INVESTIGATING THE PERFORMANCE OF MOTION ESTIMATION BLOCK-MATCHING ALGORITHMS ON GPU CARDS

Eralda Nishani
Department of Computer Engineering
Polytechnic University of Tirana
Albania
enishani@gmail.com

Betim Çiço
CST Faculty
SEEU, Tetovo
FYR of Macedonia
b.cico@seeu.mk.edu

Neki Frashëri, Prof. Dr.
Department of Computer Engineering
Polytechnic University of Tirana
Albania
neki.frasheri@gmail.com

ABSTRACT
In the field of video compression, motion estimation (ME) is a process that leads to high computational complexity. Implementation of ME block-matching (BM) algorithms on general purpose Central Processing Unit (CPU), has resulted in poor performance. In this paper we investigate the performance of two BM ME algorithms: Three Step Search (TSS) and Four Step Search (4SS) on Graphics Processing Unit (GPU) NVIDIA Quadro 400 using the Compute Unified Device Architecture (CUDA) platform. Both algorithms perform motion estimation on a block-by-block basis, which is considered the simplest way in terms of hardware and software implementation. The focus is to achieve parallelization of the algorithms for a real time execution. We consider two well-known test sequences: "football" and "mad900", with different image resolution. The results show that the implementation on a GPU card can improve the performance in terms of execution time, by a factor of 1000.

Categories and Subject Descriptors
I.3.1 [**Computer Graphics**]: Hardware Architecture – *graphics processors, parallel processing*

I.4.1 [**Image Processing and Computer Vision**]: Compression (Coding) – *approximate methods*

General Terms
Algorithms, Experimentation, Performance.

Keywords
Motion Estimation, Block-Matching Algorithm, Three Step Search, Four Step Search, Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA).

1. INTRODUCTION
With the development of network and communication, multimedia service is becoming more and more popular. Video communication is more and more requested, like sending and receiving real-time video during video conferencing or mobile communication. One main problem during video transmission is bandwidth demand. Sending several frames per second in order to create the illusion of a continuous moving sequence with high resolution, requires high bandwidth. As a result, video compression was considered a solution to such a problem. There are different compression standards from MPEG-1 to MPEG-4 and H.264/AVC [1], which focus on digital video compression. The goal is to achieve compression, while providing acceptable video quality.

One important process in video compression is Motion Estimation (ME) – evaluating the motion between different frames. Specifically, it estimates the motion parameters of moving objects in an image sequence [2]. This process is quite complex and the most computational intensive (more than 50% of the entire compression process volume [3]). As a result, serial implementation on general purpose CPUs (Central Processing Unit) have not resulted effective. Attempts to implement motion estimation algorithm in VLSI (Very Large Scale Integrated) devices can be seen at [4]. The results show that the algorithm requires too many cycles to complete, the engine becomes complex and there are memory-access conflicts. In fact, to manage this kind of processes, the right solution is parallel implementation. Since VLSI implementations do not achieve the required performance, scientists have taken in consideration using GPUs (Graphics Processing Unit) and CUDA (Compute Unified Device Architecture) platform.

In this article, we will study two particular motion estimation algorithms: TSS (Three Step Search) and 4SS (Four Step Search), that belong to the class of block-matching algorithms – the most effective algorithms in ME.
Displacement measurement and interframe coding based on block-matching was introduced in 1981 [5]. Since the most simple algorithm Full Search (FS), which never found practical implementation, there has been further improvement to this method. The main focus is on the parallelization of block-matching algorithms and improving their execution time.

2. MOTION ESTIMATION AND BLOCK-MATCHING ALGORITHMS
As we have mentioned earlier, motion estimation is the process of

calculating motion between consecutive frames in a video sequence. In order to understand ME, we have to study the concept of a motion picture or a video sequence – how a video is organized. Motion picture can be described as a sequence of several frames. While frames are still pictures, that represent an instant of the video. Once encoded, a video is several consecutive frames shown at a particular high frequency. This frequency is high enough to give the illusion of a continuous animation. In practice, frame rate values vary between 24 fps (frames per second) to 300 fps. Each frame is shown for a small fraction of a second - for a frame rate of *k* fps, it is shown for *1/k* seconds. Since frames are shown so close to each other, they are expected to be quite similar. Precisely, this is what ME exploits. The fact that there is temporal correlation between frames, makes the prediction possible and quite exact.

