Published in final edited form as:
J Electron Imaging. 2011 ; 20(3): . doi:10.1117/1.3606588.

Compute-unified device architecture implementation of a block-matching algorithm for multiple graphical processing unit cards

Francesc Massanes, **Marie Cadennes**, and **Jovan G. Brankov**[1]
Illinois Institute of Technology, Medical Imaging Research Center, Chicago IL 60616, USA

Abstract

In this paper we describe and evaluate a fast implementation of a classical block matching motion estimation algorithm for multiple Graphical Processing Units (GPUs) using the Compute Unified Device Architecture (CUDA) computing engine. The implemented block matching algorithm (BMA) uses summed absolute difference (SAD) error criterion and full grid search (FS) for finding optimal block displacement. In this evaluation we compared the execution time of a GPU and CPU implementation for images of various sizes, using integer and non-integer search grids.

The results show that use of a GPU card can shorten computation time by a factor of 200 times for integer and 1000 times for a non-integer search grid. The additional speedup for non-integer search grid comes from the fact that GPU has built-in hardware for image interpolation. Further, when using multiple GPU cards, the presented evaluation shows the importance of the data splitting method across multiple cards, but an almost linear speedup with a number of cards is achievable.

In addition we compared execution time of the proposed FS GPU implementation with two existing, highly optimized non-full grid search CPU based motion estimations methods, namely implementation of the Pyramidal Lucas Kanade Optical flow algorithm in OpenCV and Simplified Unsymmetrical multi-Hexagon search in H.264/AVC standard. In these comparisons, FS GPU implementation still showed modest improvement even though the computational complexity of FS GPU implementation is substantially higher than non-FS CPU implementation.

We also demonstrated that for an image sequence of 720×480 pixels in resolution, commonly used in video surveillance, the proposed GPU implementation is sufficiently fast for real-time motion estimation at 30 frames-per-second using two NVIDIA C1060 Tesla GPU cards.

I. Introduction

Motion estimation in an image sequence has many potential uses such as detecting and tracking of moving objects in video surveillance [1], removal of temporal redundancy in video coding [2]-[4], motion compensated filtering applied along a motion trajectory in medical imaging [5], or motion compensated digital subtraction in angiography [6]. In all these applications a potential drawback is computation time needed for motion estimation.

Current applications requiring real-time motion estimation often use parallel designs for Very-Large-Scale Integration (VLSI) devices. For example, in [7][8] the block-matching algorithm was implemented on a VLSI device. It is well known that these implementations are usually costly, difficult, and time consuming to develop. Some alternative non-GPU (Graphical Processing Unit) configurable architecture approaches targeting video coding application are reviewed in [9].

[1] Corresponding author: brankov@iit.edu.

There have been a number of reported efforts to use GPU cards for motion estimation as a part of a video coding scheme, see [3] for a review. Most of the GPU implementation attempts originated before introduction of The Compute Unified Device Architecture (CUDA) [10][11], a computing engine developed by NVIDIA to facilitate easy use of the GPU. The early GPU implementations without CUDA are often complex and hard to understand. For some recent implementation of variable block size motion estimation as a part of H.264/AVC coding using CUDA, see [4], [12].

In this work we present an easy to understand, general purpose block-matching algorithm (BMA) with full grid search (FS) using CUDA computing engine and multiple NVIDIA GPU cards. Our intention is twofold: to develop a fast and accurate motion estimation for use in real-time video sequence processing and to develop a good case example for understanding the CUDA environment with use of a single or multiple GPU cards.

There are many other relevant motion estimation models (see [13] for a review), such as pixel based [14] or region based [15], even with variable region size [13]. In this work we choose a block matching model with the blocks to be equal and rectangular, however implementing different estimation models will not significantly change the presented CUDA implementation. As such, the speedup should not be diminished. In addition we chose the BMA because it has a very high computational cost; also, it is commonly used in video coding such as in MPEG and H.264/AVC [16], and as such can benefit from parallel implementation on single or multiple GPU cards. Therefore, the presented evaluation is not only a case study but a relevant implementation that one may consider using in video coding applications.

CUDA allows easy and straightforward implementation of motion estimation algorithms using standard C, with NVIDIA extensions, making program development fast. In addition, the GPU computation hardware like NVIDIA Tesla (C1060-CUDA compute capability: 1.3, released Q2 2008) delivers a staggering 933 GFLOPS in single precision at a cost of less than $1000 per single unit where six core Intel Core i7 980 XE delivers theoretical 40.0 GFLOPS (as of March 2010).

It is therefore possible to have great achievements with CUDA technology such as the computation of a shortest path in a 10 million vertex graph in less than 2 seconds [17] or an implementation of a simple color electroholography reconstruction system which are 1000× faster than the traditional computation platforms [18].

In Section II we will explain the basics of motion estimation using BMA followed by a description of GPU hardware. The algorithm design and implementation are given in Section III; Section IV contains experimental results.

II. Background

In this section we will explain, briefly the basics of the BMA motion estimation method and introduce the CUDA computational model. Knowledge in both areas is needed to fully appreciate the algorithm design described in Section III.

A. Block-matching algorithm for Motion Estimation

The block-matching algorithm [19] is the most popular method for the motion estimation [13] of local motion in an image sequence. This method essentially splits an image, of $I \times J$ pixels in size, into $K \times L$ blocks and estimates each block displacement vector $\mathbf{v}$ (also called the motion vector). For each block $B_{k\,l}, k = 1,...,K, l = 1,...,L$ this is achieved by minimizing

the matching criterion between the reference and target image over all possible candidate displacement **v** within the search window *S*, as illustrated in Figure 1.

There are several choices for matching criterion [19]. In this work, we adopted block summed absolute difference (SAD) between the reference and target image pixel defined as:

$$J_{k,l}(\mathbf{v}) \sum_{(i,j)\in B_{k,l}} \left| I^{t}(i,j) - I^{t+1}(i+v_1, j+v_2) \right|. \tag{1}$$

Here $\mathbf{v} = [v_1, v_2]^T$ is the block displacement, $I^t(i, j)$, $i = 1,2,..., I$, $j = 1,2,..., J$ represents the reference image intensity of the (i,j) pixel at a time frame t, $I^{t+1}(i,j)$ represents the target image intensity at a time frame t+1, and w_B, h_B are the block width and height.

Now we can define optimal displacement for block $B_{k,l}$ as:

$$\mathbf{v}^* = \underset{\mathbf{v}}{\arg\min} J_{k,l}(\mathbf{v}) \quad subject \quad \mathbf{v} \in S \tag{2}$$

where *S* denotes the search window of w_S, h_S pixels in width and height.

In order to make this problem computationally feasible, we will restrict the possible displacement values of **v** to a discrete regular grid (see Figure 2):

$$\mathbf{v}_{m,n} = \begin{bmatrix} \left(m - \frac{M-1}{2}\right)\Delta v \\ \left(n - \frac{N-1}{2}\right)\Delta v \end{bmatrix}, m=1,2,\ldots,M, n=1,2,\ldots,N, \tag{3}$$

where Δv is the grid step size and $M = {}^{w_S}/_{\Delta v}$ and $N = {}^{h_S}/_{\Delta v}$ are the number of grid points.

