# Real-Time Dense and Accurate Parallel Optical Flow using CUDA

Julien Marzat
INRIA Rocquencourt - ENSEM
Domaine de Voluceau
BP 105, 78153 Le Chesnay Cedex, France
julien.marzat@gmail.com

Yann Dumortier
INRIA Rocquencourt - ENSMP
Domaine de Voluceau
BP 105, 78153 Le Chesnay Cedex, France
yann.dumortier@gmail.com

Andre Ducrot
INRIA Rocquencourt
Domaine de Voluceau
BP 105, 78153 Le Chesnay Cedex, France
andre.ducrot@inria.fr

**ABSTRACT**

A large number of processes in computer vision are based on the image motion measurement, which is the projection of the real displacement on the focal plane. Such a motion is currently approximated by the visual displacement field, called optical flow. Nowadays, a lot of different methods are commonly used to estimate it, but a good trade-off between execution time and accuracy is hard to achieve with standard integrations. This paper tackles the problem by proposing a parallel implementation of the well-known pyramidal algorithm of Lucas & Kanade, in a Graphics Processing Unit (GPU). It is programmed using the Compute Unified Device Architecture from NVIDIA corporation, to compute a dense and accurate velocity field at about 15 Hz with a $640 \times 480$ image definition.

**Keywords:** image processing, monocular vision, optical flow, parallel processing, GPU, CUDA.

# 1 INTRODUCTION

## 1.1 Context

The perception of the environment is a necessary process in many robotic tasks. Indeed, it is used as well for navigation and obstacle detection in intelligent transport systems [2], which was the original context of this work, as for automatic video monitoring. To this end, monocular vision is a convenient solution since the camera is a low cost sensor providing rich two-dimensional information contained in a single frame. Also, the depth of each image point can be estimated from the study of two or more successive frames. The first step in any process, whatever it deals with obstacle detection [9] or object tracking, is the optical flow computation, that is an estimation of the apparent motion or the matching points between different images. Thus, we will focus on determining a dense subpixelic optical flow by only using two consecutive frames from a video sequence, so that the velocity field could be used by most of processes.

In addition, recent developments of Graphics Processing Units (GPUs) for High Performance Computing (HPC) were the main motivation of this work. Thanks to the Compute Unified Device Architecture (CUDA) from NVIDIA corporation, phenomenal enhancement have been performed (up to a one hundred factor [15]) by offloading computationally-intensive activities from the CPU to the GPU. The main requirement in order to use this promising solution is to have a highly parallelizable algorithm. In brief, this study is looking forward to provide an optical flow respecting the previously described constraints by taking advantage of parallel computing performs on a GPU.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

The paper is organized as follows. The rest of this section presents some optical flow estimation methods commonly used in real-time robotic systems. Then, section 2 describes in detail the chosen algorithm while section 3 explains its parallel computing scheme. Finally the last section presents experimental results and proposes a comparison tool for real-time optical flow algorithms.

## 1.2 Optical flow

**Definition** Optical flow is an approximation of image motion, that is the exact two-dimensional displacement given by the projection of real motion on the focal plane. Actually, the only information we have with a camera is the level of intensity of light received on cell sensor, so the gray value at each pixel. The optical flow computation consists in matching image points using this information in order to build a visual displacement field. Most estimation methods are so based on the brightness constancy assumption between two successive frames of a sequence. Given an image sequence $I(\mathbf{x},t) : \Omega \rightarrow \mathbb{R}_+$ that associates for each point $\mathbf{x} = (x,y)^T$, its intensity at time $t$, the gray value constancy is written:

$$I(\mathbf{x}+\omega, t+1) - I(\mathbf{x},t) = 0 \ ,$$

 ISBN 978-80-86943-93-0

where $\omega = (u,v)^T$ describes the apparent image velocity. The problem can also be expressed under the differential form which leads to:

$$\frac{dI(x(t),y(t),t)}{dt} = \frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} = 0\,, \quad (1)$$

that is equivalent to the optical flow equation:

$$(\nabla I)^T \omega + I_t = 0\,, \quad (2)$$

with $(\nabla I) = (I_x I_y)^T$, the intensity spatial gradient and $I_t$ its temporal derivative. This problem is ill-posed so that with the single Eq. (2), we are not able to compute the two-component optical flow. From there, a lot of methods propose different additional hypothesis in order to regularize the system. The most commonly used in robotic application, are presented in the following paragraphs without exhaustivity.