There are several ME techniques like: block matching, differential [6,7] and Fourier transform [8]. Since frames have a rectangular shape, dividing it into blocks is easy. As a result, block matching method is the most popular and that is the topic addressed in this article.

A video frame is composed of a number of pixels, which are grouped in 8x8 blocks. According to block matching, a frame is organized into a matrix, that contains macro blocks, composed of the aforementioned blocks. The size of macro blocks is a multiple of 8x8. Simulations and practical implementations have shown that the most suitable size is 16x16. The process of ME is that a macro block in the current frame is compared to another macro block in the reference frame. If a similar block is found, the motion vector is transmitted instead of the whole block. The motion vector represents the result of motion estimation and is the most important result of the process. Since the goal is to reduce the amount of calculations, the search area is limited to a certain number of *p* pixels around the macro block. This is called the search parameter. A high *p* value means that a higher number of calculations are needed to estimate motion. For the 16x16 macro block, the suitable value for the search parameter is *p=7*. This process is demonstrated in figure 1.

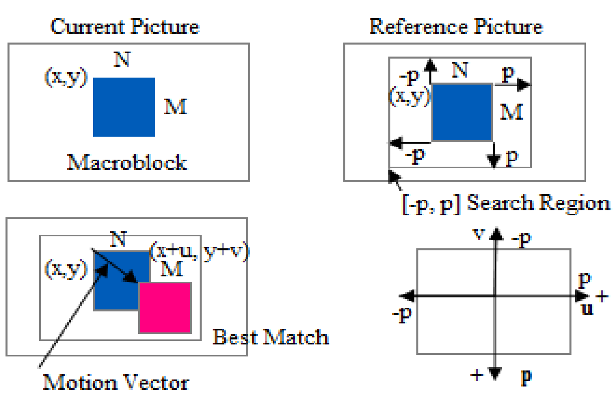


Figure 1. The process of Motion Estimation

The matching of macro blocks is based on the cost parameter. The macro block with the lowest cost, is considered the right one. There are several functions to calculate the cost, among which we have chosen the following:

MSE (Mean Squared Error) – represents the expected value of squared error loss,

$$MSE = \frac{1}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}(C_{ij} - R_{ij})^2$$

where N is the side of the macro block, C and R are the pixels that are being compared in the respective macro blocks.

The problem is how to search for the most suitable macro block. The method defines the block matching algorithm. Researchers have made several attempts to find the most effective algorithm. The most simple is FS (Full Search), that compares each macro block of the current frame with the candidates in the reference frame. The required computations are huge due to the large number of candidates to evaluate. As a result, it remains an ideal algorithm, mostly theoretical and not implemented in practice. Among the variety of block-matching algorithms that exist, we will study:

1. TSS (Three Step Search) – The first attempt to build a fast algorithm, that could be implemented in real life.

2. 4SS (Four Step Search) – An improvement to TSS, resulting in a more stable and hardware-oriented algorithm.

We will shortly describe both algorithms, in order to have a general ide on how they work and how is motion estimation calculated.

2.1 Three Step Search Algorithm

The algorithm was first proposed by Koga et. al [9]. It is based on the method of block-matching as mentioned earlier. In order to implement the algorithm, the following steps are followed, as is shown in the graph (figure 2):

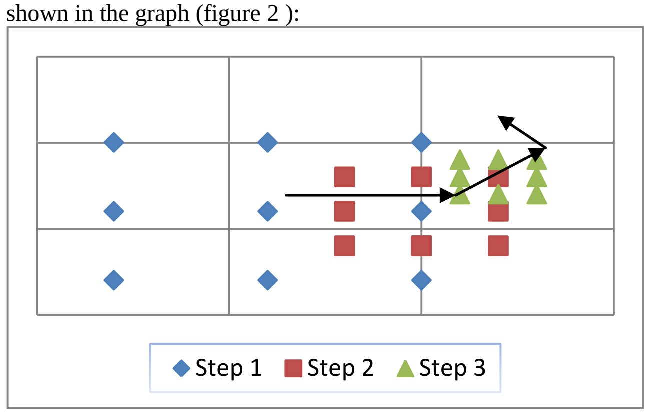


Figure 2. Example of steps followed for the TSS algorithm

1. Nine points are searched in an area with the resolution of 4-pixels/4-rows or 9x9 points. The search origin is the center point, with the (0,0) coordinates. The point with the minimal cost is considered the search origin for the next step.

