EFFICIENT IMPLEMENTATION OF OPTICAL FLOW ALGORITHM BASED ON DIRECTIONAL FILTERS ON A GPU USING CUDA

Robert Hegner[*], *Ivar Austvoll*[+], *Tom Ryen*[+], *Guido M. Schuster*[*]

[*]University of Applied Sciences
of Eastern Switzerland in Rapperswil
{robert.hegner, guido.schuster}@hsr.ch

[+]Dept. of Electrical and Computer Engineering
University of Stavanger, Norway
{ivar.austvoll, tom.ryen}@uis.no

ABSTRACT

This paper describes an optical flow estimation algorithm using directional filters and an AM-FM demodulation algorithm, and its efficient implementation on a NVIDIA GPU using CUDA. The resulting implementation is several thousand times faster than the corresponding MATLAB code, which makes the described scheme suitable for real-time applications.

This paper also describes a new multiresolution scheme for our algorithm which allows estimating high speeds without aliasing.

The accuracy of our algorithm has proved to be comparable to the accuracy of well-known optical flow algorithms.

1. INTRODUCTION

The estimation of the apparent motion in an image sequence is used in many application like video compression, object detection and tracking, robot navigation, and so on.

In the past decades, many optical flow estimation algorithms have been developed. The ones proposed by Horn & Schunck and by Lucas & Kanade are among the most famous ones. A good overview and comparison of different optical flow estimation algorithms can be found in [1].

This work is based on an algorithm originally developed by Ivar Austvoll in [2, 3] and modified by Espen Kristoffersen in [4]. We refer to this modified algorithm as *Basic Algorithm* in this paper. A multiresolution scheme was also suggested in [2], but not implemented. For this work we have also implemented and evaluated that multiresolution scheme (which we refer to as *Pyramid Algorithm* throughout this publication).

In this paper, the focus is on the efficient implementation of these algorithms, utilizing the massive (and cheap) parallel computing power of modern GPUs (Graphic Processing Unit). With NVIDA's CUDA (Compute Unified Device Architecture), using a NVIDIA GPU for all kinds of computations has become easier than ever.

Section 3 gives a brief introduction on GPU programming using CUDA. The actual implementation of the Basic Algorithm is described in section 4.

Section 5 compares the execution times of the Matlab implementation of the Basic Algorithm with the optimized CUDA implementation.

Section 6 introduces the multiresolution scheme which allows estimating high speeds without aliasing. In section 7, the results of this new algorithm (Pyramid Algorithm) are compared with the results of some well-known algorithms.

Detailed information about the CUDA implementation, the Pyramid Algorithm and the results can be found in [5].

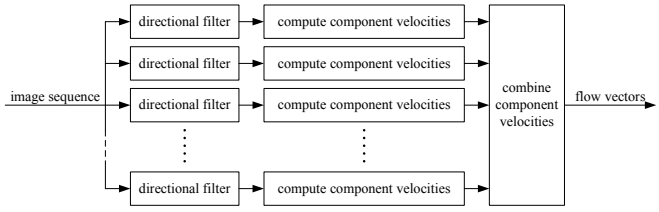

Figure 1: Basic Algorithm Overview.

This report, as well as the source code, is also available online: `http://www.medialab.ch/EUSIPCO2011/`.

2. BASIC ALGORITHM

Figure 1 shows the structure of the Basic Algorithm. Each frame in the image sequence is decomposed using a set of directional filters to reveal the structure of the image. In a second step, the *component velocity* (magnitude and sign of the velocity in a given direction) is computed for every direction. When looking at the image sequence as a three-dimensional volume, the component velocity for a given direction can be determined by slicing the cuboid in the time direction along the respective direction and examining the structure of this so called *ST-slice*. The component velocity v_c is directly related to the angle α in which features move along the time axis of the ST-slice ($v_c = \tan\alpha$). Finally, solving a linear system of equations allows finding the x- and y-components of the resulting optical flow vectors.