**Block Matching method** This is the historical algorithm and probably the simplest. Considering a Region Of Interest in first image (a block), the purpose is to find the displacement of this ROI in the next one. To this end, we compare the correlation scores between the original block and a family of candidates into a search area $\Omega_{ROI}$, in the second frame. Among the correlation criteria of the most often used, one can find some well-known cost functions to minimize, as the sum of absolute differences (SAD) or the sum of squared differences (SSD):

$$\begin{array}{lcl} SSD & \doteq & \sum_{\Omega_{ROI}} (I(x,y,t) - I(x+u,y+v,t+1))^2 \\ SAD & \doteq & \sum_{\Omega_{ROI}} |I(x,y,t) - I(x+u,y+v,t+1)| \end{array}.$$

To avoid exhaustive search in $\Omega$ and speed up the process, a lot of exploration algorithms have been developed [4]. However, the main problem of such a method remains its pixelic accuracy. Working on over-sampled images is a way to solve this issue, but also increases the amount of computation.

**Variational Methods** The methods based on the differential approach consists in an optimization problem resolution (local or global) by minimizing a functional containing the term (2) with an additional constraint. The historical global differential method has been developed by Horn & Schunck [4]. It aims to minimize on the whole image domain $\Omega$ the following functional:

$$J_{HS} = \int^{\Omega} ((\nabla I)^T \omega + I_t)^2 + \alpha((\nabla v_x)^2 + (\nabla v_y)^2) dx\,dy\,.$$

This criteria is based on the idea that the adjoining velocities are very similar (continuity constraint). There exists other versions of this method, using different regularization operators [2]. The main problems of that kind of methods are its high noise sensibility and the lack of accuracy: global method implies global movement so that small local displacements are not well tracked. This can be very harmful for processes that aims to detect small moving objects.

Also, local differential methods use an additional assumption on a small domain of the image to particularize the computed optical flow. The most famous local method is the algorithm of Lucas & Kanade [5] : the velocity is supposed constant on a neighborhood $\Omega_{ROI}$. Then we minimize on this domain the following functional built with the optical flow Eq. (2):

$$J_{LK} = \sum_{\Omega_{ROI}} (\nabla I . \omega + I_t)^2\,.$$

This is equivalent to the least square estimation on the neighborhood $\Omega_{ROI}$. Such a method is very interesting because of its robustness to the noise, and the local assumption makes the small motions trackable.

**Ruled out methods** Frequency-based methods, using the Fourier transform version of Eq. (2), have been developed. They go by tuned families of filter, phase [7] or wavelet models [8]. But all these methods provide sparse velocity fields or over-parametrized approaches.

# 2 ALGORITHM

The algorithm we choose according to the previously described requirements is the pyramidal implementation of the Lucas & Kanade algorithm with iterative and temporal refinement.

## 2.1 Lucas and Kanade tracker

The basis idea of the Lucas & Kanade algorithm has already been presented in section 1.2, and the optical flow estimation is based on the least-square resolution method. Considering a patch of size $n$ with an uniform velocity, and centered on the considered pixel. Thanks to Eq. (2), its displacement $\omega = (u,v)^T$ can be written as following:

$$\begin{bmatrix} u \\ v \end{bmatrix} = (A^T A)^{-1} A^T b\,, \quad (3)$$

with:

$$A = \begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \vdots \\ I_{xn} & I_{yn} \end{bmatrix}, b = \begin{bmatrix} I_{t1} \\ I_{t2} \\ \vdots \\ I_{tn} \end{bmatrix}. \quad (4)$$

In order to improve the robustness of the resolution (the least square matrix can be singular), we propose to use the regularized least-square method with the L2 norm. This finally yields:

$$\begin{bmatrix} u \\ v \end{bmatrix} = (A^T A + \alpha I)^{-1} A^T b\,, \quad (5)$$