2. The size of the search window is changed to 5x5.
The lookup still occurs through nine points. The point with the minimal cost is considered the search origin for the next and the last step.

3. The search window is reduced to the size of 3x3. The point with the minimal cost in this step, defines the motion vector.

This procedure is repeated for every macro block in the frame, for each frame. The result motion vectors represent motion estimation. For a search parameter *p* with the value of 7, the maximum number of search points is 9+8+8=25.

2.2 Four Step Search Algorithm

This algorithm was first introduced from Po et. al. [10] in 1996. It came as a further improvement to TSS algorithm. As the name suggests, the algorithm includes the following four steps:

1. Nine points are searched in a 5x5 search window, located in a bigger search area of 15x15 size. If the point with the minimal cost is found at the corner of the window, then the flow falls immediately to the last step; otherwise it goes to the next step.

2. The search window is still maintained at 5x5. The search model depends on the previous point location:

a) If the previous point is located in the corner, then five more points are searched, according to the model in the graph (figure).

b) If the previous point is located in the middle of the horizontal or vertical axis of the search window, then three more points are searched; according to the model in the graph (figure).

c) If the point is at the center of the window, then the flow falls immediately to the last step; otherwise it goes to the next step.

3.This step is the same as the second step, but in the end it is followed directly by the fourth step.

4.The search window is reduced to the size of 3x3. The point with the minimal cost found here, defines the direction of the motion vector.

The procedure (example in figure 3) is repeated for each block in the frame; the same as TSS algorithm. It is easy to tell, that if every minimal cost point is located in the middle of the search window, then the intermediate steps can be eliminated.

2.3 TSS versus 4SS

Compared to the first FS algorithm, TSS was faster and reduced the number of calculations [10]. Also, it was efficient and easy to implement. Nevertheless, the algorithm still had problems in evaluating small movements. This was extremely important, because experimental results [11], have shown that for real world moving sequences, the movement area changes slowly.

On the other hand, 4SS had no problems in this area. Its performance did not change in the case of complex movements, like closeness of the camera or quick movements. This algorithm was more stable and easier to implement, since it provided hardware-oriented features.

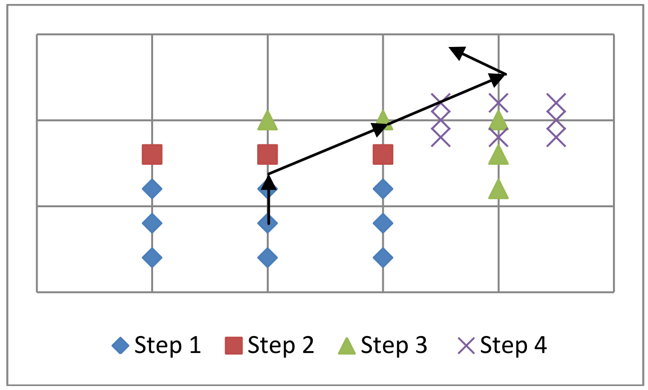


Figure 3. Example of steps followed for the 4SS algorithm

3. CUDA CAPABLE GPUs

GPUs represent a special kind of processor, mostly built to deal with graphical problems. In the beginning, they were suitable for a class of applications with the following characteristics [12]:

- Requirements for high amount of calculations
- Main focus on parallelization
- Throughput is more important than latency

Lately, GPU is transforming into a powerful programmable processor, used in other fields as well. Even though initially it was considered for academic and scientific purposes, with the development of last generation GPUs, real applications can be built. It is especially requested for applications, that include large complex calculations. In this context GPU is considered a better solution than CPU in the field of motion estimation.

The CUDA platform is a computing engine [13], developed to facilitate GPU usage. GPU implementations before CUDA are hard to understand, complex and difficult to maintain. In the case of CUDA capable GPUs, a GPU is called a GPGPU (General Purpose GPU). A GPGPU is a logical concept, according to which a GPU can be used to solve non graphical applications. A GPGPU is a special kind of GPU, which resembles more to a CPU. This is observed in the memory model. The programmer can load and save data in the main memory. A GPGPU has its own RAM memory, which is also known as global memory. In the figure, is shown the CPU-GPU memory structure. In this kind of programming model, a GPU does not act alone; there is always interaction between CPU and GPU. As it can be seen, GPU cannot handle all the necessary actions to solve a problem alone. For example, the CPU intervention is needed to provide the initial data and to save the results. We will implement our algorithms in a NVIDIA QUADRO 400 GPU, on which we will base the following GPU description.