The directional filters are two-dimensional complex filters, separable into a longitudinal (along the filter direction) and a transversal (orthogonal to the filter direction) part. The transversal part is a real lowpass filter with a narrow bandwidth to ensure that most of the energy is in the direction of the filter. The longitudinal part is a complex bandpass filter with a wide bandwidth to capture as much energy for the given direction as possible. The longitudinal part also acts as a Hilbert transform. The output of the directional filters is therefore approximately an analytic signal and can be modelled as $z(\boldsymbol{s}) = a(\boldsymbol{s}) \cdot e^{j\varphi(\boldsymbol{s})}$ where $\boldsymbol{s}$ is a vector consisting of the coordinate s along the filter direction and the time t. $a(\boldsymbol{s})$ is the amplitude function (which is used to compute a confidence measure, but not for the optical flow estimation itself), and $\varphi(\boldsymbol{s})$ is the phase function.

The next step is to compute the phase gradient (or *instantaneous frequency*) $\nabla\varphi(\boldsymbol{s})$. A discrete 2D AM-FM demodulation algorithm is used to estimate the instantaneous frequency directly from the real- and imaginary-part of the directional filter output [6].

The instantaneous frequencies of a local neighbourhood are used to build a structural tensor. The eigenvector $\boldsymbol{e_1}$ of the structural tensor, which is associated with the smallest eigenvalue, points in the direction α of the movement of the features in the local ST-slice. Therefore, the component velocity for the given direction is $v_c = \tan\alpha = e_1(1)/e_1(2)$.

3. CUDA

CUDA is a parallel computing architecture developed by NVIDIA which allows utilizing the parallel computation power of modern NVIDIA GPUs. CUDA basically provides some extensions to the standard C language which allow kernel code (code that runs on a GPU) to be written in C. Additionally, an API to manage devices, threads, memory, etc. is provided.

CUDA kernels are executed in threads, which are grouped into thread blocks. The GPU is free to allocate thread blocks to its multiprocessors in any order. Therefore, dependencies between thread blocks must be avoided. This leads to a highly scalable architecture.

A GPU provides different kinds of device memory:

The vast bulk of the available memory is *global memory*, which is shared by all threads. Access to global memory is slow, since it is located off-chip. Whenever possible, several accesses to global memory should be coalesced to a single transaction.

Local memory is also located off-chip and therefore slow. Each thread has its own local memory. This type of memory is only used by the compiler to hold automatic variables when there are not enough registers available.

Access to *registers* is very fast. The scope of a register is one thread and the number of registers is limited.

Shared memory is much faster than global memory (about $100\times$ lower latency) and can be as fast as accessing registers. All threads in a thread block have access to the same shared memory. Care has to be taken to avoid bank conflicts, which can slow down the access to shared memory.

Constant memory can be read from all threads but can only be written by the host. Since the constant memory is cached, access can be very fast.

Texture memory is also a read-only memory which can be accessed by all threads. To use this kind of memory, a properly aligned global memory area is bound to a texture. Data can then be read by texture fetches. Texture memory is cached (optimized for spatial locality in two dimensions) and offers some interesting addressing features:

- A texture can be addressed by floating point values and return interpolated values (nearest neighbour or linear interpolation).
- Boundary cases (out of range addressing) can be handled automatically (clamping or wraping).
- A normalized addressing mode is available which allows accessing a texture with addresses in the range $[0,1]$, independent of the actual texture dimensions.
- When a texture is bound to a memory area storing integer values, these values can automatically be converted to floating point values in the range $[0,1]$ or $[-1,1]$.

More information about CUDA programming can be found in the NVIDIA CUDA Programming Guide and in the NVIDIA CUDA C Programming Best Practices Guide.

4. IMPLEMENTATION

The algorithm is implemented as a Matlab MEX file, so that it can be easily used from Matlab. A MEX file is basically a Dynamic Link Library (DLL) with the extension .mexw64 (on a 64-bit Windows system). Exporting *mexFunction*, which has a defined signature, allows Matlab to execute the code of the MEX file. Matlab libraries provide an API to interact with Matlab (*libmex.lib*) and to exchange parameters and return values with Matlab (*libmx.lib*). From the Matlab point of view, the code in the MEX file can be called like any other Matlab function.