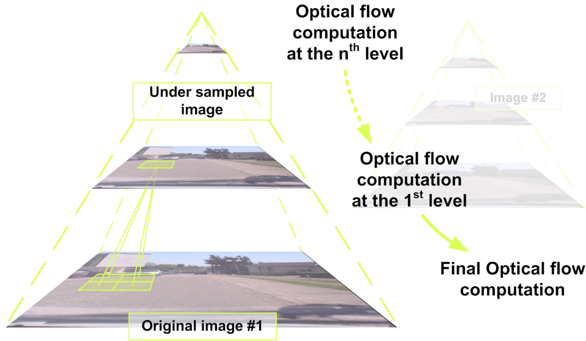



Figure 1: Pyramidal implementation.

with $0 < \alpha < 10^{-3}$ and $I$ the identity matrix. This technique avoid matrix singularity problems since the determinant is always different from zero.

## 2.2 Pyramidal Implementation

Pyramidal implementation, in a coarse-to-fine scheme enables the algorithms to track all kind of displacement. In this way, the largest movements can be identified in the lowest resolution while the original image size allows to determine the finest components of the optical flow. Let us describe briefly how this computation is performed.

In a first step, a gaussian pyramid should be built for each of two consecutive frames of the video by their successive sub-sampling. The level 0 of each pyramid is filled with the original image, the level n is then built with the image of level n-1 sub-sampled by a factor 2, and so on until the maximum level is reached. The number of levels should be determined by the resolution of the original images: typically, 3 or 4 levels represent common values for a $640 \times 480$ sequence. Then, the implementation is as follows: the optical flow is computed at the lowest resolution (*i.e.* the highest level) before being over-sampled by a factor 2, with bilinear interpolation, to be used at the lower level as initial value for a new estimate. To this end, the new research area is translated as the displacement vector previously calculated. This process continues until the 0 level is reached.

## 2.3 Iterative & Temporal Refinement

Iterative refinement is performed at every level of the pyramid. It consists in minimizing the difference between the considered frame, warped by the displacement field sought, and the next image, by executing the algorithm and transforming the destination image with the last computed flow, and this iteratively. Transforming the image means moving each point of the image with the corresponding displacement previously computed. If the displacement is not an integer, bilinear interpolation is performed to respect the real sub-pixelic motion.

The temporal optimization consists in the reusing of the computed velocity field between images $N-1$ and $N$ as an initial value for the computation of optical flow between images $N$ and $N+1$.

The final algorithm combines all the before-mentioned elements : pyramidal implementation of the Lucas & Kanade algorithm with iterative and temporal refinement. Fig. 2 shows an execution example with 3 pyramid levels.

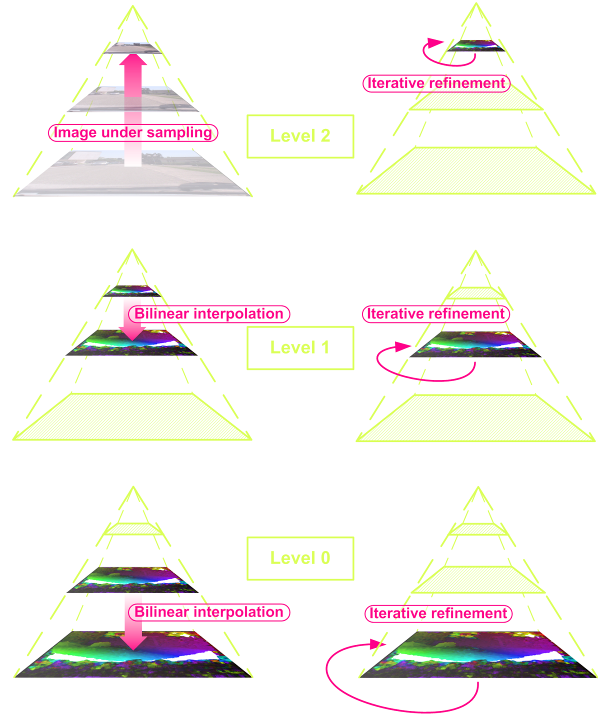



Figure 2: Execution sample on three levels

## 2.4 Parameters

The method we use has three parameters to set up : the number of levels, the number of refinement iterations per level and the size of the patch where the velocity is supposed constant. There is often a lack of information in the literature concerning parameters tuning. We propose to base the parameters according to the minimization of angular (AEE) and norm errors, respectively measured such as:

