-linearly.

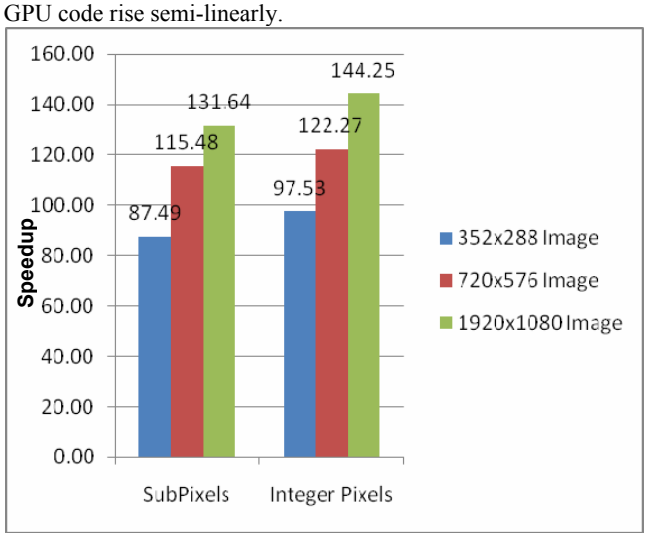

Figure 16.Comparison speedup with increased resolution for 9800 GTX graphics card for search range of 16

Figure 16 shows significant speedup for all three resolutions regardless if motion estimation is done on only integer pixels or on sub pixels also.

5.2.3 285 GTX

This card is one of the most powerful card available today, with highest CUDA capabilities of 1.3 and almost double multiprocessors compared to 9800 GTX. Following figures represent difference in speed when high-end card is used.

