Imperial College London

Department of Computing

GPU based real-time optical flow computation

by

Clément Moussu [C.M.]

Submitted in partial fulfilment of the requirements for
the MSc Degree in Advanced Computing of Imperial College London

September 2010

Abstract

This document is a report for the individual project submitted in partial fulfilment of the requirements for the MSc degree in advanced computing of Imperial College London. Recent developments around parallel optical flow computation motivated us to develop – more than a simple library – a prototype for a free and open-source real-time optical flow computation platform. We developed, at first, a library that provides real-time optical flow computation routines using GPU parallelisation and a platform, built on top of it, includes tools and interfaces to allow people with only a small computing background to use it. The library gives good results in terms of efficiency – between 10 and more than 60 frames per second depending on the algorithm used – and average results in terms of accuracy – 14.53° of average angular error at best for Yosemite sequence. The command-line tools, `flow-compute` and `flow-tool`, are combined in makefiles wrappers for an automatic flow generation, accessible to any person who is able to deal with command-lines. Finally the `flow-studio` prototype application gives an insight of a graphical user interface for flow computation. This detailed report provides all the theoretical and implementation details of the library and tools.

Acknowledgements

I acknowledge, amongst others, **Dr. Andrés Bruhn** and the **nvidia** company for credited extracts of their works used or summarised in this report.

I would like to thank **Dr. Daniel Rueckert** for giving me the opportunity and responsibility of working on such an interesting, technical and demanding subject. The confidence he seemed to have in my abilities and the freedom of action he let me enabled me to give my best and to be creative on solutions to be found.

This project could not have given any results without all the help given to me by **Dr. Luis Pizarro**, and I would like to extend my heartfelt gratitude to him. His wide knowledge and perfect teaching skills helped my fast understanding of complex concepts that would have been much more time-consuming to comprehend without him. He constantly encouraged my ideas and challenged me to implement new features, never accepting less than my best efforts. His sense of humour and humanity also helped me to always keep a good mental during that long project.

I thank **Dr. José Delpiano** for his help on multigrid solvers and *CLG* method comprehension. I also want to thank **Dr. Richard Newcombe** for his help on getting better performances with *CUDA* and the time he spent on giving me solutions for some of my problems.

I thank **Alexandra Frommlet** who, in addition to the revision and correction of this report, supported me during the entire project and always trusted in me.

Finally I give special thanks to my fellows and friends **Matthieu Oviedo** and **Olivier Jolit**. Matthieu for his idea of using makefiles for flow generation, Olivier for his idea of blending sequence images with color flow representations, both for our frequent – stormy – debates about computer science and the interest they showed for my project.

I want to thank the **open-source community** for providing all the great tools that I used for this project: `GNU Make`, `gcc`, *ImageMagick*, `gnuplot`, `vim`, LaTeX, Ti*k*z & PGF... And more generally to make our life easier faster and more organised.

See also sequences credits in appendix C.

Contents

List of Figures

Introduction

Many scientific fields such as medicine, biology or computer vision require to track objects in video sequences. Optical flow problem aims at finding the displacement of some features over successive frames of a video sequence. To this end the same features have to be identified on different images. Then a displacement can be computed between the previous and the new location of a feature. Features may move, appear or disappear over time which complicates a lot the task. Moreover issues such as noise, illumination and shadows make this problem impossible to be solved exactly.

Several approaches and models have been developed around optical flow computation. Block-based models are the simplest approach to the problem but provide poor results. Continuous models were then introduced in improve quality of results and to be able to compute dense optical flows. Dense means that one value can be computed for each pixel of a sequence. A branch in continuous models, called variational methods, regroups all the methods that perform constrained minimisation of continuous energy functions. This strategy involves the numerical resolution of coupled systems of partial differential equations which is computationally demanding. This large amount of computation results in high quality results.

Image frames are in most of applications acquired at a rate of more than thirty images per second. This lets less than thirty milliseconds to one processing algorithm – practically some – to be executed. Computation of optical flow using best methods being time-consuming, such applications cannot afford using these methods and can only use basic algorithms getting poor results.

Parallelisation allows some algorithms to run faster by performing many tasks at the same time. Obviously it also increases the necessary amount of computational power to supply. Graphics processing units or GPU processors are highly parallel and specialised computing units that provide more computational power than a classic CPU processor. Originally conceived to achieve computer graphics acceleration, they have become more and more efficient for general-purpose applications. Some simple and repetitive tasks can then be executed faster on a GPU than on a CPU. Due to the slowness of variational optical flow computation, researchers began to focus on parallel implementations for their algorithms. Some of them used GPU processors.

The objective of this project is then to study how variational optical flow methods can take advantage of parallelisation using GPU processors. We decided to do this by developing parallel implementations for some of the state of the art variational methods, targeting real-time biomedical applications. To be able to deal with real-time situations, we will focus more on efficiency of algorithms than on accuracy. Thus we will optimise them with efficiency as first priority even at the expense of accuracy. Eventually the accuracy has to stay as small as possible for the flow computed to be usable.

Recent publications about parallel approaches for optical flow computation showed some really good results in terms of both accuracy and efficiency[16] (some non-parallel approaches too[10][6]). But we noticed that the source code of these implementations is never provided. That motivated us to develop

– more than a simple library – a prototype for a free and open-source real-time optical flow computation platform. The library should provide real-time optical flow computation routines using GPU parallelisation. The platform, built on top of it, should include tools and interfaces to allow people with a small computing background to use it.

In a first chapter we will introduce correspondence problems to then define more formally one of them: the optical flow problem. We briefly mention the issues related to this problem and the different kind of approaches.

In chapter 2 we detail the theoretical aspects of different optical flow computation methods. Starting with block-based approaches, this chapter continues with continuous models and succinctly sets the theory of variational methods we decided to implement. It also develops the theory of the solvers used giving some pseudo-code algorithms. This chapter tries to summarise some parts of Andrés Bruhn's PhD thesis[9], whose huge work contributed a lot to the field of variational optical flow computation.

The chapter 3 is about *nvidia* GPU processors and *CUDA* application programming interface. After summarising GPU processors history, we describe their architecture and introduce kernel functions. We also give details about the GPU used for our experiments and important rules of GPU programming that we tried to follow in our implementations.

Then the chapter 4 details the implementation choices of the GPU algorithms implemented in the library. It quotes some selected pieces of code used in our application and illustrates with figures the images generated at different steps of our algorithms.

The chapter 5 gives an overview of the tools forming our real-time optical flow computation platform. Starting with the main executable `flow-compute` which is a command line tool for flow computation, it also details `flow-tool` and the makefiles wrappers for automatic flow generation. Eventually it mentions a proof of concept of graphical user interface `flow-studio` that could be shipped in the platform.

We finally present and discuss our results in chapter 6 before concluding on this project.

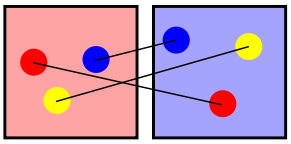
Chapter 1

Optical flow

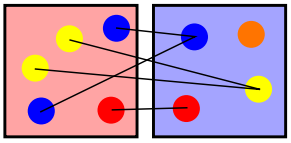
1.1 Correspondence problems

Many computer vision algorithms involve the establishment of a correspondence between identical features in several images. We can classify those problems in four categories. Assuming two sets of items with the same size, these categories are:

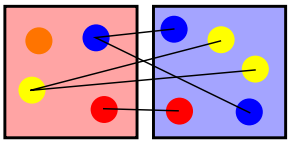
One-to-one: Each item of the first set has one and only one pair in the second set.



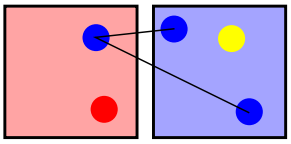
Many-to-one: Many items of the first set have the same pair in the second set. Items of the second set may not have a pair in the first.



One-to-many: One item in the first set have many pairs in the second set. One item in the first set may not find a pair in the second.



Nondense: Some items in both sets may not have any pair.



1.1.1 Stereo vision

Stereo vision aims at achieving a 3D reconstruction of a scene given several images taken at different positions. The position of a given feature point has to be found in all images: this is not always the case. It can be hidden by the object it belongs to or by another object. This is what we call occlusion. Knowing the position of the cameras and the position of feature points on each image, it is possible to compute the depth of this point. Stereo vision is an example of *nondense* correspondence problem as features may be occluded in some images. Figure C.5 gives an example of a depth map computed with several images.

Figure 1.1: Example of stereo vision [Credits: Middleburry university]

1.1.2 Particle image velocimetry

In *particle image velocimetry* one wants to determine the instant velocity vector of particles in an image sequence. The correspondence that has to be established here, is the correspondence between one particle in the first image and the same particle in the second one. That has to be done for each particle in the sequence, this is a *one-to-one* correspondence problem. Figure 1.2 shows a flow field that could represent moving particles.

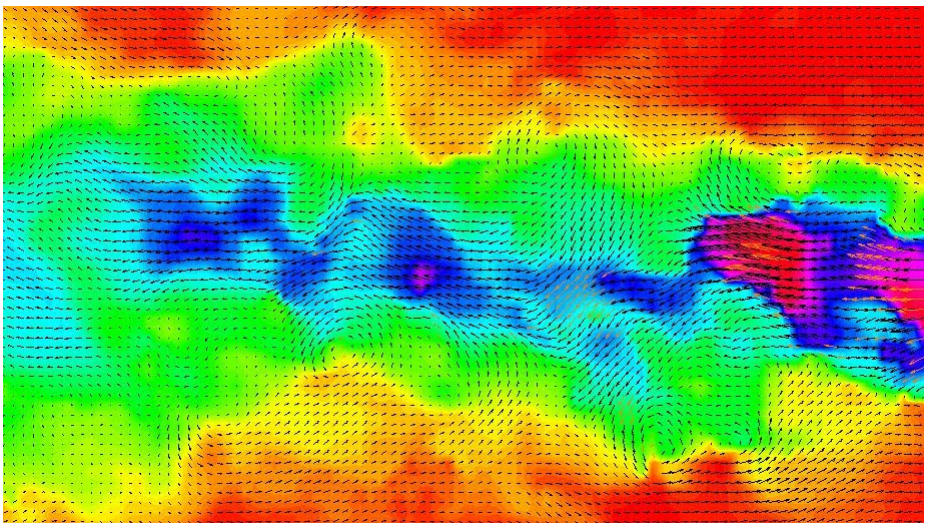


Figure 1.2: Example of particle image velocimetry [Credits: PIVlab]

1.1.3 Medical image registration

In *medical image registration* people want to align several images so that corresponding features can easily be related. The images are acquired with different sensors, most of times not a standard camera.

The aim may be to fit an image with a computer model, to fit together scans of the same patient at different times or steps of the disease, or to align an image with locations in physical space. Because of motion, growing, shrinking, rotation, occlusion, this can involve all kinds of correspondence problems. Figure 1.3 is an example of medical image registration.