In addition, if the search grid locations are restricted to integer values, e.g. $\Delta v = 1$, then the $J_{k,l}(\mathbf{v})$ calculation will not require interpolations of $It+1(i+v_1, j+v_2)$ image values. In this work we will examine both integer and not-integer valued search grids.

Finally, in order to avoid influence of noise and false displacement vectors, e.g. in uniform regions, Eq (2) is modified to suppress motion of blocks where the SAD is too small. This is described mathematically as:

$$\mathbf{v}\% = \begin{cases} \mathbf{v}^* & J_{k,l}(\mathbf{v}^*) > w_B h_B C \\ 0 & otherwise \end{cases}. \tag{4}$$

Here $\mathbf{v}\%$ is the final $B_{k,l}$ block displacement estimate and *C* can be empirically optimized for a given type of image sequence.

Note that for the BMA a proper selection of the number of blocks and the search window size is needed. This is usually adjusted empirically, visually or quantitatively, until satisfying results are obtained. Alternatively, for the motion estimation of a known object the search window size can be estimated using expected object velocity.

B. Computational complexity of the block-matching algorithm

The block-matching algorithm's (BMA) computational complexity increases directly with the size of the search window and how the search is performed. The full search (FS) used in this work is a method that gives the best results and lowest matching error but is also the most computationally consuming implementation. Other searches such as the cross- and

three-step search [19] can reduce computational time considerably, but potentially provide less accurate motion estimation and are slightly more complicated to implement. In addition, the FS method is not image content sensitive but only image size sensitive, whereas the cross- and three-step search are sensitive to both.

The computational complexity (CC) of the BMA with FS is:

$$CC \sim O\left(K \cdot L \cdot w_B h_B \cdot \frac{w_S}{\Delta v}\frac{h_S}{\Delta v}\right) \quad (5)$$

for an image with $K \cdot L$ square blocks of w_B,h_B pixels in width and height and with $\frac{w_S}{\Delta v}\frac{h_S}{\Delta v} = M \times N$ possible candidate displacement vectors where w_s,h_s defines the search window size, Δv is the grid step size, and $M \times N$ is the total number of possible displacement vectors per block. Note that $K \cdot w_B \times L \cdot h_B = I \times J$ so that computational complexity can also be expressed as $\sim O(I \cdot J \cdot M \cdot N)$. Furthermore, the computational complexity of the methods increases considerably when the displacement candidates are considered to be non-integers due to the needed interpolation of the image values.

C. CUDA capable Graphical Processing Unit

The Compute Unified Device Architecture (CUDA) computing engine from manufacturer NVIDIA exposes powerful GPU hardware to C, C++, Fortran and other programming interfaces [10][11]. GPUs are capable of executing a high number of threads simultaneously. Furthermore, GPUs have specific hardware for floating point arithmetic, 2D and 3D matrix cached access [11]. To a programmer, a CUDA capable card is a collection of multiprocessors (30 for Tesla C1060) where each multiprocessor has a number of processors (8 for Tesla C1060), see Figure 3. Each multiprocessor has its own fast shared memory (16KB for the C1060) that is common to all the processors within it. In addition, every processor has its own fast memory registers (16384 for the C1060). Every multiprocessor shares the GPU card's global memory (4GB for the C1060) that includes texture and constant memory. In addition, each processor within multiprocessor performs cached access to texture and constant memory. The use of cache reduces the average memory access time since the cache is a smaller and faster memory which stores copies of the data from the most frequently used memory locations. In addition, by using the attached texture hardware to cached memory, one can perform linear interpolation (1D, 2D and 3D) – when this memory is accessed on non-integer location - at no added computation time.

From the program developer point of view, the CUDA model allows for a collection of functions (or “kernels” in CUDA-speak) running in parallel threads. The program developer decides the number of threads to be executed in a thread-block, and then the device will schedule the execution of the thread-blocks. This execution will start with joining thread-blocks into a grid followed by scheduling execution of a grid on the collection of multiprocessors. See Figure 4 for a visual explanation. The developer can define a thread, level 0, and a number of threads in a thread-block, level 1. Further decisions on the execution are left to the GPU hardware which attempts to group contiguous thread-blocks together, but this is not guaranteed.

In addition, in the CUDA model the threads in thread-blocks are sub-grouped in warps (a group of 32 threads). Each processor in the multiprocessor can perform, sequentially, the same operation on each thread of a warp, which makes each of them a Single Instruction, Multiple Data (SIMD) processor. Therefore, for optimal performance, the programmer should minimize thread branching so that all the threads in a warp execute exactly the same instruction to fully utilize the SIMD technology.

Ideally, for the Tesla C1060 card, if a warp uses the fastest operation, like integer addition, 8 threads from the warp are processed in one GPU cycle. In reality, an average GPU processor processes 3 threads per cycle (this is if a warp uses a floating point operation), making the top performance of 3 × 8 processors × 30 multi-processors × 1.296 GHz - 933 GFLOPS.

Each processor warp scheduler can switch content quickly and put a warp on hold during time consuming operations like global memory fetching (400-600 cycles) or cached texture memory fetching (200-300 cycles). During this hold time, while the memory is being accessed, it will attempt to execute other warps (up to 3 additional). For this reason, to fully utilize CUDA capabilities, it is important to submit a larger number of threads then the number of processors. C1060 can execute 240 (30×8) warps simultaneously and have 30720 (30×1024) active threads.

III. ALGORITHM DESIGN AND IMPLEMENTATION

A. Kernel Function

In this work we implemented a BMA [19] with an FS over all possible candidate vectors on a regular grid. The classical, serial, algorithm is very straightforward: for each block within the reference image, calculate the SAD for every candidate displacement vector and choose the best displacement as the one that minimizes the SAD.

The multicore GPU implementation has two relevant stages:

1. Code 1: Start a thread to work with quadruplet (k,l,m,n) where k and l are image block identifiers and m and n are identifiers of one candidate displacement vectors. Each thread will compute the $J_{k,l}(\mathbf{v}_{m,n})$, defined as SAD, for the $B_{k,l}$ block and the displacement candidate, $\mathbf{v}_{m\,n}$, and store it to a global memory. So for each block $B_{k,l}$, we will have a total of $M \times N$ threads computing all possible values of $J_{k,l}$, $(\mathbf{v}_{m,n})$ (See: Code 1).
2. Code 2: Next, a trivial thread is launched to find the minimum value over m = 1,2,..., M, n = 1,2,..., N, $(M \times N)$, stored values of $J_{k,l}(\mathbf{v}_{m,n})$

Global memory access is one of the main GPU bottlenecks. To minimize this, in Code 1 we use two mechanisms: 1) the reference, $I^t(i,j)$, and target, $I^{t+1}(i,j)$, images are stored in 2D cached texture memory, 2) all other variables are stored in fast register memory associated with the processor, and only one write to global memory is done at the thread end in order to store the calculated value of $J_{k,l}(\mathbf{v}_{m,n})$.

A flowchart, which schematically describes the proposed implementation, is shown in Figure 5.

The number of threads per thread-block was optimized using the CUDA occupancy calculator that is provided with the Software developer kit (SDK) from NVIDIA [10]. From the device code below it is easy to estimate that each thread will use 10-11 registers and zero shared memory. By entering these numbers into the occupancy calculator, we can obtain thread-block sizes of 128, 256, and 512 threads that will provide full (100%) GPU utilization.