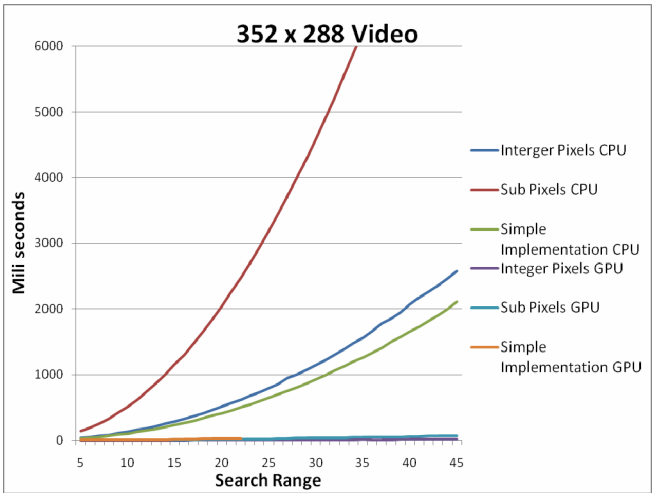

Figure 17.Results for the video file of 352x288 resolution

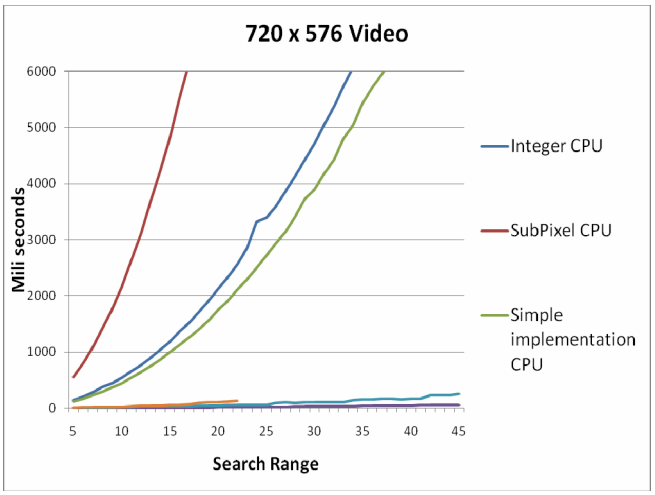

Figure 18.Results for the video file of 720x576 resolution

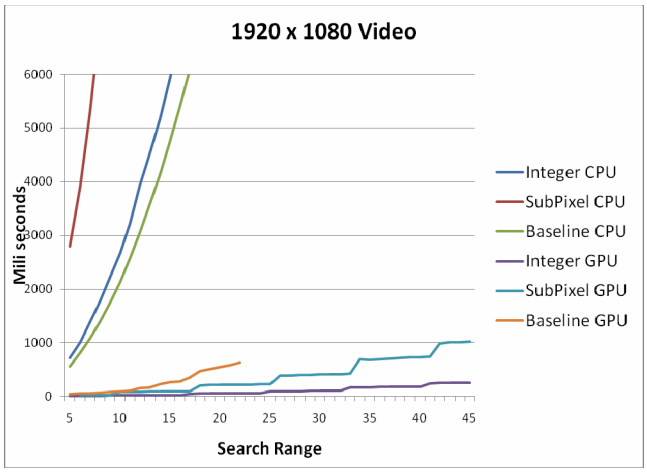

Figure 19.Results for the video file of 1920x1080 resolution

It is hard to comprehend changes since all the figures look, or better said behave the same. They are there mostly to show the trend, how GPU code is behaving versus CPU code. For better feel of actual speed increase we provide Figure 20.

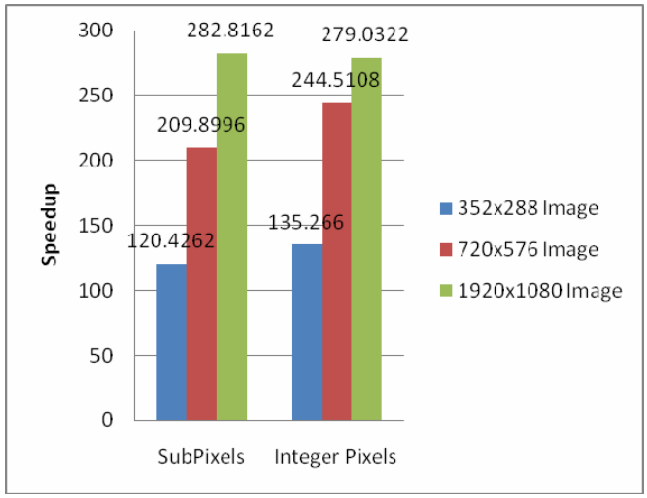

Figure 20.Comparison speedup with increased resolution for 285 GTX graphics card for search range of 16

5.2.4 Comparison

Since 9800 GTX graphics card has two more multiprocessors available for computing we were roughly expecting speedup increase of 20 percent compared to 8800 GT. More than double number of multiprocessors against 8800 GT, and almost double against 9800 GTX were logically leading us to conclusion that 285 GTX will approximately half the time of the execution compared to other two cards. Figure 21 confirms our expectation.

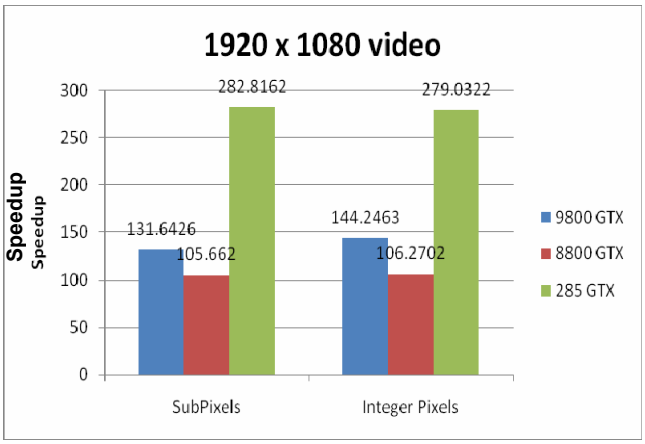

Figure 21.Comparison speedup with 1920x1080 resolution for 9800 GTX and 8800 GT graphics cards for search range of 16

5. CONCLUSION

Although using CUDA requires balancing between desires and hardware and software limitations results shown in this paper suggest that by following a set of optimization guidelines and carefully breaking problem down into parallelizable portions gains from using CUDA can be drastic. Simply measured in terms of a speedup, cutting execution time by hundred times should be enough motivation for every engineer to start considering using CUDA. Important lesson is the significance of the domain knowledge in developing parallelization strategies.