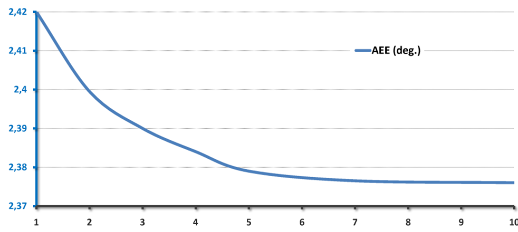
$$\frac{1}{|\Omega|}\sum_{\Omega}\arccos\left(\frac{u_c u_r + v_c v_r + 1}{\sqrt{(u_c^2+v_c^2+1)\,(u_r^2+v_r^2+1)}}\right)$$

and:

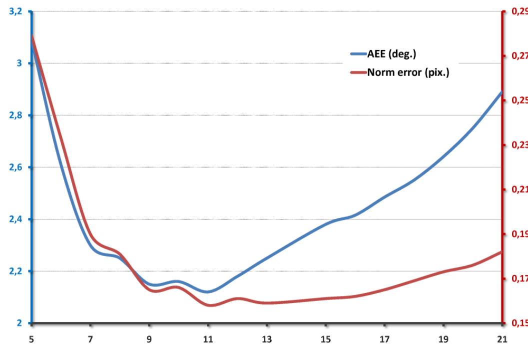
$$\frac{1}{|\Omega|}\sum_{\Omega}\sqrt{(u_c-u_r)^2+(v_c-v_r)^2}.$$

where $(u_r, v_r)^T$ and $(u_c, v_c)^T$ mean real and computed displacements. This is done using some synthetic sequences with complex motions, like *Yosemite* (Fig. 5).

 ISBN 978-80-86943-93-0




(a) related to the iteration number




(b) related to the patch size

Figure 3: Optical flow error.

Thanks to the results illustrated in Fig. 3(a), 4 iterations is a good compromise in term of accuracy, in order not to unnecessarily increase the execution time. A larger value does not improve much the flow. Also, the optimal patch size is from $9 \times 9$ to $11 \times 11$ (Fig. 3(b)). Concerning the number of pyramid levels, for a resolution of $300 \times 200$ it is useless to go over 3 levels. Bouguet [6] gives the following formula expressing the maximum trackable displacement gain related to the number of levels $l$:

$$gain_{max} = (2^{l+1} - 1)\,.$$

For the $640 \times 480$ resolution we can use up to 4 levels. In the rest of the paper the following parameters will be used: 4 levels of pyramids, a patch size of $10 \times 10$ pixels and 3 refinement iterations.

## 2.5 Results on synthetic and real sequences

The previously described algorithm has been assessed on both synthetic (Fig. 5) and real sequences (Fig. 6). In order to represent the computed optical flow, each velocity vector is encoded according to the color map

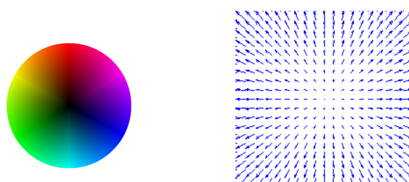


Figure 4: Color map representation equivalent vector field.

illustrated Fig. 4: the color gives the angle while the brightness represents the norm of the considered displacement. The main advantage of this representation comes from the possibility of drawing a dense optical flow, whereas a vector field representation can mask or highlights some absurd points due to its sampling. Thus, Fig. 5 illustrates the motion field performs on the Yosemite sequence with an average angular error of 2.13 deg.

The real sequence we choose involves an embedded camera taking $640 \times 480$ pictures of another car moving towards the vehicle (Fig. 6). There is obviously no ground truth for this sequence, so the comparison is made with the OpenCV implementation [6] of the same algorithm, which performs an angular error of 2.61 deg on the Yosemite sequence. The spatial aliasing present in the final OpenCV motion field, due to a bad passing through the pyramid levels, disappears from our results thanks to the bilinear interpolation, indeed while the radian flow is well retrieved. Moreover, the coming vehicle has clear borders which is promising for a detection application for example. All these aspects validate the accuracy of the chosen method.