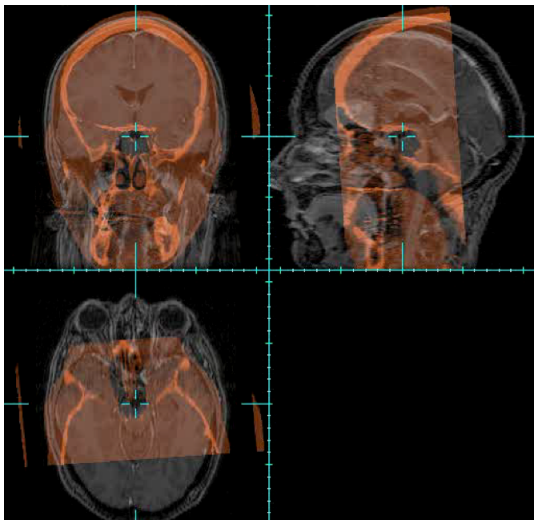


Figure 1.3: Example of medical image registration [Credits: Daniel Rueckert]

1.1.4 Optical flow

Finally *optical flow*'s goal is to track an object in a sequence of images. To be able to do that, the displacement of some pixels (most of time each) has to be computed for each pair of images, basically assuming that the grey values at the old and new locations are similar. There are plenty of applications for optical flow: robotics and computer vision, driving assistance, video compression, surveillance and security, virtual reality, medicine and surgery... As occlusion can occur between two images this is often a *nondense* correspondence problem. The figure C.6 gives an example of optical flow computation between two images.

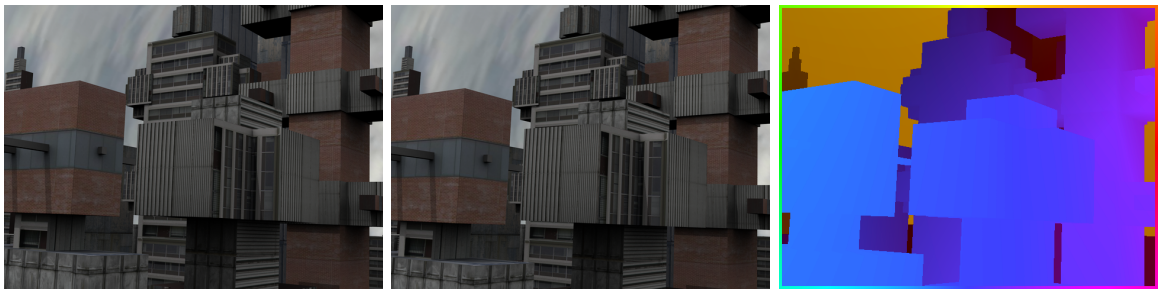


Figure 1.4: Example of optical flow [Credits: Middleburry university]

1.2 Optical flow problem

1.2.1 Grey value constancy assumption

Given a sequence of images we want to compute the displacement $\mathbf{u} = (u, v)^\mathsf{T}$ of some feature points between two image frames taken at t and $t+1$. We call $I(x, y, t)$ the intensity (grey value) at location

(x, y) in the image frame taken at time t. The grey value of the pixel at its new position is supposed to be the same. Thus we have:

$$I(x, y, t) = I(x + u, y + v, t + 1) \tag{1.1}$$

This equation is known as the **grey value constancy assumption**.

1.2.2 Linearisation

We assume the displacement to be small during one unit of time, which is often the case in a video. Thus we can develop the right hand of the grey value constancy assumption using Taylor series. This step is called *linearisation*:

$$\begin{aligned} I(x + \delta x, y + \delta y, t + \delta t) &= I(x, y, t) \\ &+ \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} \\ &+ \mathcal{O}(x) + \mathcal{O}(y) + \mathcal{O}(t) \end{aligned} \tag{1.2}$$

1.2.3 Brightness constancy assumption

Using linearised grey value constancy assumption we can deduce the following equation:

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \tag{1.3}$$

It is known as **brightness constancy constraint equation** (BCCE) and is the starting point [1] of all continuous models. It can be rewritten as:

$$\nabla I^{\mathsf{T}} \cdot \mathbf{u} = -I_t \tag{1.4}$$

1.3 Issues to deal with

Given the description made in 1.1 there are plenty of issues to solve to be able to compute an accurate and reliable optical flow. This section explains some of these issues.

1.3.1 Occlusions

Occlusion is the first issue that comes in mind when dealing with optical flow. Depending on the motion of an object, some feature points can disappear or show up. If a feature point we were tracking suddenly disappears, the most similar grey value can be far away in the image: a wrong correspondence is then established. The algorithm has to detect and penalise such wrong matchings. The same problem occurs when new objects suddenly appear in a scene.

[1] some techniques which handle large motion do not perform the linearisation step

Figure 1.5: Occlusion example: while moving, the cars occlude each other [Credits: unknown]

1.3.2 Illumination

Conditions of illumination can change and make tough the grey or color values matching. It can also come from the camera that achieves an automatic brightness adjustment or the intensity perceived by a scanner (depends on the distance between it and the patient)... The shadows are also a big problem because they can generate locally both illumination and occlusion problems. The grey value of a given region can suddenly significantly decrease because of shadows.

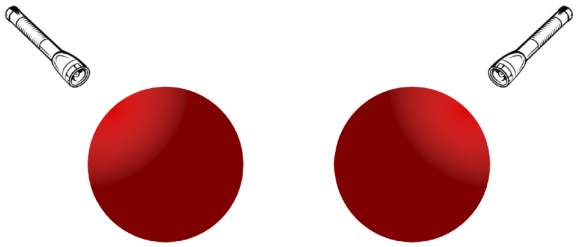


Figure 1.6: Illumination and shading example [Credits: Werner Trobin]

1.3.3 Assumptions on motion

A-priori assumptions are often made on the motion. For instance, if the camera moves we can assume that each pixel in the image sequence will have the same motion. Some parts of one image can be prone to one kind of motion more that another, for instance a road with cars. We can sometimes assume that the motion at one pixel will be really close to its neighbour. This is why arbitrary motion is a complicated problem to address.

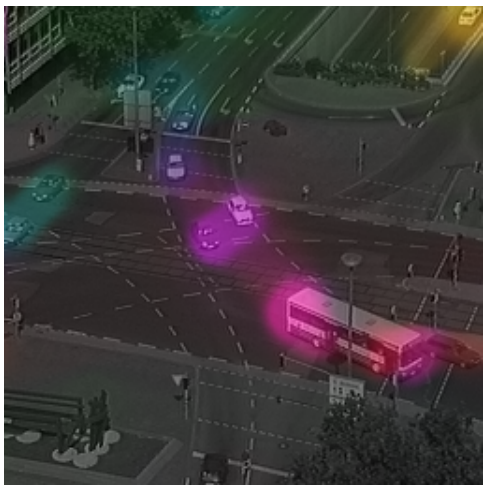


Figure 1.7: Assumptions on optical flows: on a road with cars the motion will almost always be the same at a given locations of the image

1.3.4 Noise and distortions

The sensor may introduce noise which causes unpredictable changes in grey values. This can introduce wrong matchings or missed matchings. The sensor may also introduce some geometrical distortions as showed in figure 1.3.4. Depending on the assumptions made this can prevent the algorithm from validating a matching, for instance if the matching point is too far because of the distortions.

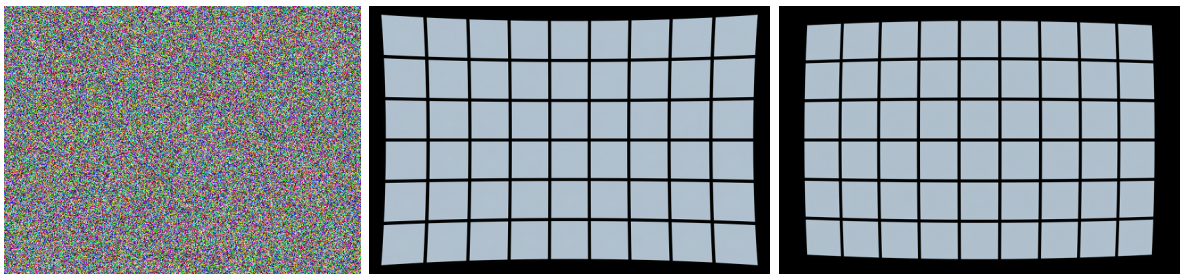


Figure 1.8: Camera noise and distortion [Credits: Nicolas Burtey]

1.3.5 The trade-off between accuracy and efficiency

Two factors have to be constantly taken into consideration in optical flow algorithms: *efficiency* and *accuracy*. There is a trade-off, as often in computer science, between the time of computation and the error made in the solution. But we will see in chapter 2 that, combining advanced equations and robust solving algorithms, some techniques can provide an accurate solution within a reasonable amount of time. But many applications do require to have information on motion in real time, and sometimes a reasonable amount of time is still too much. Even if accuracy is really important, because computing wrong values does not make any sense, efficiency is for these reasons also an important characteristic of an algorithm.

1.4 Diversity of algorithms and models

Many approaches have been tried so far to solve the optical flow problem with different constraints. One algorithm has to be time-efficient to be able to deal with real time situations. It also has to be accurate to compute values of the motion that are close to the real values. But depending on the application

or the type of input data, the models can change. This section describes different categories of optical flow models.

1.4.1 Feature-Based and energy-based models

These two approaches are quite different. In a feature-based approach we take as input a finite set of points. They can be manually selected by an operator, a prior knowledge on starting positions or the output of a point of interest detector such as Harris or SIFT. An energy-based method relies on the minimisation of a cost function – called energy – that expresses constraints on motion. Energy-based models are often dense and continuous.

1.4.2 Nondense and dense models

In a feature-based approach, only the motion of the features given as input has to be computed. As a result the motion will be unknown for any other point. In this case the model is *nondense*. In an energy-based model the energy function is often continuous and can therefore be evaluated at each point in the image. This kind of model is *dense*. Moreover in variational methods with smoothing term (explained in chapter 2), even if the input data is nondense the output can be dense, as some continuity is introduced (see filling-in effect in 2.5).

1.4.3 Continuous and discrete

The two notions of continuity and discreteness should not be confused with dense or nondense. Both continuous and discrete models can be dense i.e. can have values computed for each pixel. Continuous only implies that the model has to be eventually discretised to provide a value for each pixel of the image. A discrete model deals with a finite set of values and not a continuous function.

1.4.4 Synthetic Figure of Classification

To summarise the classification we provide the following schema:

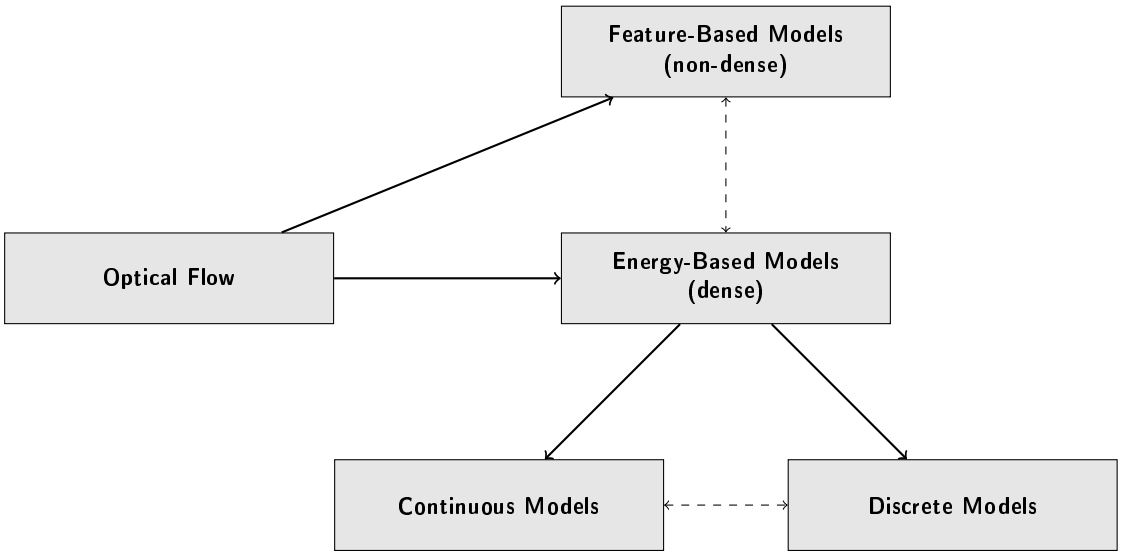


Figure 1.9: Optical flow algorithms classification

Chapter 2

Theoretical aspects

2.1 Error measure and ranking

As many approaches can be used to solve the optical flow, it can be useful to benchmark them with a common set of data to provide a ranking. To be able to compare their algorithms researchers had to define a universal error measure. Several error measures are used but the common reference is the *angular error*.