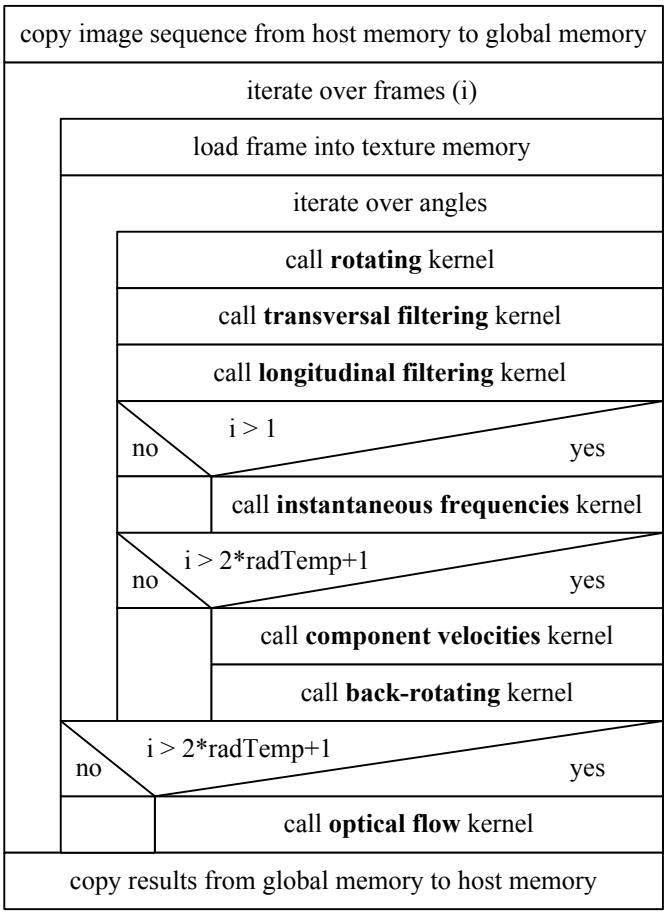

Figure 2: Implementation of the Basic Algorithm.

Figure 2 shows the structure of the (optimized) implementation of the Basic Algorithm. There are several things to note here:

- Data transfers between the host memory (RAM) and device memory (e.g. global memory) are very expensive. To reduce the number of these transactions, the whole image sequence is copied to global memory at the beginning in one transaction. All intermediate results are kept on the GPU. In the end, the resulting flow vectors are copied back to the host memory.
- To compute the instantaneous frequencies of one frame, the directional filter response of the previous, the current, and the next frame are needed. Similarly, to compute the component velocities of one frame, the instantaneous frequencies of a range of frames are needed (depending

on the temporal radius of the local ST-slice, *radTemp*).

- The directional filter is separable, allowing performing the longitudinal (along the filter direction) and the transversal (orthogonal to the filter direction) filtering sequentially.
- Instead of using a set of directional filters with different directions, the images are rotated. The results are rotated back before combining the component velocities to the final flow vectors.

The kernels are implemented to operate on a single pixel of the image sequence. The thread block size is 16×16. All image dimensions are padded to a multiple of 16 in both directions. This makes the code more readable and efficient, since no boundary checks are needed.

For the implementation of the *rotating kernel*, using texture memory is the obvious solution. The image dimension (bounding rectangle) of the rotated (and padded) image is in general bigger than the original image. However, using the clamping addressing mode of the texture memory, no explicit boundary checks have to be performed in the code. Furthermore, a bi-linear interpolation of the rotated image can be performed implicitly by the texture memory. Note that the texture is bound to each input image only once (see figure 2).

Due to the large number of filter coefficients of the directional filter (49 real coefficients for the transversal part and 15 complex coefficients for the longitudinal part), a straightforward implementation of these convolutions leads to many redundant read accesses from global memory. Two approaches to improve memory access were evaluated:

- Before performing the convolution, values are copied from global memory to shared memory, which allows fast access to these values by all threads within the thread block. This approach has to be optimized for a given filter length, but does not add any complexity on the host side.
- The second approach is to use texture memory for the source image to take advantage of its caching capabilities (which are optimized for spatial locality). This approach adds some complexity on the host side (binding the global memory to a texture), but is more flexible regarding the filter length.