# 3 PARALLEL IMPLEMENTATION

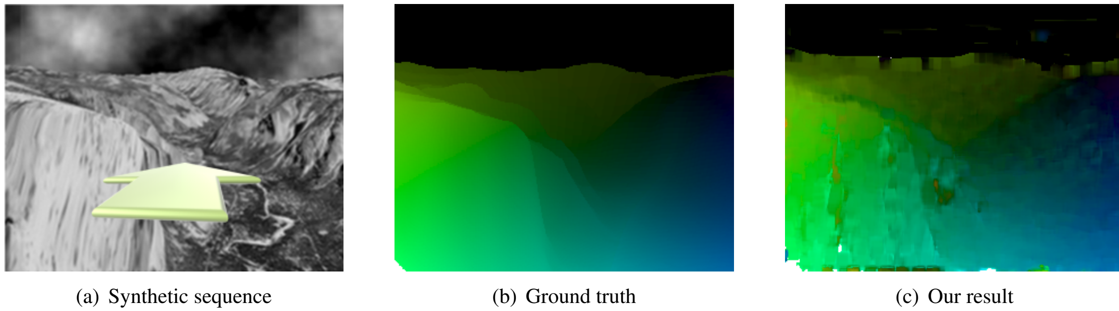
Nevertheless the execution time remains about 7 seconds with an optimized sequential implementation in C, on a 3 GHz mono-core processor. That is why in order to reach 15 Hz, or 67 ms, parallel computing must be used.

## 3.1 GPU and scientific processing

Until recently, programming was sequential with a Single Input Single Output (SISO) architecture. The development of parallel architectures is relatively new, and includes particularly the General Purpose computing on GPU (GPGPU), that is to say using existing graphical chipsets, based on a Single Input Multiple Data (SIMD) architecture, to perform intensive computation of highly parallelizable algorithms. The growing importance of such approaches has motivated NVIDIA to produce graphical chipsets allowing an access to their multi-processors as well as their registries, via a Compute Unified Device Architecture (CUDA) [10].

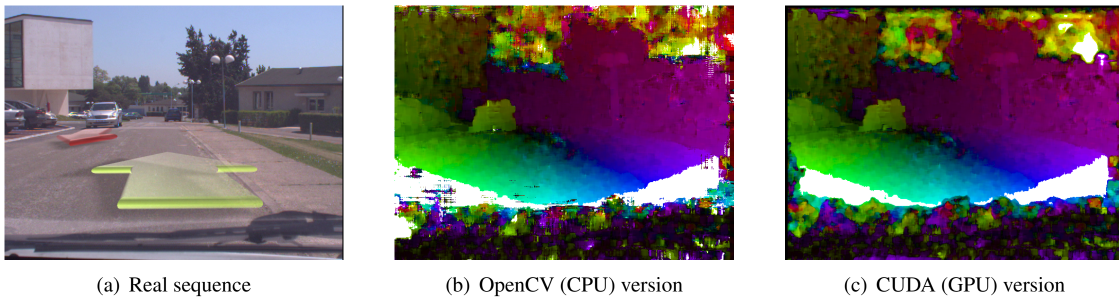
## 3.2 CUDA

**Generalities** CUDA makes possible to write software compatible with the next GPU generations. Du to technical considerations, such programs have to be organized into three levels of abstraction: elementary sequential instructions, called threads, are clustered in different blocks which are themselves divided into grids. All threads contain the same sequential instructions that are executed on different data. Each block is executed on one multiprocessor which can alternate with several other blocks in order to hide latencies



(a) Synthetic sequence (b) Ground truth (c) Our result

Figure 5: Real flow and computed flow on the Yosemite synthetic sample.



(a) Real sequence (b) OpenCV (CPU) version (c) CUDA (GPU) version

Figure 6: OpenCV and our results, on a real video sequence (green and red arrows give dominant motions).

due to some not-cached memory access. Whenever the number of blocks is higher than the number of multiprocessors available, the remaining blocks are queued. Obviously, performances depend of hardware specifications, like the number of multiprocessors or the cached memory quantity, that affect the optimal size of blocks. But an efficient implementation allows to write a program GPU-independent since the block size has not to be hard written. In that way, the only changes in executing a CUDA program on a new GPU, will be the size of blocks and their distribution over all the multiprocessors.