For a given pair of images we write the ground truth as $\mathbf{u^g} = (u^g, v^g)^\mathsf{T}$ and the estimated flow as $\mathbf{u} = (u, v)^\mathsf{T}$.

2.1.1 Angular error

At the position (i, j) in the image, the angular error is given by the 3D angle between the vectors $(u_{i,j}, v_{i,j}, 1)^\mathsf{T}$ and $(u^g_{i,j}, v^g_{i,j}, 1)^\mathsf{T}$. The third component stands for time. It can be seen as a vector that starts from the feature point in the first image and ends at the location (or evaluated location) of the same feature point in the second image.

It provides a relative measure of performance that avoids divisions by zero at sites where flow equals zero.

$$AE(i,j) = \arccos\left(\frac{1 + u^g_{i,j}u_{i,j} + v^g_{i,j}v_{i,j}}{\sqrt{1 + u^{g^2}_{i,j} + v^{g^2}_{i,j}}\sqrt{1 + u^2_{i,j} + v^2_{i,j}}}\right) \tag{2.1}$$

2.1.2 Endpoint error

The problem with angular error is that the error in regions of smooth nonzero motion are penalised more than errors in region of zero motion. Despite the fact that angular error is a kind of reference, this absolute error measure is probably more appropriate for most applications.

$$EE(i,j) = \sqrt{\left(u^g_{i,j} - u_{i,j}\right)^2 + \left(v^g_{i,j} - v_{i,j}\right)^2} \tag{2.2}$$

2.1.3 Average error

Given an error measure $ERR(i, j)$ being $AE(i, j)$ or $EE(i, j)$, the average error is:

$$AERR(i,j) = \frac{1}{NM}\sum_{i=1}^{N}\sum_{j=1}^{M} ERR(i,j) \tag{2.3}$$

Average error provides an error measure for the flow computed on the whole image which is typically used to rank algorithms.

2.2 Block-based approaches

Let us consider equation 1.1. Due to noise of the sensor and varying conditions of illumination, this equation will never be satisfied over the entire image. We get rid of the time for the moment. To find the best possible solution we can transform it in the following discrete minimisation problem:

$$\mathbf{u}_{i,j} = \underset{u,v}{\operatorname{argmin}} \left[(f_{i,j} - g_{i+u,j+v})^2\right] \tag{2.4}$$

The square is only here to have the same cost for negative and positive deviations.

2.2.1 Block matching

Looking, for each pixel, in the whole image to find a matching is inefficient. The assumption of a small motion reduces the search space to only a small neighbourhood. Then we can write it:

$$\mathbf{u}_{i,j} = \underset{(u,v)\in W_d}{\operatorname{argmin}} \left[(f_{i,j} - g_{i+u,j+v})^2\right] \tag{2.5}$$

where $W_d = [-d,d]^2$ is a window of size $(2d+1)^2$.

2.2.2 Sum of differences

Block matching reduces the complexity to $\mathcal{O}((2d+1)^2 NM)$ but both efficiency and accuracy are still really poor. To improve it a bit, instead of matching one grey value, it is possible to use a block matching strategy. The minimisation problem then becomes:

$$\mathbf{u}_{i,j} = \underset{(u,v)\in W_d}{\operatorname{argmin}} \sum_{(\Delta i,\Delta j)\in B_s} \left[(f_{(i+\Delta i),(j+\Delta j)} - g_{(i+\Delta i)+u,(j+\Delta j)+v})^2\right] \tag{2.6}$$

$B_s = [-s,s]^2$ is a block of size $(2s+1)^2$. Here the accuracy is better but the efficiency slightly worst. The complexity is still linear but with a large constant: $\mathcal{O}((2d+1)^2(2s+1)^2 NM)$. This method is called *sum of squared differences model*. To reduce the influence of outliers, it is possible to replace the square function in this model by an absolute value. This quite similar approach is called *sum of absolute differences*.

$$\mathbf{u}_{i,j} = \underset{(u,v)\in W_d}{\operatorname{argmin}} \sum_{(\Delta i,\Delta j)\in B_s} |f_{(i+\Delta i),(j+\Delta j)} - g_{(i+\Delta i)+u,(j+\Delta j)+v}| \tag{2.7}$$

2.2.3 Normalised cross correlation method

In addition of being computationally demanding, the sum of differences techniques are sensitive to illumination variation. The block matching strategy reduces the probability of wrong matches but it only relies on sum of grey values. So if the illumination changes, the cost function will increase and the model becomes wrong. One idea to improve it is to use the fact that illumination can change the grey values of an image $f_{i,j}$ to $af_{i,j} + b$. Therefore it would be a good thing to compare images centered

around the mean and normalised.

The cross correlation of two images evaluates their similarity. The bigger it is, the closer the images are. The normalised cross correlation achieves it independently of a bias or multiplicative constant. The *normalised cross correlation* technique offers to use a correlation measure in the minimisation problem to get rid of the illumination issues. The minimisation problem then becomes a maximisation problem:

$$\mathbf{u}_{i,j} = \underset{(u,v)\in W_d}{\operatorname{argmax}} \left[\frac{\sum_{(\Delta i,\Delta j)\in B_s} \left(f_{(i+\Delta i),(j+\Delta j)} - \overline{f}_{i,j}\right) \cdot \left(g_{(i+\Delta i)+u,(j+\Delta j)+v} - \overline{g}_{i+u,j+v}\right)}{\sqrt{\sum_{(\Delta i,\Delta j)\in B_s} \left(f_{(i+\Delta i),(j+\Delta j)} - \overline{f}_{i,j}\right)^2} \sqrt{\sum_{(\Delta i,\Delta j)\in B_s} \left(g_{(i+\Delta i)+u,(j+\Delta j)+v} - \overline{g}_{i+u,j+v}\right)^2}} \right] \tag{2.8}$$

2.2.4 Occlusion detection

Occlusion is a crucial problem. When there are occlusions the matching point in the second image may not be found. The best matching grey value (or grey neighbourhood) can be everywhere in the image, sometimes really far and the best match cost can still be very high. The solution for that problem is the *forward-backward check*. The idea is to compute the flow from $I(t)$ to $I(t+1)$ which is called the forward flow $\mathbf{u}_f$. Then it is possible to compute the flow from $I(t+1)$ to $I(t)$ which is called the backward flow $\mathbf{u}_b$.

For each pixel the scalar $\delta i, j = |\mathbf{u_f} + \mathbf{u_b}|$ can be computed. A threshold of this value, typically 0, gives a condition to consider the pixel as occluded in the image $I(t+1)$.

2.2.5 Conclusion

Block based approaches are a good base model as they are simple to comprehend and to implement. They involve only basic operations such as translation or rotations. But the results are dramatically poor in terms of both accuracy and efficiency. For these reasons, they do not fit with our target application. Therefore we will not implement them and will investigate continuous models.

2.3 Continuous models

As discrete models appear to be time consuming and not really accurate, researchers began to focus on continuous model. They are the the object of this section.

2.3.1 Gaussian presmoothing

Continuous methods are mostly based on derivatives of images. As an image is always discrete, there are also discontinuities in its derivatives. One preprocessing which is always applied before using a continuous model is the *Gaussian presmoothing*. The convolution of the initial images with a Gaussian kernel K_σ improves the differentiability. It also reduces noise and therefore reduces final error.

$$f_1 = K_\sigma \star I_1 \tag{2.9}$$
$$f_2 = K_\sigma \star I_2 \tag{2.10}$$

2.3.2 Normal flow

The optical flow problem has two unknowns u, v but only one equation to solve them: the BCCE (equation 1.4). As a result, only the flow component in direction of the gradient – orthogonal to the

edges – can be computed: the normal flow $\mathbf{u}_n$.

$$\mathbf{u}_n = -\frac{f_t}{|\nabla f|}\frac{\nabla f}{|\nabla f|} \tag{2.11}$$

2.3.3 The aperture problem

The aperture problem manifests as an uncertainty on flow computation. Depending on the value of the gradient we can have either an uncertainty on the precise direction of the flow or if the gradient is zero no information on the flow. It actually depends on the rank of the motion tensor (defined after). Let us describe the figure 2.1. If the rank is 2 the flow can be fully computed by two linearly independent feature gradients (left on the figure). If it is 1 one feature gradient is available to compute the flow and then an uncertainty is introduced (middle). If it is 0 no flow can be computed because there is no information available (right).

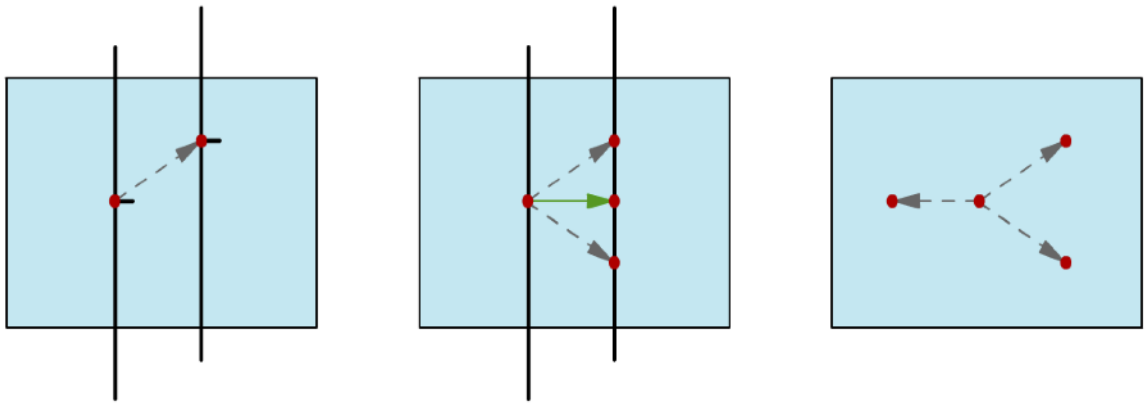


Figure 2.1: The aperture problem: uncertainty on flow computation [Credits: Andrés Bruhn]

The aperture problem find its roots in our brain and in the way we feel and interpret the motion. Look at the next figure and imagine that you are only able to see what is inside the square and do not see the arrows. Now imagine the lines moving in the direction given by the arrows. Because the pattern of lines has a gradient in only one direction you will not be able to make the difference between the left and right squares.