Experiments showed that it is most efficient to use the shared memory approach for the *transversal filtering kernel*. Each thread first loads four values from global memory to shared memory and then performs the convolution for one output value. The *longitudinal filtering kernel* is more efficient with the texture memory approach.

The *instantaneous frequencies kernel* was mainly improved by analytically simplifying the AM-FM demodulation algorithm (see [5]).

CUDA offers vector data types (e.g. *float2*) which can help reduce the number of memory accesses by reading or writing several values in a single transaction. These data types can also be used with texture memory. A good example for this is the implementation of the *back-rotating kernel*, where each thread reads and interpolates a component velocity together with its confidence value using a single instruction.

Further improvement of the total execution time was achieved by optimizing the host code, namely by reducing the number of device memory allocation and freeing operations, kernel calls and texture binding operations.

5. SPEEDUP

The following experiments were carried out on an Intel Core 2 Quad Q6600 system with $2.4\,GHz$, $2\,GB$ DDR2-SDRAM, and Windows 7 64-bit. The GPU used (in a PCIe 1.0 slot) was a NVIDIA GeForce GTX 260, which has CUDA compute capability 1.3, 27 multiprocessors, 216 cores, and $896\,MB$ global memory.

The impact of the optimization steps described in section 4 was examined using the CUDA Analysis Tools (for the execution times of the kernels) and the CUDA event framework (for the total execution time of the algorithm) with an image sequence of dimensions $240 \times 256 \times 21$. Some parameters of the algorithm have an influence on the execution time (e.g. size of the local ST-slice). For these experiments, the default parameters of the algorithm (see [5] for details) were used.

Table 1: Timing analysis for the unoptimized CUDA implementation.

Kernel	**%**	**Time [ms]**	**Calls**
Total	100.0	212.25	1
Total Kernels	41.5	88.04	570
Transversal Filtering	16.2	34.42	84
Component Velocities	10.7	22.78	68
Longitudinal Filtering	6.6	13.91	84
Instantaneous Frequencies	3.8	8.13	76
Back-Rotating	2.2	4.60	136
Rotating	1.4	2.95	84
Optical Flow	0.4	0.89	17
Padding	0.2	0.36	21

Table 2: Timing analysis for the optimized CUDA implementation (see section 4).

Kernel	**%**	**Time [ms]**	**Calls**
Total	100.0	96.65	1
Total Kernels	59.0	57.06	481
Component Velocities	24.3	23.50	68
Transversal Filtering	15.3	14.83	84
Instantaneous Frequencies	7.8	7.56	76
Longitudinal Filtering	7.3	7.02	84
Rotating	2.0	1.90	84
Back-Rotating	1.6	1.51	68
Optical Flow	0.8	0.74	17

The timing details for the unoptimized CUDA implementaion can be found in table 1. The timing details for the optimized version (using the techniques described in section 4) can be found in table 2.

Note that a separate padding step to extend the image di-

mensions to a multiple of 16 was needed in the unoptimized implementation. In the optimized version, where texture memory is used as input for the first step (rotating), padding can be done implicitly using the clamping addressing mode of the texture memory.

By using vector data types, the number of calls to the back-rotating kernel could be reduced by 50%.

Applying the techniques described in section 4, the execution times of the kernels could be reduced by up to 67%. However, the most significant reduction in computing time (both absolute and relative) was achieved on the CPU side by optimizing the program structure.

Table 3 compares the optimized CUDA implementation with a Matlab implementation for different image sequence dimensions. Again, the default parameters of the algorithm were used. In this case, the timing was measured using Matlab. The execution times for the CUDA implementation therefore also contain some Matlab overhead for setting up the algorithm and calling the MEX file.

Table 3: Comparison of the Matlab and the CUDA implementation for different image sizes.

Image sequence size	Matlab [*min*]	CUDA [*sec*]	Speedup
$190 \times 256 \times 21$	10.3	0.124	4977
$240 \times 256 \times 21$	12.8	0.141	5434
$512 \times 512 \times 50$	343.2	1.372	15009

When performing the Matlab algorithm on the largest image sequence, Windows seems to run out of memory and starts using the swap file extensively. The optimized CUDA implementation can handle this image sequence without any problems.