**Constraints** The GPU used in this study is a Tesla C870 (G80 type) consisting in 128 gathered multiprocessors and 1.5 GB of global (not cached) memory. Concerning memory into the NV G80 chipset, each thread accesses only 32 registers and each block commands 16 KB of shared (cached) memory common to all these threads. Furthermore, as memory transfers between CPU and GPU are very time consuming, it is preferable to perform all the calculations on data stored in the global memory.

About the execution, the major constraint comes from that only one kernel per GPU should be active at any time. There are also many memory constraints to consider. A Tesla card guaranties 8192 registers, which means there should be only 256 threads active at a time. Moreover the number of threads per block has to be set up, between 64 and 512, to optimize the block distribution and avoid latency. Finally, all blocks has to fit into 16 KB of shared memory.

### 3.3 Algorithm parallelization

The key idea behind the parallelization of the algorithm described in section 2, is that the optical flow computation at one pixel is independent from each other, computed at the same time. More precisely, there are four parallelizable parts in the algorithm : building the pyramids, computing the derivatives, interpolating (size doubling) the velocity fields and computing the visual displacement for each image point. Building the pyramids is both a sequential and parallel activity: it is indeed necessary to compute successively the undersampling of the images but at each level the value in the considered pixel only depends on the lower level and not on its neighborhood. The same reasoning can

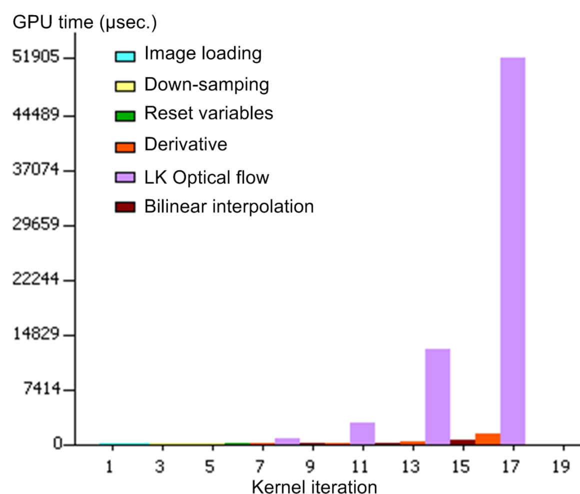



Figure 7: GPU time for the real sequence.

 ISBN 978-80-86943-93-0

be applied to the interpolation (passing from one pyramid level to another) and to the derivative computation.

Finally, the computation itself is only performed with the derivatives information and so the calculation in each pixel is independent from the neighboring pixels that is why we can use one CUDA thread per pixel. The implementation is divided into different kernels: the initialization (memory allocation), the iterative loop on the pyramid levels and the iterative refinement inside each pyramid level, which are launched sequentially by the CPU (Fig. 7). The figure 8 sums up the parallelization scheme. We can see there is not any exchange of memory between CPU and GPU during the entire process, except for loading the input images

# 4 RESULTS

### Execution time

With the CUDA implementation we obtain the same results than described in section 2.5. Execution time on the $316 \times 252$ Yosemite sequence is 21 ms per frame (47 frames per second). On the real $640 \times 480$ sequence, the execution time is exactly 67 ms per frame (15 frames per second), or an increase by a factor of 100 compared to the CPU implementation.