B. Multi-GPU image sequence processing

In addition to evaluation of a single GPU implementation we examine two approaches for image sequence processing using multiple GPU cards. Ideally one would hope for a linear reduction in execution time with the increased number of GPU cards in use. However due to

the overhead in image data transfer and GPU cards scheduling efficiency, this will not be the case. Therefore we test two possible scenarios:

Parallel splitting—In this scenario, schematically shown in Figure 6 (top), the motion estimation threads between two consecutive frames, are grouped into super-blocks and then each super-block is submitted to be processed by a different GPU, all at the same time. Here, since each GPU processes only part of the original image pair, the total processing time is reduced.

Serial splitting—In this scenario, each GPU processes one image pair, Figure 6 (bottom), and since more than one image pair is processed at a time, the total processing time is reduced.

It is worth noting that in serial splitting the memory requirement to store the reference and target images as well as values of $J_{k,l}(\mathbf{v}_{m,n})$ for all possible displacement vectors $\mathbf{v}_{m,n}$ should not exceed GPU memory limits, where in parallel splitting this is not the case. In the experiments to follow, even when using images at a resolution of High Definition Television, we did not exceed GPU memory limitations.

IV. ALGORITHM PERFORMANCE

A. Motion estimation accuracy evaluation

First we compared correctness, visually and quantitatively, of the implemented parallel design for the Tesla C1060 GPU in respect to a serial, single CPU core design using optimization flags for Xeon E5520 @ 2.27GHz CPU (-O3 and -fno-strict-aliasing). We compared the two methods using several images at a resolution of High Definition Television (HDTV) (1920×1080 pixels) of which a pair is shown in Figure 7. In this testing, images were split into 400 blocks (20×20), each of 96×54 pixels in size, and the search window was set to double the image-block size (192×108 pixels) with a 0.5 pixel (non-integer) displacement step so that for each block we evaluated 82944 candidate vectors. The parameter C described in Eq (3) was empirically chosen to be 4.

An example of estimated motion using the serial algorithm (CPU implementation) is shown in Figure 8(left) and the output of the parallel algorithm (GPU implementation) can be found in Figure 8(right).

In the entire test set no visually significant difference in estimated motion was observed. A quantitatively small difference, on the order of the numerical precision, was found. This can be explained by the difference in numerical precision between the CPU and the GPU hardware [10]. At present the arithmetic operations in the NVIDIA GPU cards do not follow the floating-point IEEE-74 standard commonly supported by CPUs. New releases of NVIDIA GPUs (like Fermi) will use the IEEE-74 standard.

The GPU execution time for HDTV images was 7.23 seconds whereas for CPU implementation it was 8025.00 seconds or 2 hours 13 min 45 seconds, so the speedup for processing images in HDTV resolution was almost 1110 fold.

B. Multi-GPU vs. CPU execution time comparison

Non-integer search grid—Next we tested the proposed implementation using image sequences of various sizes with a fixed aspect ratio of 3:2 and 300 images in a sequence. In all experiments, the blocks are set to be 5%, the search window 10% of the image size and the search step size of 0.5 pixels.

In Figure 9 we reported the average frames-per-second (FPS) achievable for GPU and CPU implementations. First, one can see that the maximum number of FPS for CPU implementation is about 4.7fps at 130×74 pixels image, where at the same resolution one GPU can achieve around 600fps (out of the range in the presented figure). At a more common frame rate of 30fps, one GPU can process images of up to 420×280 pixels in size. Next, it is interesting to see the change in image size that GPU implementation in parallel splitting achieves at a 30fps rate: two GPUs can process 510×340 pixel images; three GPUs 549×366 and four GPUs 600×400 pixel images. In serial splitting these numbers are even better at a rate of 30 fps: two GPUs can process 516×344 pixels; three GPUs 570×380 and four GPUs 621×414 pixel images.

Next, in Figure 10, the GPU vs. CPU speedup curves are shown. It is interesting to note that for each GPU configuration in use, these curves flatten at some point. We postulate that this is the point at which the number of scheduled threads allowed by the thread scheduler reaches the maximum capacity of ~30k active threads with the use of content switching and therefore offsets memory fetching and other delays. Two other observations can be made. The sequential splitting (graph on the right) has a faster increase in performance (for images smaller than 800 pixels in horizontal size), but its performance declines after image size reaches 2500 pixels.

In addition, we also tested if the CPU execution time follows the predicted model complexity of $O(I \cdot J \cdot M \cdot N)$, and we find an excellent agreement with a small difference for small images where the data transfer time overhead is more significant than the actual computation time.

Integer search grid—For completeness of the presented analysis, and since the CPU does not have dedicated interpolation hardware, we also performed a test using an integer search grid. The GPU execution time did not change at all, due to cache memory interpolation hardware, whereas the CPU computation time was reduced approximately by a factor of five. For this comparison we resized the search window so that the number of vectors in each test case would be the same as for a non-integer grid. This will make blocks to be 10% and the search window 20% of the image size.

In all of the test examples, no visually or quantitatively measurable difference was found for the integer search grid. Two speedup curves, one for integer and one for non-integer search grid, are shown Figure 11. One can observe that the omission of the interpolation reduces speedup by a factor of 5 to about 200.

C. Comparison with other implementations of BMA for motion estimation

• **OpenCV implementation—**In these tests we used a video sequence (10 second in duration with 30 frames-per-second) of a moving car while the camera is panning in the opposite direction [20]. This allowed for direct comparison with OpenCv [21]-[23], a library developed for Intel CPUs. We consider this to be an interesting comparison since OpenCv was used recently for real-time video processing [24]. The OpenCV implements the Pyramidal Lucas Kanade Optical flow algorithm [25] over a selected number of feature points.

In our implementation we used 400 blocks to match OpenCv which uses 400 feature points. For a given sequence with 720×480 pixels per image, this yields a block size of 36×24 pixels (5% of image size). Next we modified the search window size until satisfying results were obtained in all frames, resulting in a 72×48 pixel search window (10% of image size).

For OpenCv it takes 17.7 seconds to process a 10 second video sequence producing effective 16.9 frames-per-second (fps). For the same sequence, serial splitting GPU implementation using two Tesla C1060 cards, it takes less than 10.1 seconds giving a possibility to achieve 30 fps (15 fps for one GPU card), i.e. real-time data processing. Therefore, the proposed multi-GPU implementation delivers a 1.75× speedup over OpenCV. Moreover, if one inspects Figure 12 it is evident that the proposed multi-GPU implementation provides estimated motion on a denser grid and of a higher quality.

• **H.264/AVC-SmpUHex implementation—**Another relevant comparison to assess the performance of the proposed GPU implementation is to use the CPU based Simplified Unsymmetrical multi-Hexagon search (SmpUHex, [26]) implementation used in H.264/AVC standard which we will denote CPU-H.264/AVC-SmpUHex. We used a highly optimized implementation of H.264/AVC-SmpUHex for Intel CPUs which can be downloaded at [16]. For fairness of the comparison, we use 16×16 pixels blocks and a search area of 32×32 pixels in both codes to process the sequence of the car [20] of 720×480 pixels.