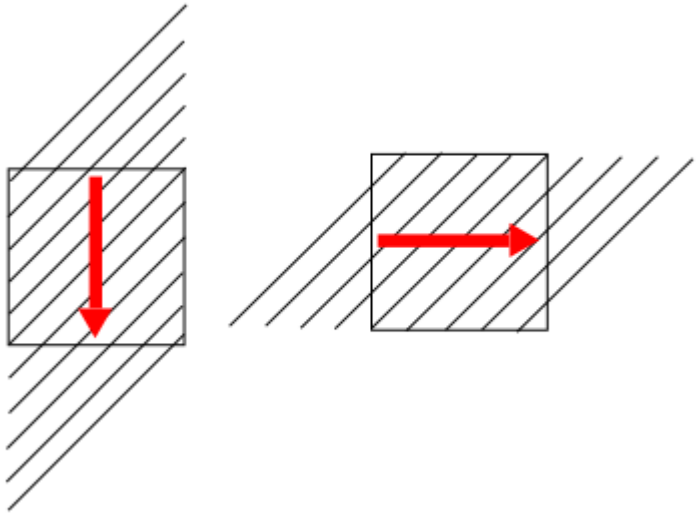


Figure 2.2: The aperture problem: motion interpretation

2.4 *Lucas & Kanade* method

Lucas & Kanade method [18] is a continuous spatial method that assumes the flow to be locally constant and use the neighbourhood information. This method was published in 1981.

2.4.1 Problem definition

$W_d(x_0, y_0)$ being a hard window of size d around (x_0, y_0), the method involves minimisation the following energy functional:

$$E(u(x_0,y_0), v(x_0,y_0)) = \int\limits_{W_d(x_0,y_0)} \left(f_x(x,y,t)u + f_y(x,y,t)v + f_t(x,y,t)\right)^2 dxdy \tag{2.12}$$

The minimisation implies the first order derivatives to be zero:

$$\frac{\partial E}{\partial u} = 2 \int\limits_{W_d(x_0,y_0)} f_x(f_x u + f_y v + f_t)dxdy = 0 \tag{2.13}$$

$$\frac{\partial E}{\partial v} = 2 \int\limits_{W_d(x_0,y_0)} f_y(f_x u + f_y v + f_t)dxdy = 0 \tag{2.14}$$

It can be rewritten in a matrix-vector notation as:

$$\begin{pmatrix} \int\limits_{W_d} f_x^2 dxdy & \int\limits_{W_d} f_x f_y dxdy \\ \int\limits_{W_d} f_x f_y dxdy & \int\limits_{W_d} f_y^2 dxdy \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -\int\limits_{W_d} f_x f_t dxdy \\ -\int\limits_{W_d} f_x f_t dxdy \end{pmatrix} \tag{2.15}$$

Replacing the hard window by a Gaussian kernel gives us the system to solve for each pixel:

$$\begin{pmatrix} K_\rho \star f_x^2 & K_\rho \star f_x f_y \\ K_\rho \star f_x f_y & K_\rho \star f_y^2 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -K_\rho \star f_x f_t \\ -K_\rho \star f_x f_t \end{pmatrix} \tag{2.16}$$

This system of the form $A\mathbf{x} = \mathbf{f}$ is easy to solve as both A and $\mathbf{f}$ do not depend on $(u, v)^\mathsf{T}$. We only need to compute A for each pixel, invert it (direct inversion is possible as A is 2×2), and compute the flow $(u, v)^\mathsf{T} = A^{-1}\mathbf{f}$.

2.4.2 Conclusion

Lucas & Kanade method is the continuous version of the block matching. Although it is better in terms of efficiency, as the minimisation is direct, and the matrices to invert are really small. Use of neighbourhood informations allows it to be more robust under noise and thus more accurate than black based models. Nevertheless having discretised the BCCE it assumes only small displacements and it smoothes the discontinuities in the flow. Moreover there are some parts where the problem cannot be solved, resulting in potentially nondense flows. Finally it deals well with translational motion but not with rotations.

Even if the accuracy is not really high this model is interesting for its simplicity and performances. We therefore made the choice to implement it at first before dealing with more complicated models.

2.5 Variational approaches

Variational methods are currently the kind of continuous model used in all the state of the art techniques to achieve an accurate and dense optical flow computation. All these techniques involve the minimisation of an energy functional of the form:

$$E(u, v) = \int_\Omega D(u, v) + \alpha S(u, v) dxdy \tag{2.17}$$

$D(u, v)$ is called data term and penalises deviations from constancy assumptions made. To overcome the lack of information in some areas, a smoothing term $S(u, v)$ can be introduced ($\alpha > 0$). It penalises deviations from the smoothness of the solution and therefore prolongates the local solution in case of non solvability of the flow. This is known as the filling-in effect. We give an illustration of this phenomena in figure 2.3.

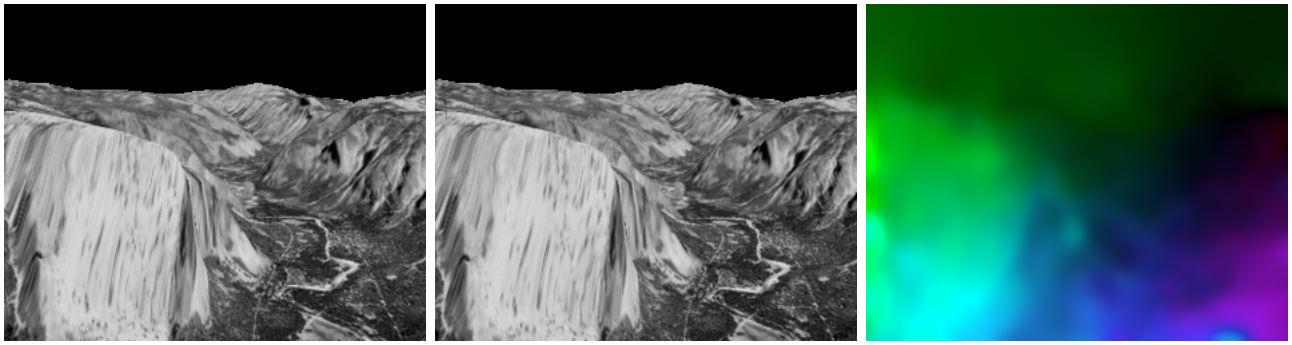


Figure 2.3: Illustration of filling-in effect with Yosemite sequence by *Lynn Quam*

Let $F(x, y, u, v, u_x, u_y, v_x, v_y)$ be a functional such as:

$$E(u, v) = \int_\Omega F(x, y, u, v, u_x, u_y, v_x, v_y) dxdy \tag{2.18}$$

The minimisation requires the derivatives with respect to u and v to be zero:

$$F_u - \frac{\partial F_{u_x}}{\partial x} - \frac{\partial F_{u_y}}{\partial y} = 0 \tag{2.19}$$

$$F_v - \frac{\partial F_{v_x}}{\partial x} - \frac{\partial F_{v_y}}{\partial y} = 0 \tag{2.20}$$

with Neumann boundary conditions where $\mathbf{n}$ denotes the, typically exterior, normal to the boundary $\delta\Omega$:

$$\mathbf{n}^\mathsf{T}\nabla u = 0 \tag{2.21}$$

$$\mathbf{n}^\mathsf{T}\nabla v = 0 \tag{2.22}$$

The coupled system of partial differential equations [2.19, 2.20] is called *Euler-Lagrange equations*. These equations can most of times be easily discretised and yield in a linear (sometimes nonlinear) system of equations to solve. Their solution gives the flow over the entire image that minimises both the deviation from constancy assumptions and smoothness of the solution.

2.6 *Horn & Schunck* method

Horn & Schunck method [17] makes the assumption that the flow is smooth over the whole image. We could think that a smoothness term was simply added to the *Lucas & Kanade* energy function but this method is actually anterior to *Lucas & Kanade* method as it was published in 1980.

2.6.1 Problem definition

For this method the energy functional looks like:

$$E(\mathbf{w}) = \int_\Omega \mathbf{w}^\mathsf{T} J \mathbf{w} + \alpha(|\nabla u|^2 + |\nabla v|^2) dxdy \tag{2.23}$$

The Euler-Lagrange equations then become:

$$J_{11}u + J_{12}v + J_{13} - \alpha\Delta u = 0 \tag{2.24}$$

$$J_{12}u + J_{22}v + J_{23} - \alpha\Delta v = 0 \tag{2.25}$$

Having discretised versions of supposed continuous images we have:

$$u_{i,j} = u(ih_y, jh_x) \tag{2.26}$$

$$v_{i,j} = v(ih_y, jh_x) \tag{2.27}$$

The discretised versions of Euler-Lagrange equations yield as:

$$0 = [J_{11}]_{i,j}u_{i,j} + [J_{12}]_{i,j}v_{i,j} + [J_{13}]_{i,j} - \alpha \sum_{d\in\{x,y\}} \sum_{(k,l)\in\mathcal{N}_d(i,j)} \frac{u_{k,l} - u_{i,j}}{h_d^2} \tag{2.28}$$

$$0 = [J_{12}]_{i,j}u_{i,j} + [J_{22}]_{i,j}v_{i,j} + [J_{23}]_{i,j} - \alpha \sum_{d\in\{x,y\}} \sum_{(k,l)\in\mathcal{N}_d(i,j)} \frac{v_{k,l} - v_{i,j}}{h_d^2} \tag{2.29}$$

Let us define $\mathbf{x}$ and $\mathbf{f}$ with a row-major layout:

$$\mathbf{x} = (\mathbf{flat}u, \mathbf{flat}v)^\mathsf{T} \tag{2.30}$$
$$\mathbf{f} = (\mathbf{flat}J_{13}, \mathbf{flat}J_{23})^\mathsf{T} \tag{2.31}$$

Let us define the following sub-matrices:

$$M_{11} = \mathrm{diag}\,(\mathbf{flat}J_{11}) \tag{2.32}$$
$$M_{12} = \mathrm{diag}\,(\mathbf{flat}J_{12}) \tag{2.33}$$
$$M_{21} = \mathrm{diag}\,(\mathbf{flat}J_{12}) \tag{2.34}$$
$$M_{22} = \mathrm{diag}\,(\mathbf{flat}J_{22}) \tag{2.35}$$

Finally, considering the discretised laplacian operators introduced in appendix A.1 and calling M the matrix we have:

$$M = \left[\left(\begin{array}{c|c} M_{11} & M_{12} \\ \hline M_{21} & M_{22} \end{array}\right) - \frac{\alpha}{h^2}\left(\begin{array}{c|c} L_{N\times M} & 0 \\ \hline 0 & L_{N\times M} \end{array}\right)\right] \tag{2.36}$$

The problem can be expressed as:

$$M\mathbf{x} = \mathbf{f} \tag{2.37}$$

The matrix M is sparse and its nonzero structure is given in figure 2.4. Since M is highly sparse we precompute and store the diagonals as vectors.