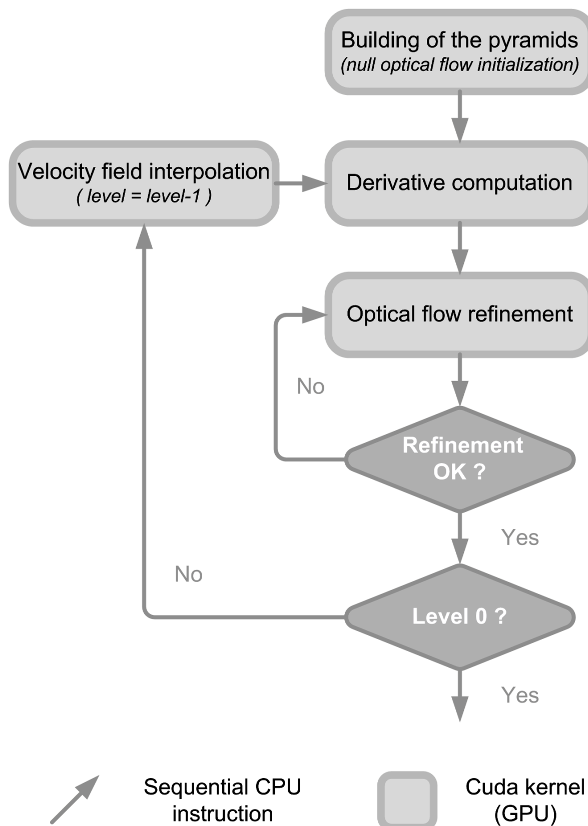



Figure 8: CUDA implementation.

| | *AAE* (deg) | *Time per pixel* ($\mu s$) | *ETATO* ($\mu s$.deg) |
|---|---|---|---|
| *Bruhn* | 2.63 deg | $0.68\mu s$ | 1.785 |
| *HSCuda* | 4.36 deg | $2.5\mu s$ | 10.9 |
| ***LKCuda*** | 2.34 deg | $0.226\mu s$ | 0.53 |

Table 1: Compared results

### Trade-off measurement

In order to compare the different implementations for the calculation of optical flow in connection with the double problem of execution time and accuracy, we propose a measurement of the Execution Time and Accuracy Trade-Off (ETATO). This number is obtained by the product of the calculation time per pixel and the angular error obtained on the well-known Yosemite sequence:

$$ETATO = exeTime/pixel.AngError$$

The best (theoretical) result that can be achieved is 0. Obviously, the less ETATO is, the best the trade-off will be.

A few authors have already addressed the problem of providing a real-time estimation of optical flow that can be use in embedded systems. The best CPU result is obtained by Weickert, Bruhn & al [12] with a dense variational method. They perform the dense optical flow for the $316 \times 252$ Yosemite sequence in 54 ms per frame with an angular error of 2.63 deg, so their ETATO is $1.78\mu s$.deg. A CUDA attempt has also been made with the implementation of the Horn & Schunck method [14]. Finally, our method (LKCuda) achieves a 0.53 ETATO level. All these results are listed in Tab. 1.

# 5 CONCLUSION

The well-known and heavy used Lucas and Kanade optical flow algorithm has been described in this study. The parallel CUDA programming model has been presented along with the parallelization of this algorithm. The obtained results are outperforming the previous attempts on real time optical flow. We achieved 15 velocity field estimations per second on $640 \times 480$ images. This opens a new way in image processing since high resolutions are not any more a constraint with parallel approaches. Though, this study uses a G80 card released at the end of year 2006. The currently most powerful GPU (type GT200, released on summer 2008) has double power and registries so that we can expect half execution times and even more in the future, always with the same CUDA program. The optical flow implementation developed in this work is voluntary unfiltered in order to be used as a basis for different image processing processes, as obstacle detection [9] for example. Finally, our work is freely available on the form of a library on the CUDA zone [15], or directly at `http://www.nvidia.com/object/cuda_home.html#state=home`.