3.1 GPU hardware and programming model

We can better understand the GPU architecture, by comparison to CPU, which is also shown in the figure 4 and figure 5.

CPU architecture organization:

- Bigger size cache memories
- Limited number of ALUs (Arithmetic Logic Unit)
- Main focus on latency reduction
- SISD (Single Instruction Single Data) architect

GPU architecture organization:

- Small size cache memories
- High number of ALUs (Arithmetic Logic Unit)
- Main focus on throughput increase
- SIMD (Single Instruction Multiple Data) architecture

The SIMD architecture means that a single instruction is executed on different multiple data. Furthermore, multiple SIMD units, result in the MIMD (Multiple Instruction Multiple Data) architecture, where each unit controls a set of functional ALUs. Data streams that require different actions, are assigned to different controllers. Every element of the stream is processed in one of the functional units. This is how operations on huge data sets are easier on GPUs.

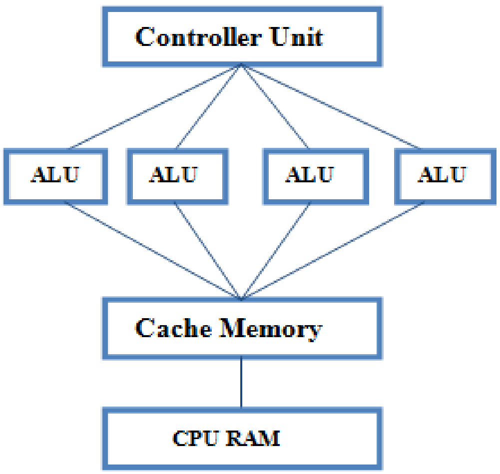


Figure 4. Architectural organization of a typical CPU

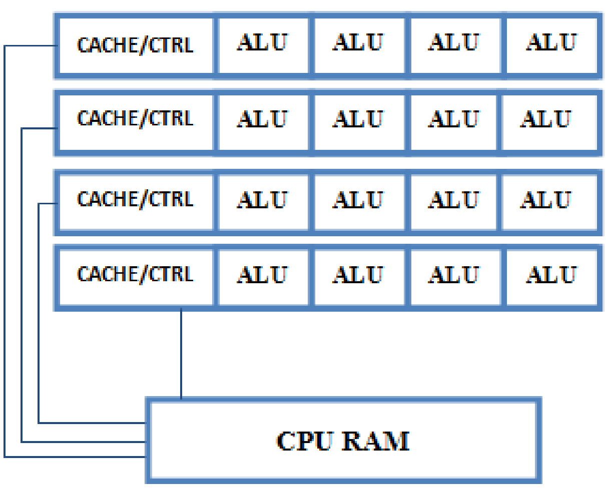
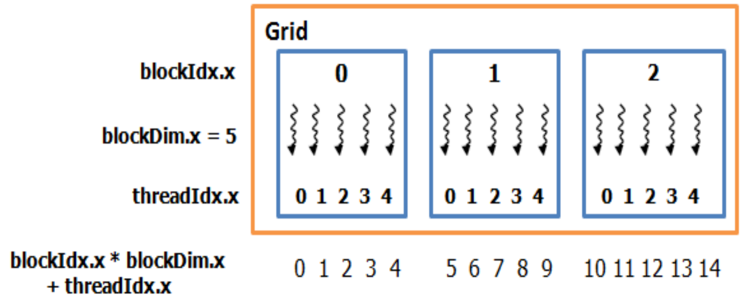


Figure 5. Architectural organization of a typical GPU

Paralell programming in GPU is translated in programming with threads. Threads are organized in a hierarchy in GPUs. If we look at figure 6, we can clearly see the threads, the blocks and the grids. Executing a program in GPU, which is known as a kernel, creates a grid with blocks of threads. Threads within a block can: (a) share data using a shared memory and (b) can synchronize

Figure 6. Identification of threads through block and grid IDs

their execution. Threads in different blocks cannot cooperate, while threads in different blocks are expected to be located in the same processor core. For this reason, the number of threads inside a block is limited by the memory resources of the processor. A kernel can be executed from several blocks of threads and



furthermore the blocks are organized in a grid.