The maximum image sequence dimensions that can be handled at once by the CUDA implementation depend on the chosen parameters, mainly on the number of directions. In our setup with a global memory of $896MB$ and four directions, the algorithm can handle 2344 frames of a 100×200 image sequence or 70 frames of a 500×1000 image sequence, for example. Longer sequences could be handled by splitting them into batches.

6. PYRAMID ALGORITHM

With the Basic Algorithm described in section 2, velocities of up to approximately $2\,pixels/frame$ can be estimated without aliasing. A common approach to avoid this problem is to use a multiresolution scheme, where smoothened and downsampled versions of the original image sequence are used to estimate a coarse estimation of the velocities first. These first estimates are used on the next lower downsampling level to refine the estimation (Pyramid Algorithm).

To refine the estimation on a lower level of the pyramid, the rough estimation from the next higher level is used to compensate the speed on the lower level before estimating it. In our case it is easy to compensate speeds by integer values by shearing the local ST-slice. Figure 3 illustrates this concept.

The filled circle is the current pixel, for which the velocity is being estimated. The line illustrates a feature originally moving with $1.5\,pixels/frame$ along the direction of the directional filter. This movement is then compensated by $1\,pixel/frame$ by looking at the values of a rhomboid-shaped area of the ST-slice (empty circles) instead of using a rectangular local ST-slice (dots). Estimating the velocity in this compensated ST-slice gives a velocity of $0.5\,pixel/frame$.

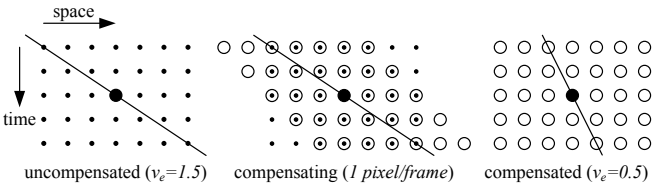

Figure 3: Compensation of high speeds.

With a downsampling factor of 2 in both spatial directions, the estimated (uncompensated) speed on a higher level is half the speed that would be estimated on the next lower level. Rounding the coarse estimates on a higher level to multiples of 0.5 therefore gives integer compensation values on the next lower level.

More illustrations and details about the actual implementation of this multiresolution scheme can be found in the original work [5].

7. COMPARISON

In the original work [5], the results of the Pyramid Algorithm were compared with other algorithms using several accuracy measurements. In this paper we present the results of the Fleets angular error measurement (in degrees) which takes both the error in angle, and the absolute speed error into account [7]. We present only mean values here. More details can be found in [5].

Table 4 compares the results of the Pyramid Algorithm with the well-known Lucas-Kanade algorithm [8] (OpenCV implementation of the multiresolution Lucas-Kanade algorithm) for nine popular image sequences. Sequences number 1 and 2 are simple sequences from the University of Western Ontario[1]. The Yosemite Cloudless sequence can be downloaded from Brown University[2]. Numbers 4 to 9 are more complex sequences with higher speeds from Middlebury College[3].

For most of the image sequences, the accuracy of the Pyramid Algorithm is better than or comparable to the accuracy of the OpenCV implementation of the Lucas-Kanade Pyramid algorithm. However, the problems of the Pyramid Algorithm in the border regions due to the large spatial support of the directional filters (and the size of the local ST-slice) were ignored by skipping the border region for the evaluation. But this fundamental problem can also be seen at other motion discontinuities, particularly in the Yosemite Cloudless sequence, where the motion is undefined on one side of the motion discontinuity.

8. CONCLUSION AND OUTLOOK

By efficiently implementing it with CUDA and extending it to a multiresolution scheme, this algorithm based on direc-

[1] `ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA/`
[2] `http://www.cs.brown.edu/~black/images.html`
[3] `http://vision.middlebury.edu/flow/`, see also [1]

Table 4: Comparison of our algorithm with the Lucas-Kanade Pyramid algorithm, using Fleets angular error in °.