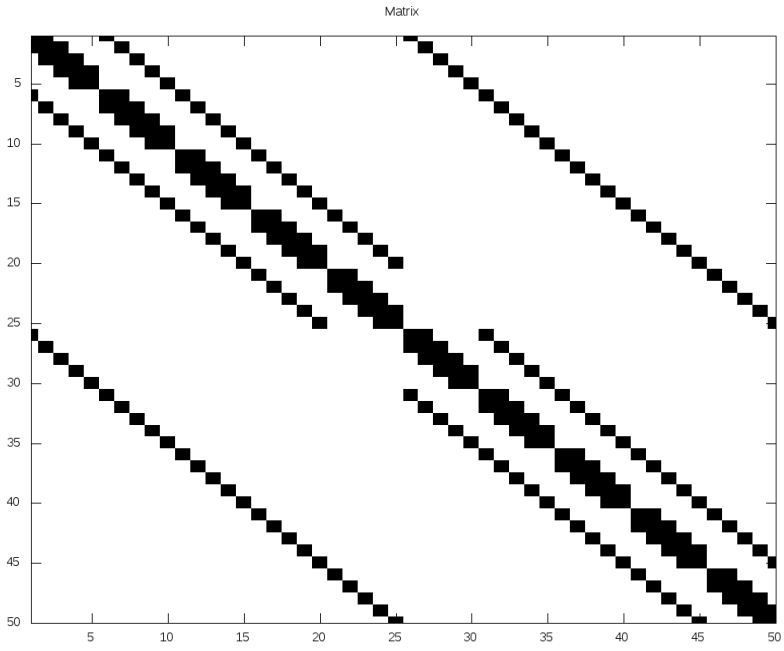


Figure 2.4: Structure of the matrix to invert to solve discretised Euler-Lagrange equations for optical flow problem (5×5 image)

2.6.2 Conclusion

This method has the drawback to use an iterative solver that is lower to converge than the direct computation of *Lucas & Kanade* method. But we will see in 2.8 that some other models of solver can converge a lot faster than traditional ones. Moreover this method can handle rotational motion, yields dense flow fields and can achieve sub-pixel precision. It is definitely harder to implement than previous methods but it is worth an effort.

The quality of results given by this method in terms of accuracy and the ability of iterative solvers to be parallelised led us to implement this method.

2.7 *CLG* method

The *CLG* method, published in 2002 by Bruhn et al.[12][9], can be seen as a combination of *Lucas & Kanade* and *Horn & Schunck* methods. As in *Horn & Schunck* method it introduces a smoothing term to overcome lack of information and as *Lucas & Kanade* it postsmoothes the motion tensor in order to benefit of the local neighbourhood information.

2.7.1 Problem definition

The energy functional for the *CLG* method is given by:

$$E(\mathbf{w}) = \int_{\Omega} \Psi_D(\mathbf{w}^\mathsf{T} J_\rho \mathbf{w}) + \alpha \Psi_S(|\nabla u|^2 + |\nabla v|^2) dxdy \tag{2.38}$$

Note that J_ρ is the postsmoothed motion tensor introduced in *Lucas & Kanade* method.

Firstly if the penaliser $\Psi(s^2)$ is quadratic, which means $\Psi(s^2) = s^2$, we are in the linear case. In the linear case for $\rho = 0$ we have *Horn & Schunck* , and for $\alpha = 0$ we get *Lucas & Kanade* . The Euler-Lagrange equations for the linear case are therefore the same as the *Horn & Schunck* 's ones 2.25.

Now if we use a non-quadratic penaliser such as $\Psi(s^2) = \sqrt{s^2 + \epsilon^2}$[9][11] for the data and/or smoothness term we introduce nonlinearities. This particular penaliser introduces a small constant ϵ that avoids the data and smoothing term from being zero in zones where there would be no information.

In that case the Euler-Lagrange equation become:

$$\begin{aligned} \Psi'_D(J_{11}u + J_{12}v + J_{13}) - \alpha \mathrm{div}(\Psi'_S \nabla u) &= 0 & (2.39)\\ \Psi'_D(J_{12}u + J_{22}v + J_{23}) - \alpha \mathrm{div}(\Psi'_S \nabla v) &= 0 & (2.40) \end{aligned}$$

The same discretisation strategy as in *Horn & Schunck* method gives us:

$$\begin{aligned} 0 &= [\Psi'_D]_{i,j}([J_{11}]_{i,j}u_{i,j} + [J_{12}]_{i,j}v_{i,j} + [J_{13}]_{i,j}) & (2.41)\\ &- \alpha \sum_{d\in\{x,y\}} \sum_{(k,l)\in\mathcal{N}_d(i,j)} \frac{[\Psi'_S]_{k,l} + [\Psi'_S]_{i,j}}{2} \frac{u_{k,l} - u_{i,j}}{h_d^2} \\ 0 &= [\Psi'_D]_{i,j}([J_{12}]_{i,j}u_{i,j} + [J_{22}]_{i,j}v_{i,j} + [J_{23}]_{i,j}) & (2.42)\\ &- \alpha \sum_{d\in\{x,y\}} \sum_{(k,l)\in\mathcal{N}_d(i,j)} \frac{[\Psi'_S]_{k,l} + [\Psi'_S]_{i,j}}{2} \frac{v_{k,l} - v_{i,j}}{h_d^2} \end{aligned}$$

For the particular penaliser $\Psi(s^2) = \sqrt{s^2 + \epsilon^2}$ the derivative is such as:

$$\Psi'(s^2) = \frac{1}{2\sqrt{s^2 + \epsilon^2}} \tag{2.43}$$

Thus the discretised versions of the data and smoothing penalisers are computed as:

$$[\Psi'_D]_{i,j} = \frac{1}{2\sqrt{(u_{i,j}, v_{i,j}, 1)[J]_{i,j}(u_{i,j}, v_{i,j}, 1)^\mathsf{T} + \epsilon_D^2}} \tag{2.44}$$

$$[\Psi'_S]_{i,j} = \frac{1}{2\sqrt{|\nabla_2 u_{i,j}|^2 + |\nabla_2 v_{i,j}|^2 + \epsilon_S^2}} \tag{2.45}$$

2.7.2 Conclusion

The elegance of this method is definitely that it combines advantages of both *Lucas & Kanade* and *Horn & Schunck* methods. It is robust against noise and computes dense flow field. All kinds of constancy assumptions can easily be introduced in the energy function and this method is thus easily tunable. Finally it is not computationally more expensive than the other methods and the results are slightly better.

This method will be the target method of our project, fitting perfectly the requirements of our target application.

2.8 Solvers

According to *Bruhn et al.*[11][6] the faster iterative solver to converge for optical flow is the point-wise Gauss-Seidel using successive over-relaxation. This solver is fine on a CPU, but for a GPU solver the amount of communication between blocks (see 3.2.2) has to be minimised. Therefore the Jacobi solver appears to be the best to be implemented on a GPU.

2.8.1 Jacobi solver

We want to solve iteratively the following linear system of equations:

$$M\mathbf{x} = \mathbf{f} \tag{2.46}$$

The matrix M can be decomposed in a diagonal matrix D and the rest R:

$$(D + R)\mathbf{x} = \mathbf{f} \tag{2.47}$$

$$D\mathbf{x} = D^{-1}(\mathbf{f} - R\mathbf{x}) \tag{2.48}$$

The Jacobi solver can then be formulated as:

$$\mathbf{x}^{(k+1)} = D^{-1}\left(\mathbf{f} - R\mathbf{x}^{(k)}\right) \tag{2.49}$$

or:

$$\mathbf{x}_i^{(k+1)} = \frac{1}{a_{ii}}\left(\mathbf{f}_i - \sum_{j \neq i} a_{ij}\mathbf{x}_j^{(k)}\right), \quad i = 1, \ldots, n \tag{2.50}$$

We then come up with the following algorithm:

Algorithm 2.8.1: JACOBI$(M, \mathbf{f}, \mathbf{x}_0, n)$

$\mathbf{x} \leftarrow \mathbf{x}_0$
for $it \leftarrow 1$ **to** n
do
  for $i \leftarrow 1$ **to** N
  do
    $sum \leftarrow 0$
    for $j \leftarrow 1$ **to** M
    do
      if $j = I$
        then continue
      $sum \leftarrow sum + M_{i,j} * \mathbf{x}_j$
    $\mathbf{x} \leftarrow (1/M_{i,i})\,(\mathbf{f}_i - sum)$

2.8.2 Jacobi solver with lagged nonlinearities

Jacobi solver with lagged nonlinearities[9][16][14][13], or nonlinear Jacobi solver as we will call it, is based on the technique of *frozen coefficients*[15]. It solves nonlinear problems by doing a new linearisation at each iteration point.

We want to solve iteratively the following nonlinear system of equations:

$$N(\mathbf{x}) = \mathbf{f} \tag{2.51}$$

Here $N(\mathbf{x})$ is a nonlinear operator that can be decomposed into:

$$N(\mathbf{x}) = A\,(\mathbf{x})\,\mathbf{x} + \mathbf{b}\,(\mathbf{x}) \tag{2.52}$$

$A(\mathbf{x})$ and $\mathbf{b}(\mathbf{x})$ are nonlinear operators. As for the Jacobi algorithm we can find out an iterative form for the computation of $\mathbf{x}$:

$$x^{k+1} = \left(A\left(\mathbf{x}^{(k)}\right)\right)^{-1}(\mathbf{f} - \mathbf{b}(\mathbf{x}^k)) \tag{2.53}$$

But as the operators are nonlinear they depend themselves on $\mathbf{x}$. The method of lagged diffusivity consists in the evaluation of the nonlinear operators at step k with the value of $\mathbf{x}$ at $k-1$. Thus it involves the resolution of a linear system for each iteration step:

$$A\left(\mathbf{x}^{(k)}\right)\mathbf{x}^{(k+1)} = \left(\mathbf{f} - \mathbf{b}\left(\mathbf{x}^{(k)}\right)\right) \tag{2.54}$$

Algorithm 2.8.2: JACOBINL$(M, \mathbf{f}, \mathbf{x}_0, n)$

$\mathbf{x} \leftarrow \mathbf{x}_0$
for $it \leftarrow 1$ **to** n
do $\begin{cases} M^{NL} \leftarrow \text{UPDATE\_M}(M, \mathbf{x}) \\ \mathbf{f}^{NL} \leftarrow \text{UPDATE\_F}(\mathbf{f}, \mathbf{x}) \\ \\ \textbf{for } i \leftarrow 1 \textbf{ to } N \\ \quad \textbf{do} \begin{cases} sum \leftarrow 0 \\ \\ \textbf{for } j \leftarrow 1 \textbf{ to } M \\ \quad \textbf{do} \begin{cases} \textbf{if } j = i \\ \quad \textbf{then continue} \\ sum \leftarrow sum + M_{i,j}^{NL} * \mathbf{x}_j^{NL} \end{cases} \\ \\ \mathbf{x} \leftarrow (1/M_{i,i}^{NL}) \left(\mathbf{f}_i^{NL} - sum\right) \end{cases} \end{cases}$

2.8.3 Multigrid solver

We want to solve iteratively the following nonlinear system of equations:

$$M\mathbf{x} = \mathbf{f} \tag{2.55}$$

Traditional solvers as Jacobi or Gauss-Seidel are fast to correct high-frequent errors (during the first iterations) but slow to correct low-frequent errors (needs a lot of iterations). Multigrid solvers[4] were then invented to compensate slowness of traditional solvers in order to get accurate solutions within a small amount of time. The key concept of multigrid solvers is that, for a given system of equations, low-frequent errors are high-frequent errors of a coarser version of the same system. We implement them as adviced by Andrés Bruhn[9].