3.2 GPU memory model

We will shortly describe the memory hirerarchy as well, which is closely related to the threads. As we mentioned, during the execution, each thread has a local private memory. Each block of threads has a shared memory, which is visible to all the threads in the block. In general, all threads access the same global memory.

There are two more extra read-only memory spaces, accessible by all threads: constant and texture memory. Memory types in a GPU:

- Register memory – is implemented on a GPU and it can be accessed faster.
- Local memory – is located outside the chip of GPU, the access speed is 100 times slower than the register.
- Shared memory – is used to store the parameters of the kernel function. The access time is at the same levels as register memory.
- Global memory - is located outside the chip of GPU, the access speed is 100 times slower than the register. It has bigger capacity and is accessed by all threads.
- Constant memory – is accessible by all threads and is cached on chip, so the data hit is fast.
- Texture memory – is cached on chip and is built to serve applications that require a certain access pattern.

3.2.1 Texture memory

We will focus on the texture memory, because this is the memory kind, that we will use in our implementation. The texture memory was first introduced to represent more realistic objects, enabling image 'drawing' in a geometric space [14]. It is suitable for those applications, where memory access manifest spatial locality. This means that it is more possible for a thread to read from an address, which is near the address read by the neighbor thread. In this cases, texture memory brings performance improvement. Since we are studying motion estimation algorithms, that deal with images, this memory is suitable.

During our implementation we will consider a different number of frames (images), to study a video sequence. Layered texture memory, also known as *texture array*, which are special constructs that enable textures to be organized as an array, with access to an index. The biggest advantage of texture arrays is that they support larger extensions than that of a unit. We will utilize 2D layered textures to store the sequence of images.

4. ALGORITHMS IMPLEMENTATION

In order to implement an algorithm in GPU, we need to take into account that we are programming in a hybrid environment. A program in CUDA has two important parts:

- The host program, which is executed on CPU and is sequential
- The kernel, that part of the program executed on GPU, parallel and run by threads.

To program in CUDA, we will use a special kind of C programming language, CUDA C. It is C with several language extensions to allow heterogynous programs and to provide parallelization. To be clear, the contribution from this article is specific to the parallelization of the algorithm, not to the modification of the algorithm itself. There have been earlier articles related to this field. In most of the cases like in [16], implementations are performed on Matlab. In other articles [17], the most simple algorithm (FS) is implemented on CUDA. In fact, we have focused on the advantages that CUDA can bring to the deployment of algorithms, that can be used in real life. We have the program in C for both of our algorithms that belongs to a group of researchers that you can find in this reference [18].

4.1 Algorithms parallelization

We are studying TSS and 4SS algorithms. The methods to parallelize the algorithms are the same for both of them. That is why we will describe the process only once. Some of the methods that we use to achieve parallelism are:

1. Thread linearization – We convert the index from the 2D space in a linear one by using these two instructions:

   ```
   int x = blockIdx.x*blockDim.x +
             threadIdx.x;

   int y = blockIdx.y*blockDim.y +
             threadIdx.y;
   ```

   As we mentioned earlier on section 3.1, every thread is accessible in the thread hierarchy. According to the instructions, each parallel thread will start at an different data index. In this way we manage to process data in parallel and to accelerate the calculations.

2. 2D texture memory – more precisely, texture arrays, in which are stored the sequence of frames. Fast memory access located on the GPU chip. Studies have shown that for two consecutive frames with very little time difference, the movement pattern shows spatial locality. This means that if a pixel (block) is changing location from one frame to the other, than the neighbor pixels will follow the same model. This complies with the purpose of the texture memory. Since frames are images, we will use 2D texture memory.

3. Variables in the `__device__` functions, are stored as `volatile`. Very often, instead of keeping the variable in a register (when it is needed in several places), the CUDA compiler inlines the operations needed to compute its value. This brings instructions duplication. The 'volatile' comes as a solution to this problem; forcing the variable to be kept and used. For further clarification, `__device__` functions are executed only on GPU and can be called only by the kernel. This means that this type of variables can only be accessed by threads.