Using CPU-H.264/AVC-SmpUHex it takes 33.69 seconds to compute the motion field for a 10 second video sequence producing effectively 8.9 fps, while proposed single-GPU implementation takes 23.88 seconds producing effectively 12.6 fps. Therefore, if considered as it is, the proposed singe-GPU implementation achieves a 1.41× speedup over CPU-H.264/AVCSmpUHex. However SmpUHex, as the name suggests, does not utilizes a full grid search, though in most cases it produces the similar results as FS. It was also measured that the proposed GPU implementation achieves speedup of 28× over the full search CPU implementation denoted as CPU-H.264/AVC-FS. Note that CPU-H.264/AVC-FS implementation is fully optimized and as such it is about 10× faster than our CPU FS implementation. The H.264/AVC-FS speedup is mainly achieved by pre-calculated block locations (see Code 1) where in our CPU and GPU implementation this is not done; it may be considered in future along with implementing SmpUHex on GPUs.

• **Tesla C2070—**Lastly, we have performed initial testing on a new Tesla C2070 (aka Fermi) (CUDA compute capability: 2.0, released Q2 2010). Interestingly, even thou C2070 has a peak performance of 1.03 Tflops (C1060 has 0.933Tflops), we observed decreased speedup of only 665× and 50% GPU occupancy in contrast to C1060s 1000× and 100% occupancy. This indicates that in transferring code from C1060 to C2070, one needs to carefully re-optimze implementation so as to fully utilize C2070 hardware.

D. Demo program

For the purpose of testing the proposed GPU implementation by a wider audience, we created an executable demo for Windows OS using OpenCv library. This demo program will perform video capturing and streaming, at 360×240 pixels resolution and 30 FPS, through then Internet from the users camera to our GPU server. Our server will perform real-time motion estimation and return estimated motion field which will be displayed at the user's screen. A demo can be downloaded from: http://image.mirc.iit.edu/GPUdemo/.

V. CONCLUSION

In this paper, we presented and evaluated an implementation of the block-matching algorithm for motion estimation with full search using multiple GPU cards. At this time, our implementation is suitable for processing a surveillance video at 720×480 pixel resolution at 30 fps (real-time) using two C1060 Tesla GPU cards, outperforming the same CPU implementations by several orders of magnitude.

Further, we performed a comparison of proposed full search GPU implementation with two existing, CPU optimized, implementations which do not utilize full search, namely OpenCV implementation of the Pyramidal Lucas Kanade Optical flow algorithm [25] and Simplified Unsymmetrical multi-Hexagon search (SmpUHex, [26]), implementation available in H. 264/AVC standard. The presented results show a moderate speedup of the proposed GPU implementation, indicating that both CPU methods could be reimplemented on GPUs, and one should expect a significant reduction in computation time. This remains to be explored in future work.

In addition, the work presented here provides a good case example of how to use CUDA technology to increase the performance of video and image processing methods. It is not always easy to implement methods in a highly parallel architecture; for this reason, examples like this can provide some guidance while developing other applications.

Acknowledgments

This work was supported by NIH/NHLBI grant HL091017 and HL065425. The authors would like to acknowledge David M. Stavens for his help and generously providing data used in comparison with OpenCV.

Biography

Francesc Massanes

Mr. Massanes was born in Barcelona, Spain in 1986. In 2005 he was accepted at the Center for Interdisciplinary Studies (CFIS) at UPC-Barcelona Tech, Spain, to pursue simultaneous degrees in Computer Science and Mathematics. In 2008 he was awarded with a scholarship from the Agency for Administration of University and Research to work in the department of Languages and Computer Systems (LSI). In January 2009, he joined the department of Computer Architecture as a student collaborator. In June 2009, he finished a degree in Mathematics at UPC-Barcelona Tech. In June 2010, he finished his degree in Computer Science at UPC-Barcelona Tech with the Final Thesis titled: Emulation of Human Motion Perception. Nowadays he is a Master of Science student in the Electrical Engineering at the Illinois Institute of Technology with a fellowship grant for post-graduate studies from “La Caixa”, Barcelona, Spain.

Marie Cadennes

Ms. Cadennes was born in Belgium in 1987. Since 2007, she has been pursuing a degree in electrical engineering at ENSEA, Cergy-Pontosie Cedex, France. She is currently a research scholar at the Medical Imaging Research Center at the Illinois Institute of Technology, Chicago, USA.

Jovan G. Brankov

Dr. Brankov received his diploma of electrical engineering from the University of Belgrade, Serbia in 1996. He received M.S.E.E. and Ph.D. degrees from the Electrical and Computer Engineering Department of the Illinois Institute of Technology in 1999 and 2002, respectively.

Dr. Brankov joined the Electrical and Computer Engineering Department at the Illinois Institute of Technology in 2002 as a researcher, was promoted to Research Assistant Professor in 2004, and currently he is Assistant Professor in the same department. His current research topics include 4D and 5D tomographic image reconstruction methods for medical image sequences, multiple-image radiography (a new phase-sensitive imaging method), and image quality assessment based on a human-observer model. He is author/co-author of more then 90 publications.

REFERENCES

1. Cohen I, Medioni GG. Detecting and Tracking Moving Objects for Video Surveillance. Conference on Computer Vision and Pattern Recognition CVPR (IEEE Computer Society. 19991999:2319–2325. ISBN 0-7695-0149-4.
2. Chan MH, Yu YB, Constantinides AG. Variable size block matching motion compensation with applications to video coding. Communications, Speech and Vision, IEE Proceedings. 2009; 137(4)
3. Cheung N, Fan X, Au OC, Kung M. Video Coding on Multicore Graphics Processors. IEEE Signal Processing Magazine. March; 2010 27(2):79–89.
4. Lin D, Xiaohuang H, Quang N, Blackburn J, Rodrigues C, Huang T, Do MN, Patel SJ, Hwu WM-W. The parallelization of video processing. Signal Processing Magazine, IEEE. November; 2009 26(6):103–112.
5. Marin T, Brankov JG. Deformable left-ventricle mesh model for motion-compensated filtering in cardiac gated SPECT. Medical Physics. October; 2010 37(10):5471–5481. [PubMed: 21089783]
6. Deuerling-Zheng Y, Lell M, Galant A, Hornegger J. Motion compensation in digital subtraction angiography using graphics hardware. Computerized Medical Imaging and Graphics. 2006; 30(5): 279–289. [PubMed: 16904868]
7. Baglietto P, Maresca M, Migliardi M. Parallel Implementation of the Full Search Block Matching Algorithm for Motion Estimation. Proc. IEEE Int. Conf. Application Specific Array Processors. 1995:182–192.
8. Dutta S, Wolf W. A flexible parallel architecture adapted to block-matching motion-estimation algorithms. Circuits and Systems for Video Technology, IEEE Transactions on. February.1996 6(74)
9. Vanne J, Aho E, Kuusilinna K, Hamalainen TD. A Configurable Motion Estimation Architecture for Block-Matching Algorithms. Circuits and Systems for Video Technology, IEEE Transactions on. April; 2009 19(4):466–477.
10. NVIDIA. February 2010. wepage http://www.nvidia.com/cuda.
11. NVIDIA CUDA. Cuda programming guide, version 2.3. February.2010
12. Wei-Nien, C.; Hsueh-Ming, H. H.264/AVC motion estimation implementation on Compute Unified Device Architecture (CUDA). Multimedia and Expo, 2008 IEEE International Conference on; June 23 2008-April 26 2008; p. 697-700.
13. Bovik, A. The Essential Guide to Video Processing. Academic Press, Inc; 2009.