No.	Sequence	**Pyramid Algorithm** CUDA	**Lucas-Kanade Pyramid** OpenCV
1	Diverging Tree	**1.322**	3.634
2	Translating Tree	0.581	**0.307**
3	Yosemite Cloudless	6.163	**3.996**
4	Rubber Whale	**8.453**	13.249
5	Grove 2	**5.224**	5.689
6	Hydrangea	**7.562**	8.336
7	Urban 3	16.105	**11.369**
8	Grove 3	12.556	**12.483**
9	Urban 2	28.559	**18.692**

tional filters became suitable for real-world and real-time applications.

Recent publications such as [9, 10] claim that realistic speedups for a GPU implementation compared to a CPU implementation are in the range of at most 10 times. In their argumentation they refer to the inherent restrictions of current GPU architectures, such as:

- Performance loss when accessing global memory in a non-coalesced manner.
- The bandwidth bottleneck when transfering data between CPU memory and GPU memory.
- The limited amout of fast on-chip memory (shared memory vs. cache).
- Weaker single-thread performance.

However, the severity of these limitations is highly dependent on the type of algorithm at hand. For our algorithms, the limitations for accessing the global memory were avoided by cleverly using shared memory and texture memory. Furthermore, the transfers between host and GPU were reduced to a minimum. Single-thread performance is not an issue in our current implementation.

As a future work it would still be necessary to compare the CUDA implementation with an optimized C implementation (instead of MATLAB).

As shown in section 7, the accuracy results of our algorithm are good. However, there is still some room for improvement:

- Due to the large spatial support of the directional filter, the algorithm has some problems handling motion discontinuities, particularly in the border regions of the image sequence.
- In downsampled image sequences it is more difficult to estimate the velocities reliably due to the smoothing of the image features. Erroneous estimates from downsampled levels are then propagated to the full-resolution level. Here, a more sophisticated postprocessing for the rough estimates should be considered.

REFERENCES

[1] S. Baker, D Scharstein, J.P. Lewis, S. Roth, M.J. Black, and R. Szeliski, "A database and evaluation methodology for optical flow," MSR-TR 2009-179, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA, 2009.

[2] Ivar Austvoll, *Motion Estimation using Directional Filters*, Ph.D. thesis, Stavanger University College (HIS)/The Norwegian University of Science and Technology (NTNU), PoBox 2557 Ullandhaug, N-4091 Stavanger, Norway., 1999.

[3] Ivar Austvoll, "Directional filters and a new structure for estimation of optical flow," in *Proc. IEEE Int. Conf. Image Processing*. 2000, vol. II, pp. 574–577, IEEE Signal Processing Soc. Los Alamitos, CA, USA.

[4] E. Kristoffersen, I. Austvoll, and K. Engan, "Dense motion field estimation using spatial filtering and quasi eigenfunction approximations," in *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, 11-14 2005, vol. 3, pp. III – 1268–71.

[5] Robert Hegner, "Efficient implementation and evaluation of methods for the estimation of motion in image sequences," M.S. thesis, University of Stavanger, Norway, 2010.

[6] J.P. Havlicek, D.S. Harding, and A.C. Bovik, "Discrete quasi-eigenfunction approximation for am-fm image analysis," in *Image Processing, 1996. Proceedings., International Conference on*, 16-19 1996, vol. 1, pp. 633 –636 vol.1.

[7] David J. Fleet and Allan D. Jepson, "Computation of component image velocity from local phase information," *International Journal of Computer Vision*, vol. 5, pp. 77–104, 1990, 10.1007/BF00056772.

[8] Bruce D. Lucas and Takeo Kanade, "An iterative image registration technique with an application to stereo vision," in *IJCAI'81: Proceedings of the 7th international joint conference on Artificial intelligence*, San Francisco, CA, USA, 1981, pp. 674–679, Morgan Kaufmann Publishers Inc.

[9] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure, "On the limits of gpu acceleration," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, Berkeley, CA, USA, 2010, HotPar'10, pp. 13–13, USENIX Association.

[10] Rajesh Bordawekar, Uday Bondhugula, and Ravi Rao, "Believe it or not!: mult-core cpus can match gpu performance for a flop-intensive application!," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, New York, NY, USA, 2010, PACT '10, pp. 537–538, ACM.