4. Access to global memory is one of the problems in GPU programming. It is quite slow compared to the memory already on chip. In our case, global memory is accessed only once, to store the final result.

The final step is during the execution of the kernel. The syntax for calling the kernel in GPU is very particular. It specifies the grid of threads that is needed to execute the kernel. Namely, below we will demonstrate the case for the 4SS algorithm.

```
dim3 threadBlock(512,512,1);

dim3 blockGrid(width/threadBlock.x,
height/threadBlock.y, 1);
```

In these two instructions, two variables of `dim3` type are declared. This is a standart type for CUDA. They represent the number of threads per block and the number of blocks per grid, which depends on the width and height of the frame.

```
FSS_GPU<<<blockGrid,threadBlock,1>>>(i, j,
mv_GPU[i], width, height);
```

In order to call the kernel for execution, we need to provide the two aforementioned variables. This part of the syntax `<<<...>>>` determines the execution of a function as a kernel destined for execution on GPU.

5. IMPLEMENTATION RESULTS

Implementation is performed under the following conditions:

1) Host computer

- Operating System – Red Hat 4.4.6-3
- CPU – AMD Athlon II X3 455 – 800 MHz
- RAM – 3915104 kB

2) NVIDIA QUADRO 400

- 6 MP x 8 cores/MP = 48 cores
- Global memory – 511 MB
- Shared memory – 16384 B
- Nr. of threads per warp – 32
- Max. nr. of threads per block – 512

3) CUDA platform

- Driver Version – CUDA 4.2

Both algorithms are implemented using two test sequences from [19], precisely: "football (b)" and "mad900". The format of the sequences is CIF (Common Interchange Format), with the following resolutions:

- **"**futboll (b)**"** - **352 x 388**
- **"**mad900**"** - **352 x 240**

In our case, the CIF format is suitable, since the size of the frames is a multiple of the block size (16x16) that we use. We take in consideration 25 frames, since this is the most common frame rate used in video transmission to give the illusion of continuous movement. Each of the frames extracted form the video sample, are converted into the PGM (Portable Gray Map) format. Usually, in programming projects, PGM format is considered more suitable, because of the simple data process. The results can be observed in the images below (figures 7 and 8): the original frame and the frame with the superposed motion vectors.

Figure 7. An example frame from the "futboll" test sequence

Figure 8. Motion vectors on the "futboll" test sequence

In the tables is presented the execution time for each algorithm, in CPU and GPU, according to the different number of frames.

$$Speedup = \frac{Time\_of\_serial\_a\lg orithm}{Time\_of\_parallel\_a\lg orithm}$$

To calculate the speedup we use the formula above. Results show that for the "futboll" sequence speedup from the TSS algorithm is 64 times; while 4SS brings an acceleration of 80 times. For the "mad900" sequenec, speedup from the TSS algorithm is 41 times; while 4SS brings an acceleration of 52 times.

Table 1. Execution time for "futboll" test sequence

Time (ms)	4SS – CPU	4SS – GPU	TSS – CPU	TSS – GPU
5	19.621	0.128000	26.660010	0.246000
10	31.191000	0.360000	42.456001	0.711000
15	45.654999	0.780000	62.310001	1.396000
20	59.396999	1.147500	82.778000	2.291000
25	72.928001	1.179400	102.329002	3.388000

Table 2. Execution time for "mad009" test sequence

Time (ms)	4SS – CPU	4SS –GPU	TSS –CPU	TSS -GPU
5	13.684000	0.141000	16.139999	0.243000
10	22.825001	0.366600	33.034000	0.705000
15	32.131001	0.745000	45.487999	1.397000
20	42.637001	1.215600	60.893002	2.239000
25	52.935001	2.045000	75.577003	3.389000

On one hand, execution time evaluates the GPU performance compared to CPU. On the other hand, the PSNR (Peak-Signal-to-Noise-Ratio) parameter and the MSE, are used to evaluate the accuracy of the algorithms prediction. PSNR is calculated from the following formula and it depends on MSE:

$$PSNR = 10\log_{10}\left[\frac{(MAX)^2}{MSE}\right]$$

where MAX is calculated as $2^{nr\_of\_bits} - 1$

The results from the implementation are given in the graphics (figures 9 and 10). We can see that the values for the PSNR vary in the range [22.735538;27.829149]. While, the values of MSE belong in the range [132.049225;151.492944]. The acceptable values of PSNR for the process of video compression are between 30 – 50 dB. Our results show there is deterioriation in the image quality.