14. Horn BKP, Schunck BG. Determining optical flow. Artificial Intelligence. 1981; 17:185–203.
15. Bourbakis NG. Visual target tracking, extraction and recognition from a sequence of image using the LG graph approach. Int. Jour. on Artificial Intel. Tools. 2004; 11(4):513–529.
16. H.264/AVC Implementation by the International Telecommunications Union available at http://iphome.hhi.de/suehring/tml/
17. Harish P, Narayanan PJ. Accelerating Large Graph Algorithms on the GPU Using CUDA. HiPC, Lecture Notes in Computer Science. 2007; 4873
18. Shiraki A, Takada N, Niwa M, Ichihashi Y, Shimobaba T, Masuda N, Ito T. Simplified electroholographic color reconstruction system using graphics processing unit and liquid crystal display projector. Opt. Express. 2009; 17:16038. [PubMed: 19724604]
19. Murat Tekalp, A. Digital video processing. Prentice Hall; New Jersey: 1995.
20. D. Stavens webpage http://ai.stanford.edu/~dstavens/
21. Lucas BD, Kanade T. An iterative image registration technique with an application to stereo vision. Proc. of the 1981 DARPA Imaging Understanding Workshop. 1981:121–130.
22. February. 2010 OpenCV, webpage http://opencv.willowgarage.com
23. Bradski, G.; Kaehler, A. Learning OpenCV: Computer Vision with the OpenCV Library. O'Reilly Media, Inc.; 2008.
24. Zhang Lei LY-P, Zhang X-F. Research of the Real-Time Detection of Traffic Flow Based on OpenCV. Proceedings of the 2008 IEEE International Conference on Computer Science and Software Engineering. 2008:870–873.
25. Bouguet JY. Pyramidal implementation of the lucas kanade feature tracker: Description of the algorithm. Intel Corporation Microprocessor Research Labs. 2000
26. Yi X, Zhang J, Ling N, Shang W. Improved and simplified fast motion estimation for JM. Proc. JVT Meeting, Tech. Rep. JVT-P021. July.2005

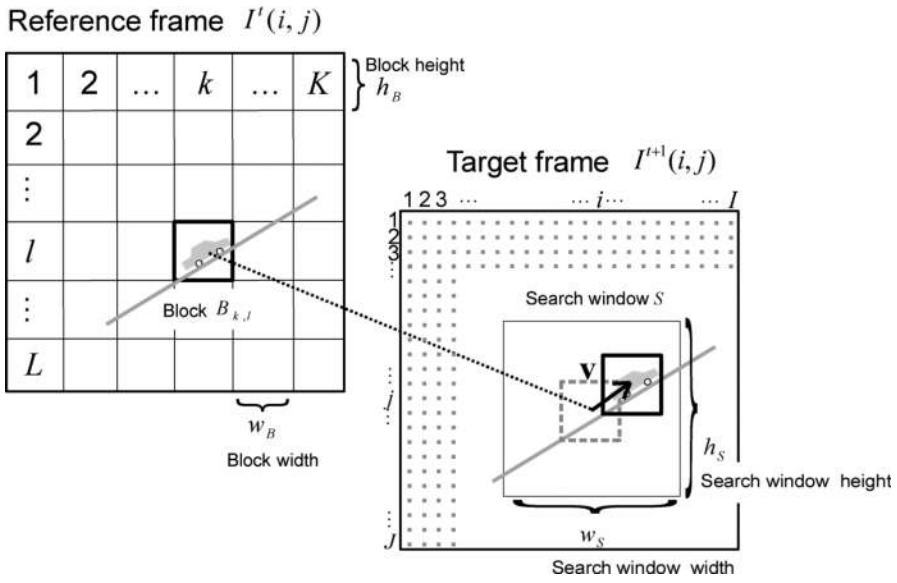

Figure 1.
Block-matching motion estimation; reference, target frame and block displacement vector **v**.

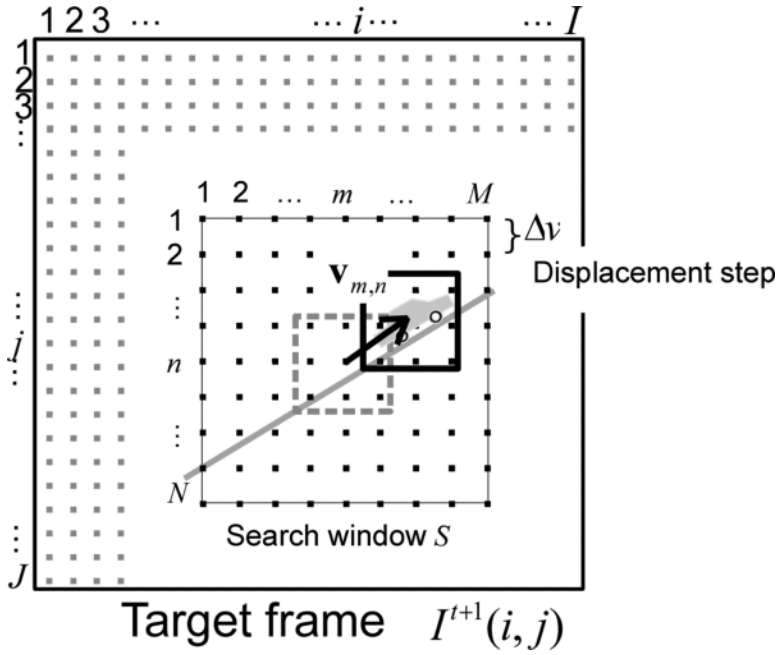

Figure 2.
Block matching methods search window and grid.

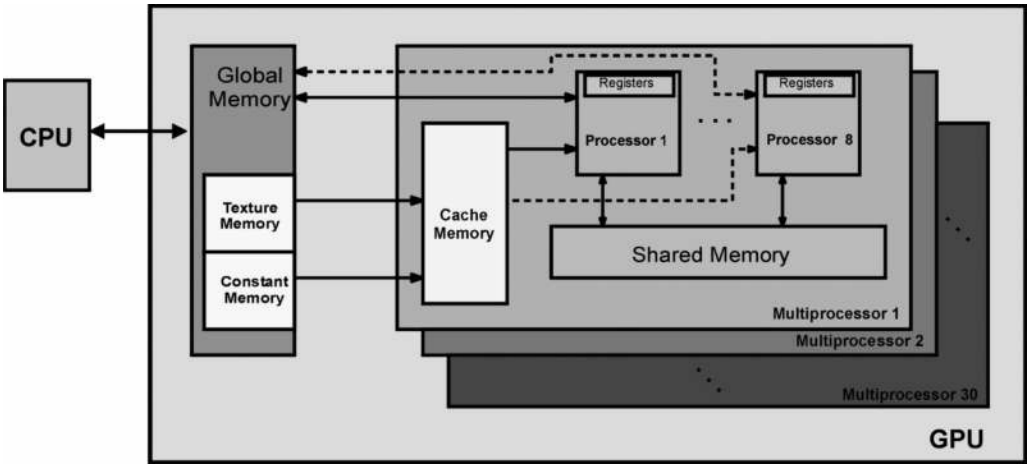

Figure 3.
Schematic of NVIDIA C1060 Tesla GPU card; Memory and processors organization.