Let us define restriction operation R that reduces a fine solution to a coarse one by averaging. We also introduce P, the dual operation, that interpolates linearly from a coarse solution to a finer one. In our application of multigrid solvers our solutions are images and we will always have a factor 2×2 between a fine and a coarse grid. Thus, h being the step of the finer grid, our operators will take the particular values:

$$R = R^{h \to 2h} \tag{2.56}$$
$$P = P^{2h \to h} \tag{2.57}$$

As our matrices and right hand depend on J, we need to compute a coarser version J^{2h} of the motion tensor J^h applying $R^{h \to 2h}$ operator component-wisely to it:

$$J^{2h} = R^{h \to 2h} J^h = \begin{pmatrix} R^{h \to 2h}\mathbf{J}_{11}^h & R^{h \to 2h}\mathbf{J}_{12}^h & R^{h \to 2h}\mathbf{J}_{13}^h \\ R^{h \to 2h}\mathbf{J}_{21}^h & R^{h \to 2h}\mathbf{J}_{22}^h & R^{h \to 2h}\mathbf{J}_{23}^h \\ R^{h \to 2h}\mathbf{J}_{31}^h & R^{h \to 2h}\mathbf{J}_{32}^h & R^{h \to 2h}\mathbf{J}_{33}^h \end{pmatrix} \tag{2.58}$$

A coarser version $\mathbf{f}^{2h}$ of the right hand $\mathbf{f}^h$ can then be computed and, the laplacian operator for the coarser grid being easy to recompute, the coarse matrix M^{2h} can be easily computed too.

Firstly, in order to use at best traditional solvers, we apply n_1 presmoothing iterations of one to the initial system in order to remove high-frequent errors. The solution $\bar{\mathbf{x}}^h$ computed is obviously not an exact solution; $\mathbf{x}^h$ being the exact solution and $\mathbf{e}^h$ being the remaining error we have:

$$\bar{\mathbf{x}}^h = \mathbf{x}^h + \mathbf{e}^h \tag{2.59}$$

As one wants to compute the most accurate solution, we are interested in finding $\mathbf{e}^h$. Let $\mathbf{r}^h$ be the residual so that:

$$\mathbf{r}^h = \mathbf{f}^h - M^h \bar{\mathbf{x}}^h \tag{2.60}$$

Then the error can be found solving the system:

$$M^h \mathbf{e}^h = \mathbf{r}^h \tag{2.61}$$

The first iterations of the traditional solver having already corrected high-frequent error in $\bar{\mathbf{x}}$, we are more interested in correcting low-frequent errors. This can be done in a few iterations by solving this system on the coarser grid:

$$M^{2h} \mathbf{x}^{2h} = R^{h \to 2h} \mathbf{r}^h \tag{2.62}$$

As the error is expected to be small, $\mathbf{x}^{2h} = \mathbf{0}$ is used for the initial guess. We then prolongate the result $\bar{\mathbf{x}}^{2h}$ to get the error:

$$\mathbf{e}^h = P^{2h \to h} \bar{\mathbf{x}}^{2h} \tag{2.63}$$

Then $\bar{\mathbf{x}}$ can be corrected to find the exact solution:

$$\mathbf{x}^h = \bar{\mathbf{x}}^h + \mathbf{e}^h \tag{2.64}$$

Finally n_2 postsmoothing iterations of a traditional solver can be applied to remove the high frequent errors introduced by prolongation operator. We come up with the following algorithm:

Algorithm 2.8.3: SOLVE2GRIDS$(M^h, \mathbf{f}^h, \mathbf{x}_0^h, M^{2h}, \mathbf{f}^{2h})$

global n_0, n_1, n_2

$\bar{\mathbf{x}}^h \leftarrow$ JACOBI$(M^h, \mathbf{f}^h, \mathbf{x}_0^h, n_1)$

$\mathbf{r}^h \leftarrow \mathbf{f}^h - M^h \bar{\mathbf{x}}^h$
$\mathbf{f}^{2h} \leftarrow R^{h \to 2h} \mathbf{r}^h$
$\mathbf{e}^{2h} \leftarrow$ JACOBI$(M^{2h}, \mathbf{f}^{2h}, \mathbf{0}, n_0)$

$\bar{\mathbf{x}}^h \leftarrow \bar{\mathbf{x}}^h + P^{2h \to h} \mathbf{e}^{2h}$
$\mathbf{x}^h \leftarrow$ JACOBI$(M^h, \mathbf{f}^h, \bar{\mathbf{x}}^h, n_2)$

return $(\mathbf{x}^h)$

The accuracy of the solution obtained after postsmoothing closely depends on how the third step is solved. If we compute $\mathbf{x}^{2h}$ accurately, the solution $\mathbf{x}^h$ obtained after correction and postsmoothing can be really accurate too. The ideal case is when $\mathbf{x}^{2h}$ is computed exactly, inverting the coarse system. Then all the error of the solution comes only from the prolongation operation and is well corrected by the postsmoohting iterations. Unfortunately if the system to solve is huge (as ours will be) the coarse system will still be too big to be invert directly. A Jacobi solver can be used to solve the coarse system but, as it is too big, the resolution will converge slowly due to the same problems we had for the fine grid.

Bidirectional approach

One elegant solution to that problem, known as the *bidirectional approach*, is to use a multigrid solver at the coarse level which implies the use of a new coarse grid. This can be done recursively until the grid is small enough to be accurately solved. This is called a V-cycle and is illustrated in figure 2.5.

Algorithm 2.8.4: VCycle$(d, \mathbf{f}^h, {\mathbf{x_0}}^h)$

---

comment: should be called with $d := 0$

global $D_{MAX}, N := D_{MAX} + 1$
global $M[N], n_0, n_1, n_2$

if $d = D_{MAX}$
  then $\begin{cases} \bar{\mathbf{x}}^h \leftarrow \text{Jacobi}(M[d], \mathbf{f}^h, \mathbf{x}_0^h, n_0) \\ \textbf{return } (\bar{\mathbf{x}}^h) \end{cases}$

$\bar{\mathbf{x}}^h \leftarrow \text{Jacobi}(M[d], \mathbf{f}^h, \mathbf{x}_0^h, n_1)$

$\mathbf{r}^h \leftarrow \mathbf{f}^h - M[d]\bar{\mathbf{x}}^h$
$\mathbf{f}^{2h} \leftarrow R^{h \to 2h}\mathbf{r}^h$
$\mathbf{e}^{2h} \leftarrow \text{VCycle}(d+1, \mathbf{f}^{2h}, \mathbf{0})$
$\bar{\mathbf{x}}^h \leftarrow \bar{\mathbf{x}}^h + P^{2h \to h}\mathbf{e}^{2h}$

$\mathbf{x}^h \leftarrow \text{Jacobi}(M[d], \mathbf{f}^h, \bar{\mathbf{x}}^h, n_2)$

return $(\mathbf{x}^h)$

Algorithm 2.8.5: $\text{WCYCLE}(d, \mathbf{f}^h, \mathbf{x_0}^h)$

comment: should be called with $d := 0$

global $D_{MAX}, N := D_{MAX} + 1$
global $M[N], n_0, n_1, n_2$

if $d = D_{MAX}$
then $\begin{cases} \bar{\mathbf{x}}^h \leftarrow \text{JACOBI}(M[d], \mathbf{f}^h, \mathbf{x}_0^h, n_0) \\ \textbf{return } (\bar{\mathbf{x}}^h) \end{cases}$

$\bar{\mathbf{x}}^h \leftarrow \text{JACOBI}(M[d], \mathbf{f}^h, \mathbf{x}_0^h, n_1)$

$\mathbf{r}^h \leftarrow \mathbf{f}^h - M[d]\bar{\mathbf{x}}^h$
$\mathbf{f}^{2h} \leftarrow R^{h \to 2h}\mathbf{r}^h$
$\mathbf{e}^{2h} \leftarrow \text{WCYCLE}(d+1, \mathbf{f}^{2h}, \mathbf{0})$
$\bar{\mathbf{x}}^h \leftarrow \bar{\mathbf{x}}^h + P^{2h \to h}\mathbf{e}^{2h}$

$\mathbf{r}^h \leftarrow \mathbf{f}^h - M[d]\bar{\mathbf{x}}^h$
$\mathbf{f}^{2h} \leftarrow R^{h \to 2h}\mathbf{r}^h$
$\mathbf{e}^{2h} \leftarrow \text{WCYCLE}(d+1, \mathbf{f}^{2h}, \bar{\mathbf{x}}^h)$
$\bar{\mathbf{x}}^h \leftarrow \bar{\mathbf{x}}^h + P^{2h \to h}\mathbf{e}^{2h}$

$\mathbf{x}^h \leftarrow \text{JACOBI}(M[d], \mathbf{f}^h, \bar{\mathbf{x}}^h, n_2)$

return $(\mathbf{x}^h)$

The quality of this approach resides in the fact that to get best accuracy only the smaller grid needs to be computed very precisely. As this grid is ridiculously small, it is easy either to invert it or to iteratively solve it with a really small error in a reasonable amount of time. By correcting the error of the error of ...of the solution we end up with a really accurate one only doing a few pre and postsmoothing iterations each time.

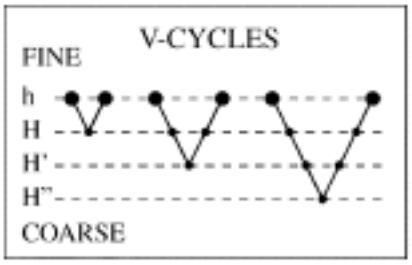


Figure 2.5: V-Cycle [Credits: Andrés Bruhn]

To get more accuracy one may want to do two error corrections at each level. This is called a W-cycle and is illustrated in figure 2.6.

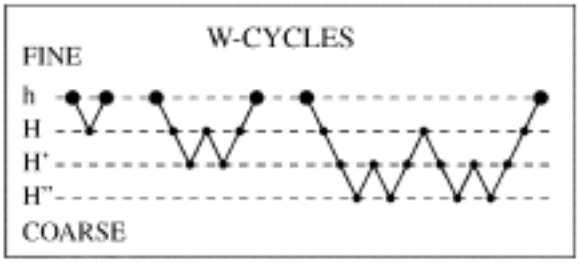


Figure 2.6: W-Cycle [Credits: Andrés Bruhn]

Cascadic approach

A different approach is the *cascadic approach*, depicted in figure 2.7. The quality of initial guess impacts strongly on the speed of convergence of an iterative solver. If the initial guess is good enough, even a traditional iterative solver will converge quickly. Coarse systems being faster to solve accurately than finer ones, we can compute a solution at the coarse level, prolongate it, and use it as initial guess for the next finer grid. This can be done several times accelerating the whole process and thus increasing the accuracy of the solution for a given time of computation.

Algorithm 2.8.6: CASCADIC$(d, \mathbf{x}_0^h)$

---

comment: should be called with $d := D_{MAX}$

global $D_{MAX}, N := D_{MAX} + 1$
global $M[N], \mathbf{f}[N], n$

$\bar{\mathbf{x}}^h \leftarrow$ JACOBI$(M[d], \mathbf{f}[d], \mathbf{x}_0^h, n)$

if $d = 0$
  then return $(\bar{\mathbf{x}}^h)$

$\bar{\mathbf{x}}_0^{2h} = P^{2h \to h} \bar{\mathbf{x}}^h$
return (CASCADIC$(d-1, \mathbf{x}_0^{2h})$)

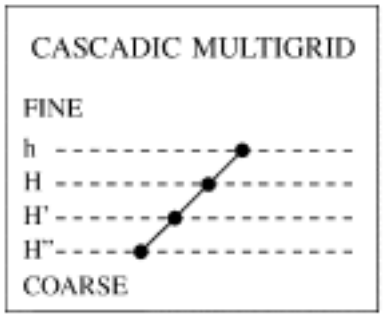


Figure 2.7: Illustration of cascadic approach for multigrid solvers [Credits: Andrés Bruhn]

Full multigrid approach

Finally the most advanced approach is the *full multigrid approach* that combines both *bidirectional* and *cascadic* approaches as illustrated in figure 2.8. It can be seen as a cascadic scheme in which at each level a V-cycle (or W-cycle) is performed to correct the error made. While for a basic iterative approach a lot of iterations are required, for full multigrid only one iteration of the entire scheme is used. Note that it would not make any sense to do several iterations as one solution computed would have to be restricted from the finest level of an iterations to the coarsest level of the next one, thus loosing all computed information. The values of n_1 and n_2 can be really small (typically 1 or 2) as the high-frequent errors are quickly removed during first iterations.