6. DISCUSSION

To evaluate the performance of GPU over CPU we refer to the execution time. Speedup calculations show that GPU gives higher performance. If we compare the algorithms, 4SS brings higher speedup than TSS, since it is an improvement to TSS. The 4SS algorithm is focused on the search pattern that begins in the frame center, reducing the calculation cost. On the other side, TSS uses a search model, that is more uniform and more inclusive.

To evaluate the compression quality; in this case the quality of motion estimation process; we refer to PSNR and MSE. According to the graphics, PSNR values increase with the increase of frame number, while MSE values decrease. This means that for a larger number of frames, the error is lower and the prediction is more accurate. Regarding the quality of the prediction,

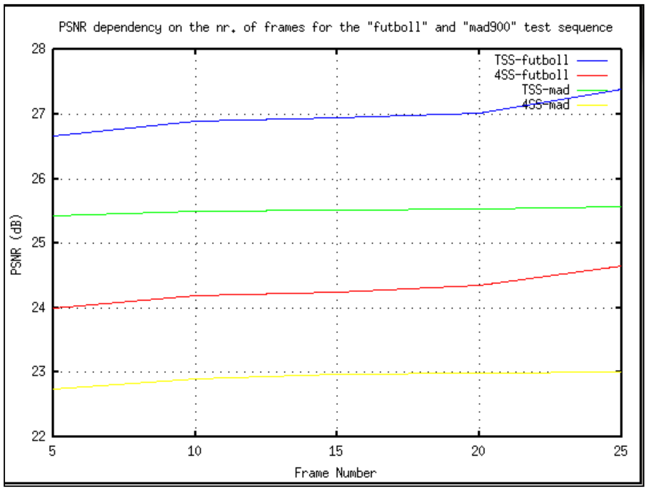


Figure 9. PSNR dependency on nr. of frames for both test sequences

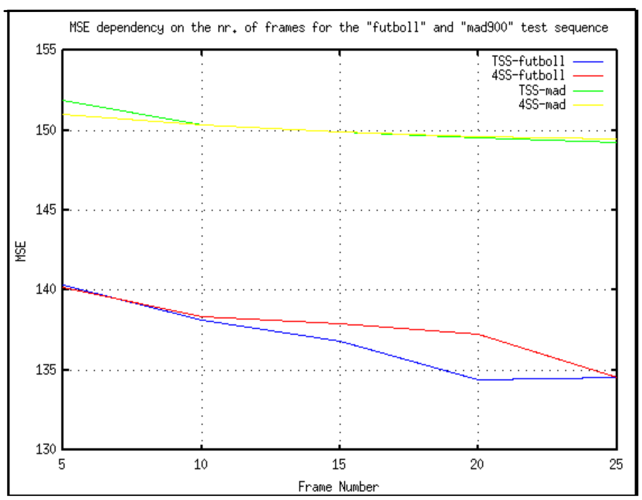


Figure 10. MSE dependency on nr. of frames for both test sequences

we observe that the algorithm that brings higher acceleration (4SS), also brings lower quality (lower values of PSNR).
This is one of the biggest dilemmas in the field of video compression: quality vs. speed. Specifically, we are more interested in the speedup of the process, since it is one of the most problematic parts. Also, one of the goals for the following phase in video compression (motion compesation), is to reduce the prediction error. So, we can expect a slight decrease in the values, as long as they are acceptable.

7. CONCLUSIONS

In this article, we studied and evaluated the performance of block-matching motion estimation algorithms, TSS and 4SS. We focused on the main problem of the process: complex and large number of calculations. The solution to this issue is parallelization. Implementing the algorithms on a CUDA capable GPU, resulted in higher performance compared to CPU. The downside is that there was a deterioration in the image quality. Even though the values were at an acceptable level, there is still need for improvement.

There is the possibility for further studies. One example could be the performance investigation of these algorithms on multiple GPU cards [17]. We would expect a linear acceleration with the growing number of GPUs. Nevertheless, there are some conditions to take into consideration. We can not know in advance what impact the overhead of data transfer between GPUs would have on the general performance. There is also the problem of GPU cards scheduling. In the end, we can say that in the future there is still work to be done in the field of implementing motion estimation algorithms on GPU cards.