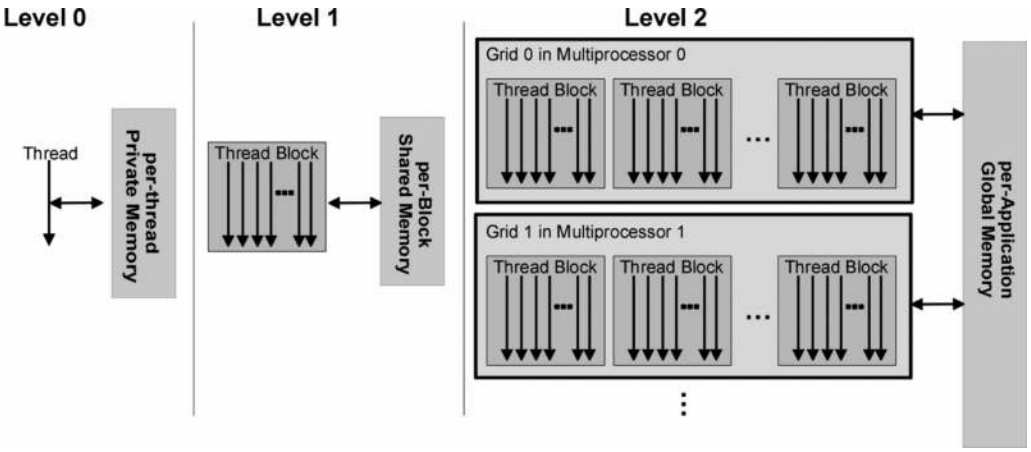

Figure 4.
CUDA hierarchy of threads, thread-blocks, grids and memory space.

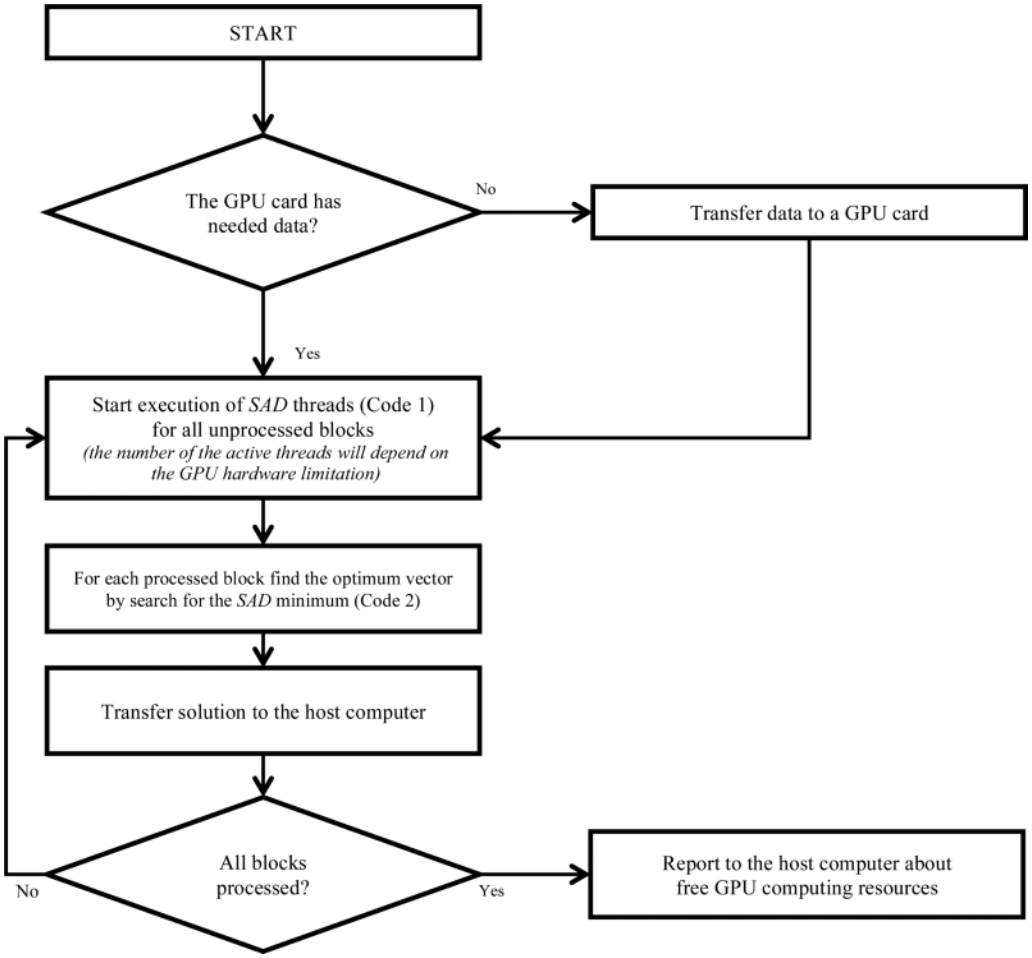

Figure 5.
Flowchart of the proposed BMA algorithm.