Algorithm 2.8.7: $\mathrm{FMG}(d, \mathbf{x}_0^h)$

---

comment: should be called with $d := D_{MAX}$

global $D_{MAX}, N := D_{MAX} + 1$
global $M[N], \mathbf{f}[N], n_0, n_1, n_2, cycles$

for $i \leftarrow 1$ **to** $cycles$
  do $\bar{\mathbf{x}}^h \leftarrow \mathrm{VCycle}(d, \mathbf{f}[d], \mathbf{x}_0^h)$

if $d = 0$
  then return $(\bar{\mathbf{x}}^h)$

$\bar{\mathbf{x}}_0^{2h} = P^{2h \to h} \bar{\mathbf{x}}^h$

return $(\mathrm{FMG}(d-1, \mathbf{x}_0^{2h}))$

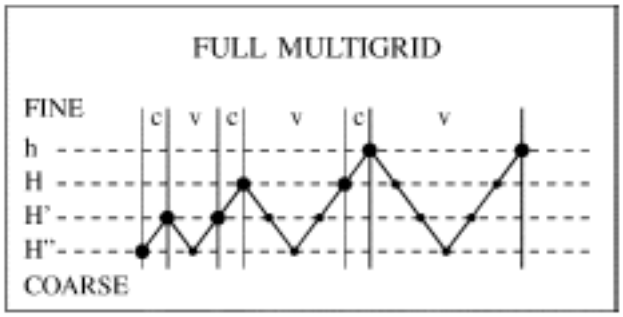


Figure 2.8: Illustration of full multigrid solver approach [Credits: Andrés Bruhn]

2.8.4 Nonlinear multigrid solver

We want to solve iteratively the following nonlinear system of equations:

$$N(\mathbf{x}) = \mathbf{f} \tag{2.65}$$

The nonlinear multigrid solver is a multigrid scheme based on the use of a traditional nonlinear solver such as Jacobi with lagged nonlinearities. The nonlinear approach differs from the linear one only by the fact that operators need to be constantly updated with the new solution computed.

After n_1 presmoothing iterations we come up with the solution $\bar{\mathbf{x}}$. We compute the residual:

$$\mathbf{r}^h = \mathbf{f}^h - N(\bar{\mathbf{x}}^h) \tag{2.66}$$

We also restrict $\bar{\mathbf{x}}$ to the coarse level and use it to evaluate the nonlinear operator $N^{2h}(R^{h\to 2h}\bar{\mathbf{x}})$.

We then solve the system with the initial guess $\mathbf{x}^{2h} = R^{h\to 2h}\bar{\mathbf{x}}^h$:

$$N^{2h}(\mathbf{x}^{2h}) = R^{h\to 2h}\mathbf{r}^h + N^{2h}(R^{h\to 2h}\bar{\mathbf{x}}^h) \tag{2.67}$$

Then the error at grid level h is computed using the result $\bar{\mathbf{x}}^{2h}$:

$$e^h = P^{2h\to h}(\bar{\mathbf{x}}^{2h} - R^{h\to 2h}\bar{\mathbf{x}}^h) \tag{2.68}$$

Finally n_2 iterations of postsmoothing are performed to get the final solution. If we use a nonlinear Jacobi solver seen previously we come up with the following algorithms for the V-Cycles and the recursive FMGNL function:

Algorithm 2.8.8: VCycleNL$(d, \mathbf{f}^h, {\mathbf{x_0}}^h)$

comment: should be called with $d := 0$

global $D_{MAX}, S := D_{MAX} + 1$
global $N[S], n_0, n_1, n_2$

if $d = D_{MAX}$
  then $\begin{cases} \bar{\mathbf{x}}^h \leftarrow \text{JacobiNL}(N[d], \mathbf{f}^h, \mathbf{x}_0^h, n_0) \\ \textbf{return } (\bar{\mathbf{x}}^h) \end{cases}$

$\bar{\mathbf{x}}^h \leftarrow \text{JacobiNL}(N[d], \mathbf{f}^h, \mathbf{x}_0^h, n_1)$

$\mathbf{r}^h \leftarrow \mathbf{f}^h - N[d]\bar{\mathbf{x}}^h$
$\mathbf{f}^{2h} \leftarrow R^{h\to 2h}\mathbf{r}^h + N[d+1] \cdot R^{h\to 2h}\bar{\mathbf{x}}^h$
$\mathbf{e}^{2h} \leftarrow \text{VCycleNL}(d+1, \mathbf{f}^{2h}, R^{h\to 2h}\bar{\mathbf{x}}^h)$
$\mathbf{e}^{2h} \leftarrow \mathbf{e}^{2h} - R^{h\to 2h}\bar{\mathbf{x}}^h$
$\bar{\mathbf{x}}^h \leftarrow \bar{\mathbf{x}}^h + P^{2h\to h}\mathbf{e}^{2h}$

$\mathbf{x}^h \leftarrow \text{JacobiNL}(N[d], \mathbf{f}^h, \bar{\mathbf{x}}^h, n_2)$

return $(\mathbf{x}^h)$

Algorithm 2.8.9: FMGNL$(d, \mathbf{x}_0^h)$

comment: should be called with $d := D_{MAX}$

global $D_{MAX}, S := D_{MAX} + 1$
global $N[S], \mathbf{f}[S], n_0, n_1, n_2, cycles$

for $i \leftarrow 1$ **to** $cycles$
  do $\bar{\mathbf{x}}^h \leftarrow \text{VCycleNL}(d, \mathbf{f}[d], \mathbf{x}_0^h)$

if $d = 0$
  then return $(\bar{\mathbf{x}}^h)$

$\bar{\mathbf{x}}_0^{2h} = P^{2h\to h}\bar{\mathbf{x}}^h$

return $(\text{FMGNL}(d-1, \mathbf{x}_0^{2h}))$

Chapter 3

CUDA Parallel programming

3.1 GPUs and CPUs: two different species

Current CPUs have 2, 4, 8, maybe more than 16 cores. But it is nothing compared to the hundred of cores available on a GPU. Graphics Processing Units (GPU) were initially designed to compute graphics calculus. Computer graphics use a lot of linear algebra which is known to be easily parallelisable. As a consequence, GPUs evolved to many-core architectures dedicated to very specific and repetitive tasks. In the meantime traditional CPUs took also advantage of parallelism but evolved to multi-core architectures (less cores than in many-core). The figure 3.1 shows the evolution of the computational power (in FLOPS: floating point operations per second) of CPUs and GPUs over the past years. We can see, in the figure, that the computational power of the GPUs has become much more higher than the best of the CPUs. It can easily be discussed as the unit of measure used to plot this figure (source *nvidia*) clearly advantages the GPUs. But with standard benchmark, i.e. complicated and not repetitive tasks, we could see things totally differently.

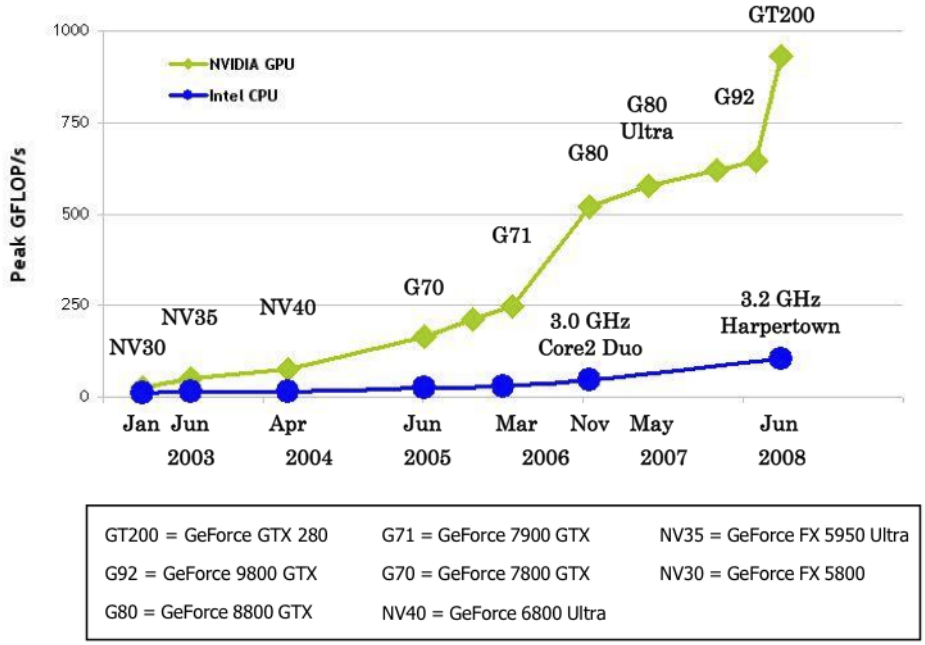


Figure 3.1: Computational power (FLOPS): CPU vs. GPU [Credits: *nvidia*]

What is the price for that huge amount of power? As figure 3.2 depicts, a GPU devotes the major

part of its surface to transistors (ALU being Arithmetic Logic Unit) i.e. to computation. A CPU devotes the major part of its surface to fast cache memory, DRAM and control. But the processor surface devoted to memory is smaller for GPUs than for CPUs, in particular the cache memory is really small on a GPU.

Obviously CPUs and GPUs are not designed to do the same tasks... A GPU is conceived to address problems that have a high arithmetic intensity, which is the ratio of arithmetic operations to memory operation. Moreover, with that number of cores the pipeline strategy cannot be as efficient as in a multi-core processor. This means that operations addressed by a GPU have to be very simple and highly repetitive. CPUs have to be fast in a lot of different situations using always more complex pipeline models that a GPU would not be able to embed.

To summarise, we could say that *a good GPU programmer should be able to take the best of both the GPU and the CPU and balance with elegance the load of computation for a given task on the best of them taking in account memory transfers.*

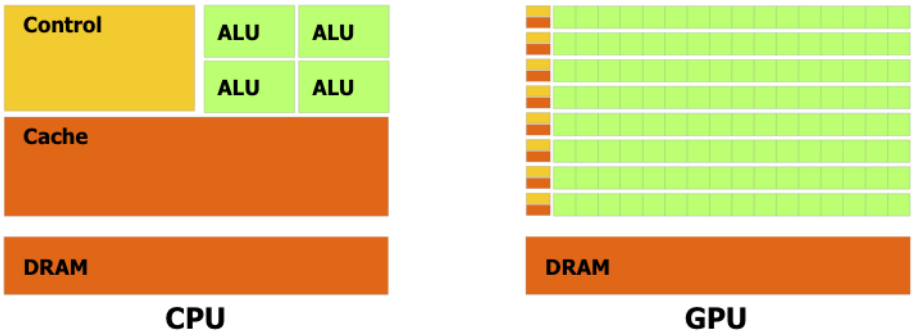


Figure 3.2: Processor Surface used for transistors: CPU / GPU [Credits: *nvidia*]

All computers over the world have a graphic card, and most of them provide graphic acceleration. GPU vendors, such as *nvidia* or *ATI*, then thought "we could use this huge amount of computation capacity for something else than graphics processing". They felt this trend and decided to open their many-core architectures releasing application programming interfaces for people to interact directly with their GPUs. Use of GPU for general purpose computing or GP-GPU is currently investigated in many fields of science and industry.