Experiment I showed the importance of using appropriate optimization techniques. Two suggestions for most significant improvements on performance are: reducing global memory accesses by using shared memory and removing un coalesced loads and stores to global memory. As shown in the figures 6-8 most significant speedup was seen when those two optimization techniques were implemented. Since that is the case, main focus of every CUDA developer would be to tackle those two issues first. And later to focus on improving speed more by optimizing further using other techniques. Contribution of the optimization techniques can be quantified as:

1. Removing un coalesced loads 40%
2. Using shared memory 40%
3. Removing shared memory bank conflicts to avoid warp serialization 10%
4. Reducing the number of registers and instructions used per thread 5%
5. Avoiding branching(using of if, else etc. statements) 5%

Experiment II focused on showing performance difference on different graphics cards as well as performance compared to different search ranges.

When different search ranges are taken into consideration it is shown that while time for CPU implementations increase exponentially, time on the GPU increases semi-linearly. Meaning that bigger speedup is going to be achieved for bigger search ranges. And since CPU times are raising faster bigger resolution videos while GPU times rise with resolution linearly conclusion is that maximum gain out of these CUDA implementations would have problems dealing with big resolutions (e.g. HD video) performing big searches in motion estimation.

One of the obvious conclusions is that CUDA architectures scales well and linear performance improvements can be achieved when run on GPUs with more multi processors. How significant that increase we can expect is nicely shown in figure 21.

Biggest issues that we encountered were need for constant balancing between performance and tradeoffs. Although following a set of general guidelines seems simple, elegant solution is not easily achievable; it takes time, practice and experience. Seems like learning and following a set of general guidelines and introducing some extra effort in approaching problem from CUDA (data parallel) point of view is really small price to pay comparing to the gains in speedup.

6. REFERENCES

[1] NVIDIA CUDA Programming Guide version 2.3 (7/1/2009) developer.download.nvidia.com/.../cuda/.../NVIDIA_CUDA_Programming_Guide_2.3.pdf

[2] S. *Yang*, et al., "Power and Performance Analysis of Motion Estimation Based on Hardware and Software Realizations," in *IEEE Trans. Computer,* vol.54, pp.714-726, Jun, 2005.

[3] C.-W. Ho, et al., "Motion Estimation for H.264/AVC Using Programmable Graphics Hardware," in *Proc. IEEE Int'l* Conf. on Multimedia and Expo, July 2006, pp. 2049-2052.

[4] C.-Y. Lee, et al., "Multi-Pass and Frame Parallel Algorithm of Motion Estimation in H.264/AVC for Generic GPU," in *Proc. IEEE Int'l Conf. on Multimedia and Expo*, July 2007, pp. 1603-1606.

[5] Y.-C. Lin, et al., "Multi-Pass algorithm of Motion Estimation in Video Encoding for Generic GPU," in *Proc. IEEE International Symposium on Circuit and Systems*, May 2006, pp. 4451-4454.

[6] R.-X. Chen and J. Fan, "Complexity reduction for SOPCbased H.264/AVC coder via sum of absolute difference", *IEEE/CIE 7th Int'l Conf' on ASIC,* pp. 1277-1280, Oct. 2007

[7] S. Ryoo, et al., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," *Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming,* Feb. 2008, pp.73-82.

[8] GPGPU. http://www.gpgpu.org/

[9] K. Mueller, F. Xu, and N. Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?" in SPIE Electronic Imaging Conference, San Diego, 2007, (Keynote, Computational Imaging V).

[10] E. G. Richardson, Iain (2003). *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. Chichester: John Wiley & Sons Ltd..

[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