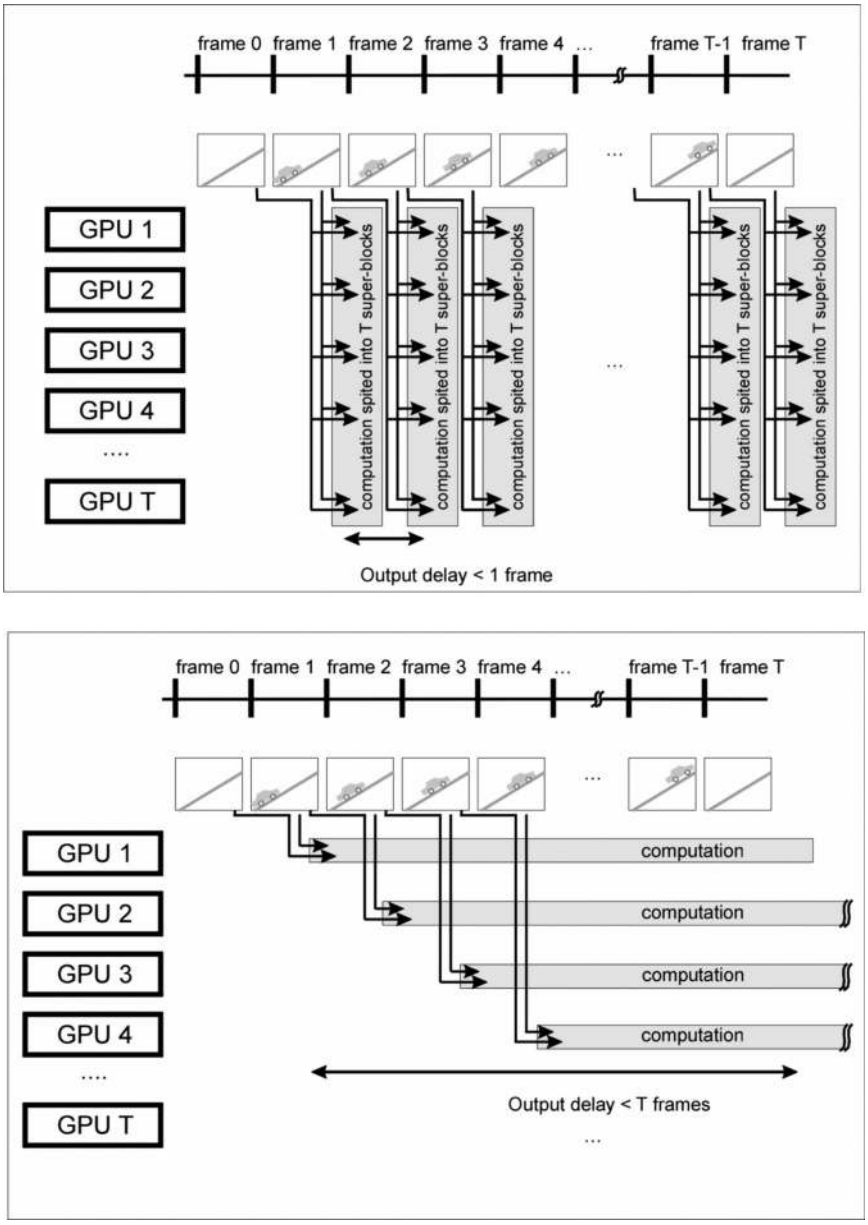

Figure 6.
Parallel splitting (Top) and serial splitting (Bottom) of an image sequence.

Figure 7.
HDTV image of 1920×1080 pixels; Reference and Target Image, respectively.

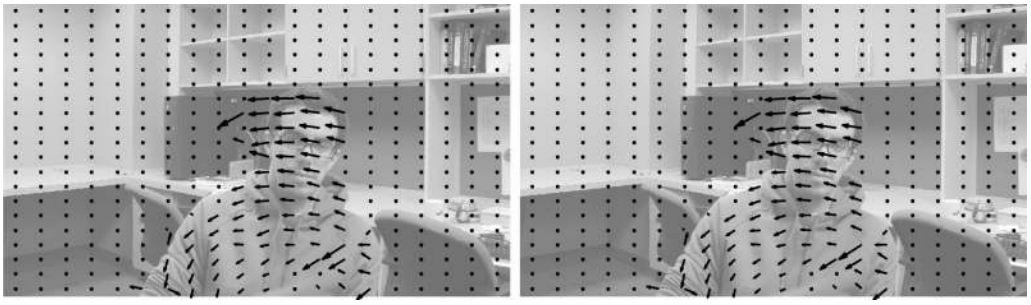

Figure 8.
Motion field estimated using CPU and GPU implementation, respectively

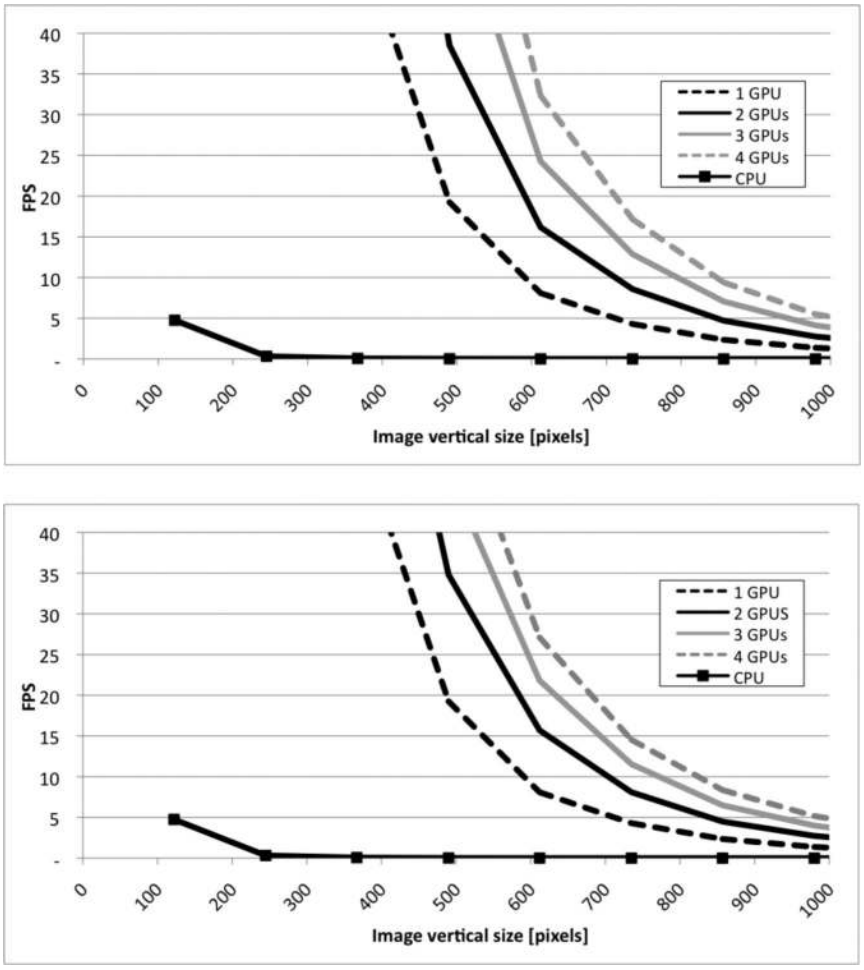

Figure 9

. Maximum achievable GPU implementation frame-rate-per-second (FPS) as a function of the frame size for: (Top) Parallel splitting and (Bottom) Sequential splitting.