8. REFERENCES

[1] Joshi, R., Rai, R. K., and Ratnottar, J. 2012. Review of different standards for digital video compression technique. In *International Journal of advancement in electronics and computer engineering* (Volume 1, Issue 1, April 2012), ISSN 2278 -1412, 22-38.

[2] Mang, H., Chou, Y., and Cheng,S.1997. Motion Estimation for video coding standarts. In *Journal of VLSI Signal Processing* (Volume 17, Issue 2/3, The Netherlands, 1997), 113-136. DOI= http://doi.acm.org/10.1023/A:1007994620638.

[3] Guttag, K. M., Gove, R. J., and Van Aken, J. R. 1992. A single-chip multiprocessor for multimedia: the MVP. In *Computer Graphics and Applications* (Texas Instrum., Houston, TX, USA, November, 1992), 53 - 64. DOI= http://doi.acm.org/10.1009/38.163625.

[4] Dutta, S., and Wol, W. 1996. A flexible parallel architecure adapted to block-matching motion-estimation algorithms. In *Journal IEEE Transactions on Circuits and Systems for Video Technology* (Volume 6, Issue 11, NJ, USA, February, 1996), 74-86.

[5] Jain, J. R., and Jain, A. K. 1981. Displacement measurement and its application in interframe image coding. In *IEEE Transactions on Communications* (Volume COM-29, Issue 12, December, 1981).

[6] Hang, H., Chou, Y. and Cheng, S. 1997. Motion estimation for video coding standarts. In *Journal of VLSI Signal Processing Systems* (Volume 17, Issue 2/3, MA, USA, November, 1997), 113-136.

[7] Limb, J., and Murphy, J. 1975. Estimating the velocity of moving images in television signals. In *Computer graphics and image processing*. (Volume 4, Issue 4, December, 1975), 311-327. DOI= http://dx.doi.org/10.1016/0146-664X(75)90001-5.

[8] Haskell, B. 1974. Frame-to-frame coding of television pictures using two-dimensional Fourier transforms. In *Journal IEEE Transactions on Information Theory*. (Volume 20, Issue 1, January, 1974), 119-120. DOI= http://doi.acm.org/10.1109/TIT.1974.1055161.

[9] Koga, T., Linuma, K., Hirano, A., Iijima, Y., and Ishiguro, T. 1981. Motion compensated interframe coding for video conferencing. In *National Telecommunications Conference.*

[10] Po, L., and Ma, W. 1996.A novel four-step search algorithm for fast block motion estimation. In *IEEE Transactions on*

Circuits and Systems for Video Technology. (Volume 6, Issue 3, June, 1996), 313-317. DOI= http://doi.acm.org/10.109/76.499840.

[11] Li, R., Zeng, B., and Liou, M. 1994. A new three-step search algorithm for block motion estimation. In *Journal IEEE Transactions on Circuits and Systems for Video Technology*. (Volume 4, Issue 4, August, 1994), 438-442. DOI= http://doi.acm.org/10.1109/76.313138.

[12] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., and Phillips, J. 2008. Graphics Processing Units - powerful, programmable, and highly parallel - are increasingly targeting general-purpose computing applications. In *Proceedings of the IEEE*. (Volume 95, Issue 5, May, 2008), 879-899

[13] NVIDIA CUDA, CUDA programming guide, version 2.3., February, 2010

[14] "The CUDA Handbook", Pearson Education, 2012

[15] "CUDA by Example", Jason Sanders, Edward Kandrot, 2010

[16] Barjataya, A. Block matching algorithms for motion estimation. DIP 6620 Spring 2004 Final Project Paper

[17] Massanes, F., Cadennes, M., and Brankov, J. G. 2010. CUDA implementation of a block-matching algorithm for multiple GPU cards

[18] *"Various Advanced Motion Estimation Research Development Package"*, Dr. L.M. Po, Dr. C.K. Cheung, Dr. C.H. Cheung, Mr. C.W. Lam http://en.pudn.com/downloads175/sourcecode/zip/detail814914_en.html

[19] Test sequences - Xiph.org Video Test Media - http://media.xiph.org/video/derf/