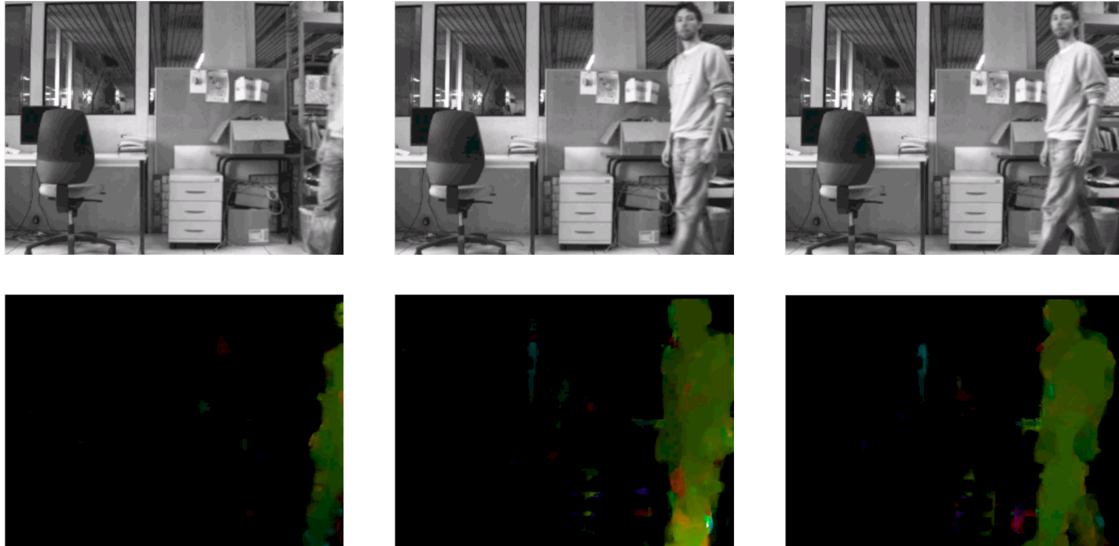


Figure 9: Video sample and its corresponding optical flow.

## REFERENCES


[1] S. S. Beauchemin and J. L. Barron, *"The Computation of Optical Flow"*, in ACM Computing Surveys (1995), Vol. 27, No. 3, pp. 433-467

[2] N. Ohnishi and A IMIYA, *"Dominant Plane Detection from Optical Flow for Robot Navigation"*, in Pattern Recognition Letters (2006), Vol. 27, No 9, pp. 1009-1021.

[3] A. Barjatya, *"Block Matching Algorithms for Motion Estimation"*, in Technical Report, Utah State University (2004).

[4] B. K. P. Horn and B. G. Schunck, *"Determining Optical Flow"*, in Artificial Intelligence (1981), Vol. 17, pp. 185-203.

[5] B. D. Lucas and T. Kanade, *"An Iterative Image Registration Technique with an Application to Stereo Vision"*, in Proceedings of the 7th International Joint Conference on Artificial Intelligence (1981) pp. 674-679.

[6] J.-Y. Bouguet, *"Pyramidal Implementation of the Lucas Kanade Feature Tracker"*, in Technical report, Intel Corporation Microprocessor Research Labs (2000).

[7] D. J. Fleet and A. D. Jepson, *"Computation of Component Image Velocity from Local Phase Information"*, in International Journal of Computer Vision archive (1990), Vol. 5 , Issue 1, pp. 77-104.

[8] Y. T Wu, T. Kanade, J. Cohn and C-C. Li, *"Optical Flow Estimation Using Wavelet Motion Model"*, in Proceedings of the 6th International Conference on Computer Vision (1998), pp._992 - 998.

[9] Y. Dumortier, I. Herlin and A. Ducrot, *"4-D Tensor Voting Motion Segmentation for Obstacle Detection in Autonomous Guided Vehicle"*, in Proceedings of the IEEE Intelligent Vehicle Symposium (2008).

[10] J. Nickolls, I. Buck, M. Garland and K. Skadron, *"Scalable Parallel Programming in CUDA"*, in ACM Queue (2008), Vol. 6, No. 2.

[11] *"NVIDIA CUDA, Compute Unified Device Architecture: Programming guide"*, in Technical Report (2008), NVIDIA corporation.

[12] A. Bruhn, J. Weickert et al, *"Variational optical flow computation in real time"*, in IEEE Transactions on Image Processing (2005), Vol. 14, Issue 5, pp. 608 - 615.

[13] Y. Mizukami and K. Tadamura, *"Optical Flow Computation on Compute Unified Device Architecture"*, in Proceedings of the 14th International Conference on Image Analysis and Processing (2007), pp. 179-184.

[14] C. Zach, T. Pock, and H. Bischof, *"A Duality Based Approach for Realtime TV-L1 Optical Flow"*, in 29th Annual Symposium of the German Association for Pattern Recognition (2007).

[15] http://www.nvidia.com/cuda


 ISBN 978-80-86943-93-0