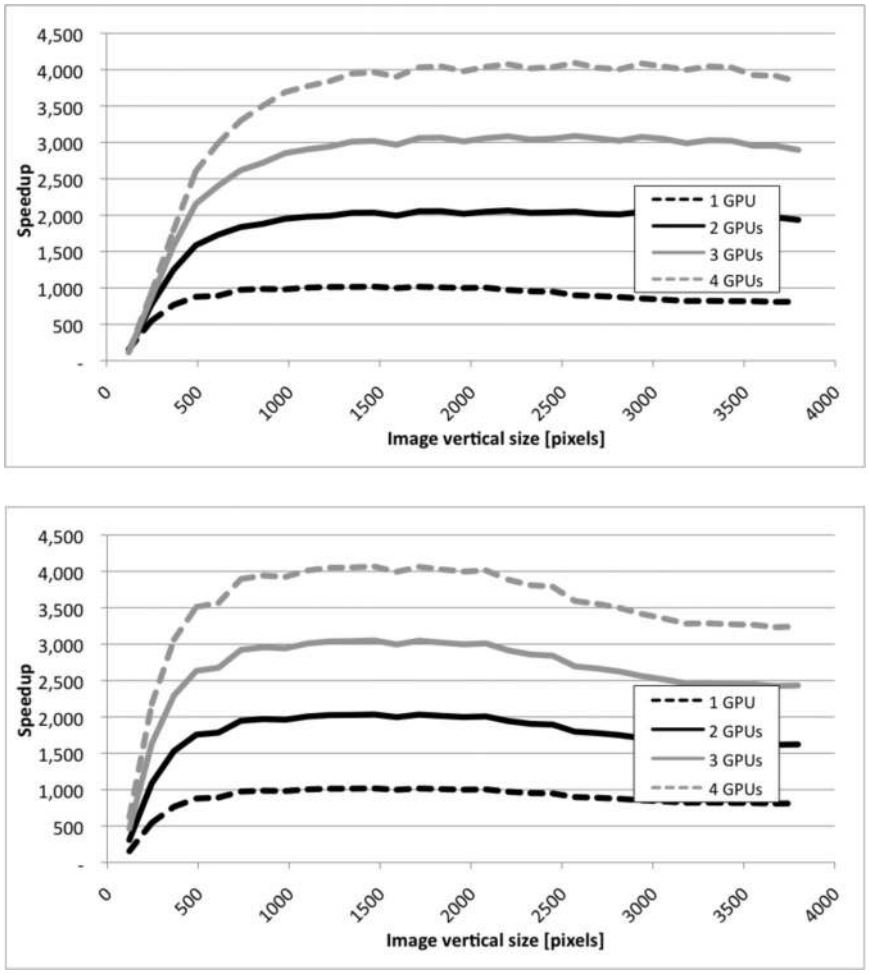

Figure 10.
Achieved GPU implementation speedup, in comparison to CPU, as a function of the frame size for: (Top) Parallel splitting and (Bottom) Sequential splitting.

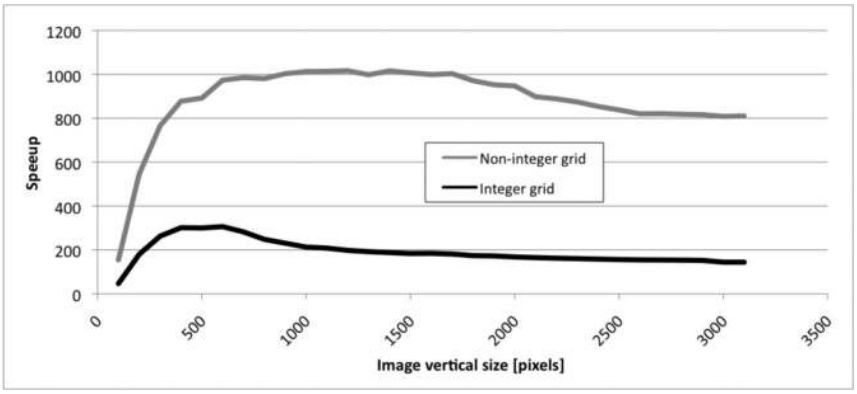

Figure 11.
Achieved GPU implementation speedup, in comparison to CPU, as a function of the frame size for integer and non-integer grid search.

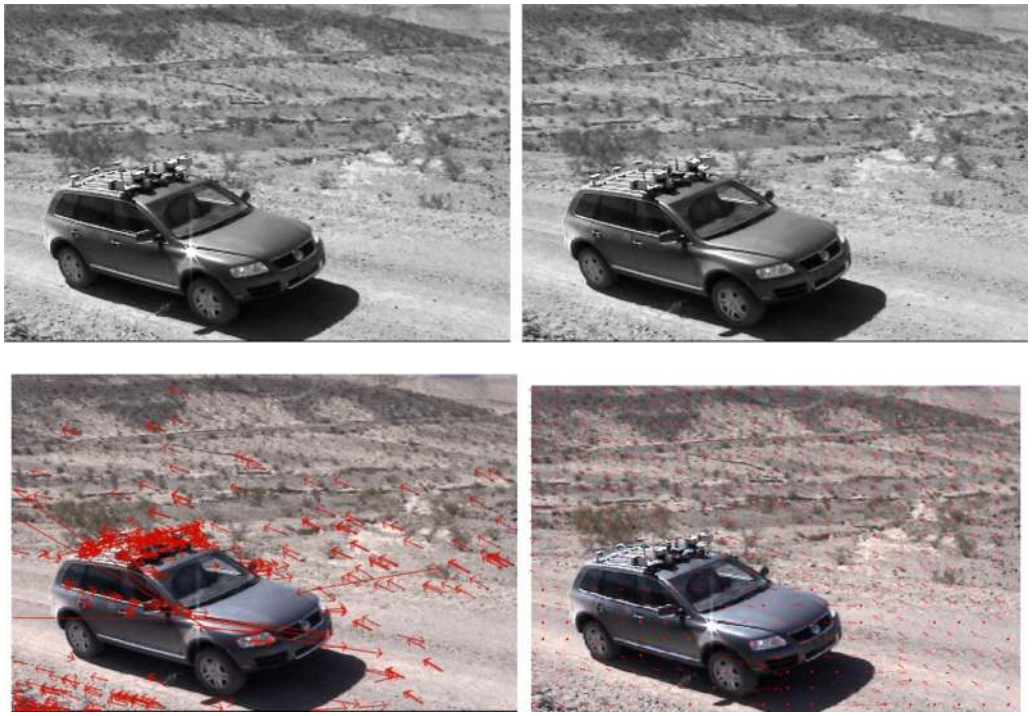

Figure 12.
Comparison with OpenCV implementation of the Pyramidal Lucas Kanade Optical flow algorithm.

Code 1: The parallel Kernel

```
//host code
int I, int J;                    // image width and height
int wB, int hB;                  // image-block width and height
int K = I/ wB;                   // number of blocks in i direction
int L = J/ hB;                   // number of image-blocks in j direction
int n_image_Blocks  = K*L;       // total number of image-blocks
float dw=0.5                      // search grid step size
int wS, int hS;                  // search window width and height
int M= wS /dw;                   // number of search grid points in horizontal direction
int N= hS /dw;                   // number of search grid points in vertical direction
int nVectors = M*N;              // total number of search grid points
int* J_kl;                       //vector containing J_{k,l}(v_{m,n}) values allocated
                                 //in global memory by cudaMallocArray
texture<float,2> Reference_image;  //allocate reference image in global memory as a texture memory
texture<float,2> Target_image;     //allocate target image in global memory as a texture memory

//device code
__global__ void exhaustiveSearchKernel (int* J_kl, int I, int J, int M, int N, int wB, int hB, int
n_image_Blocks, int wS, int hS, int nVectors)
{
// allocate variables in the register memory
volatile int idx = blockIdx.x * blockDim.x + threadIdx.x;        // idx = (K*L)* idBlock + idVector;
volatile int id_Block  = idx/nVectors + offset;                  // id_Block = L*k + I
volatile int id_Vector = idx%nVectors;                           // id_Vector = M*n+m;
if (id_Block > n_image_Blocks ) return;                          // check if this is the last block
volatile float xB = (id_Block /L)*wB + 0.5f;                     // calculates block location
volatile float yB = (id_Block %L)*hB + 0.5f;                     //
volatile float v1 = ( ( (id_Vector/M)-M/2) * wS/M ) * dw;        // calculates displacement - v(m,n)
volatile float v2 = ( ( (id_Vector %M)-N/2) * hS/N ) * dw;       //

for ( int w = 0; w < wB; ++w )
     for ( int h = 0; h < hB; ++h )
               value += abs( tex2D(Rerence_image,xB+w,yB+h) -
               tex2D(Target_image,xB+w+v1,yB+h+v2) );
                // accessing 2D cached textured memory with interpolation
J_kl[idx] = (int)value;                          // return J_{k,l}(v_{m,n}) value to a global memory}
```