3.2 *CUDA* API

CUDA or *Compute Unified Device Architecture* is a C/C++-based API for general purpose parallel computing on GPU released by *nvidia*. It provides flexible control mechanisms to take the most advantage of the parallel architecture. As figure 3.3 shows us, complex applications can be built on top of *CUDA* which abstracts low level hardware mechanisms to provide a common interface for all CUDA-capable GPUs. But this figure also mentions alternatives to *CUDA* such as *OpenCL* originally created by *Apple* now developed by *Khronos Group*, *DirectCompute* developed by *Microsoft*, or – not mentioned in the figure – *ATI Stream* developed by *ATI*.

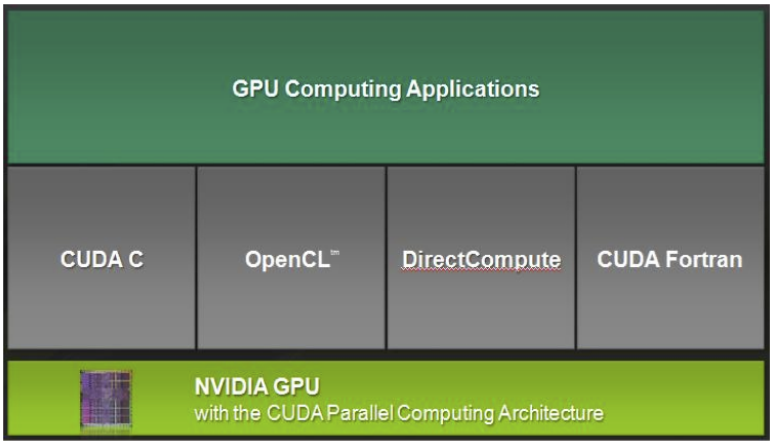


Figure 3.3: *CUDA* : abstraction layers [Credits: *nvidia*]

3.2.1 The choice of using *CUDA*

The *ATI Stream* seems to be a great technology, unfortunately we do not have the specific hardware for using it in this project. *DirectCompute* implies the use of *DirectX* and *Microsoft Windows* which are owned technology that we cannot use in our case. *OpenCL* seems a good alternative to *CUDA* as it is meant to work on either *ATI* or *nvidia* GPUs. But, at the time we are writing this report, it is told to be a bit too high level and hardly optimisable which contradicts with our application. Therefore *CUDA* seems to be the best solution to choose for our problem. The *CUDA* driver comes with a SDK full of examples and a really good documentation. The technology seems to be really fertile as the community around *CUDA* is really active on internet.

3.2.2 Architecture

A parallel algorithm does, at least for some parts, some similar tasks at the same time – rather obvious so far. Those similar tasks are performed by logical units of computation called *threads*. These threads are laid out in two dimensions, as most of the algorithms can take a matrix form, grouped in bigger units called *blocks*. Blocks are also laid out in two dimensions in the biggest unit called *grid*. Figure 3.4 depicts this hierarchy.

The hierarchy makes sense if we introduce memory. The GPU device is connected to a DRAM global memory, in the form of a heap, that can be accessed by any grid. Each block can statically allocate shared memory that all the threads it contains can access. Finally each thread can use its own local memory and access the two aforementioned memory types – not with the same bandwidth though. Figure 3.5 illustrates this memory organisation.

3.2.3 Kernels demystified

A kernel – read parallel function – is a set of instructions that is meant to be executed by all the threads of a grid. It also defines at compile time the amount of shared memory allocated for each block and the local variables instantiated for each thread. The model this architecture is *SIMT* (Single Instruction Multiple-Thread) which is close to *SIMD* (Single Instruction Multiple-Data) with a few differences. We cannot explain it better than it is done in the *CUDA* programming guide:

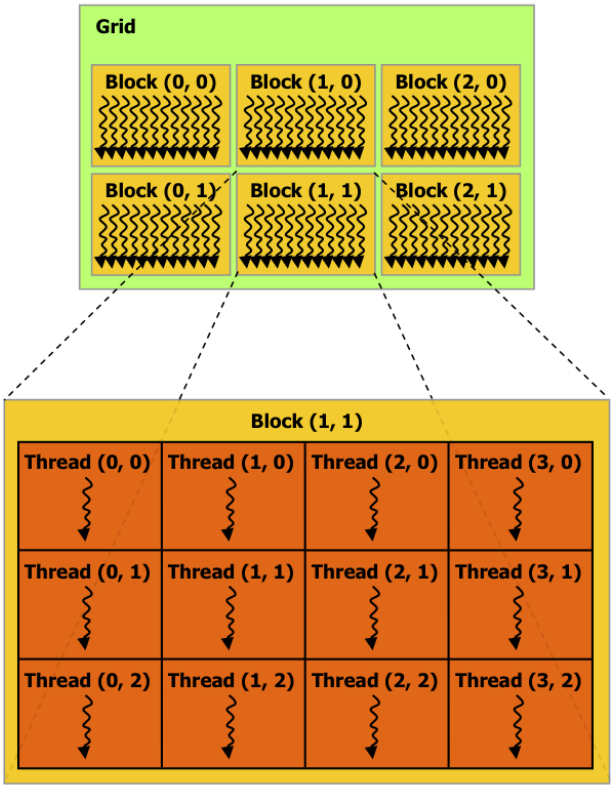


Figure 3.4: *CUDA* : Grid, blocks and threads [Credits: *nvidia*]

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term warp originates from weaving, the first parallel thread technology. A half-warp is either the first or second half of a warp. A quarter-warp is either the first, second, third, or fourth quarter of a warp.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organisations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organisations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behaviour of a single thread.

[Credits: *nvidia*]

Kernels calls can then be mixed with standard serial code, see figure 3.6. By nature a kernel call – syntax `kernel<<<gridSize, blockSize>>>(args)` – is asynchronous but the *CUDA* core directive `cudaThreadSynchronize()` allows a programmer to synchronise its host code with the device when needed.

A kernel – for instance a vector copy kernel – is declared using the following constructions:

```
1 __global__
2 void add_kernel(float* v1, float* v2, int size)
```

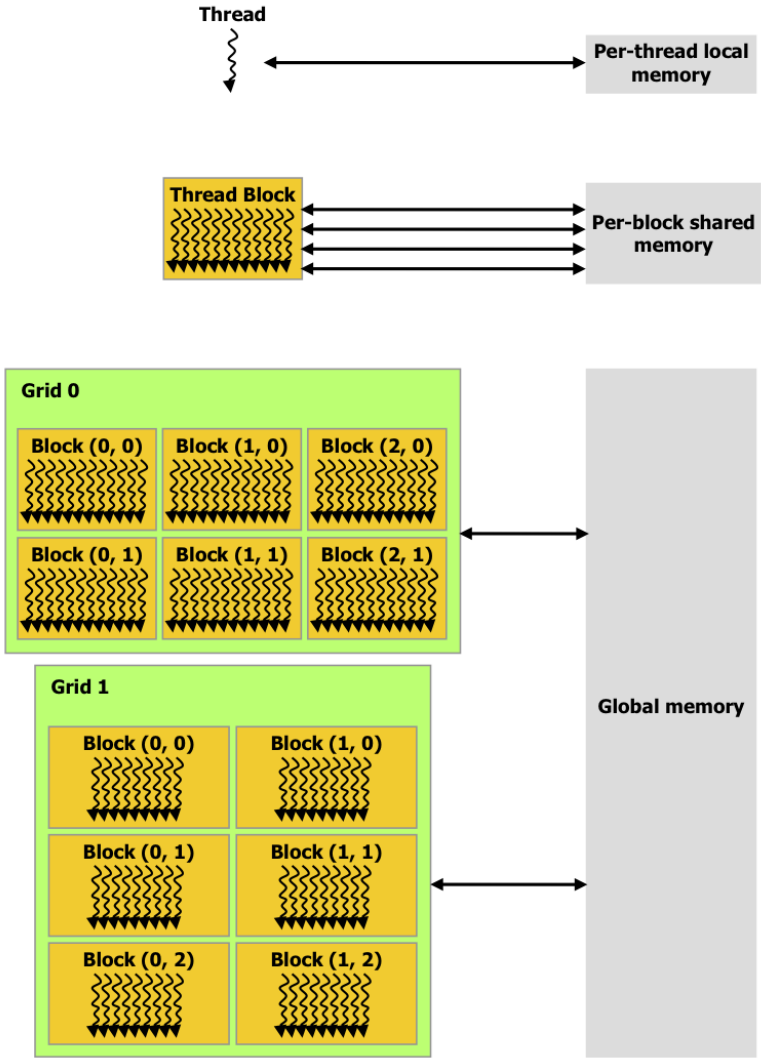


Figure 3.5: *CUDA* : Memory organisation [Credits: *nvidia*]

```
{
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  if(idx < size)
  {
    v1[idx] = v2[idx];
  }
}
```

The following example describes how this kernel can be used and the syntax to call it:

```
int main(void)
{
  int idx;
  int size = make_uint2(256, 256);
  const int nbytes = size * sizeof(float);

  dim3 blockSize(64);
  dim3 gridSize(size.x / blockSize.x);

  float* v1_host = new float[nbytes];
```

```
11    float* v2_host = new float[nbytes];
12    float* v1_dev = NULL, * v2_dev = NULL;
13    cudaMalloc(&v1_dev, nbytes);
14    cudaMalloc(&v2_dev, nbytes);
15
16    for(int i = 0; i < size; i++)
17      v2_host[i] = rand();
18
19    cudaMemcpy(v2_dev, v2_host, nbytes,
20               cudaMemcpyHostToDevice);
21
22    add_kernel<<<gridSize, blockSize>>>(v1_dev, v2_dev, size);
23
24    cudaMemcpy(v1_host, v1_dev, nbytes,
25               cudaMemcpyDeviceToHost);
26
27    for(int i = 0; i < size; i++)
28      printf("%f %f\n", v1_host[i], v2_host[i]);
29
30    cudaFree(v1_dev);
31    cudaFree(v2_dev);
32    delete[] v1;
33    delete[] v2;
34  }
```

Obviously this example is – beyond its quality of example – useless as the overhead of time introduced by the memory transfers will often be higher than the time saved by parallelism. Although for the optical flow problem, two images of a sequence have to be transfered once to the device where they can benefit from several transformations before being transfered back to the host. For instance the computation of the motion tensor J involves many operations: copy, subtractions, convolutions, element-wise multiplication, additions... In this case the transfers represent a small portion of the total runtime. The time saved by parallelisation is thus higher than transfer times and we can experience good speedups.

3.3 Best practises

In order to experience the best performances a few things – at least – have to be taken into consideration when programming a *CUDA* based application. The *CUDA programming guide*[20] and *CUDA best practices guide*[19] are written to detail those things. We strongly advice the reader to read those before being faced to the development of a *CUDA* based application. We want here to explain a few concepts and quote some short extracts of these manuals relevant with our application.

3.3.1 Memory allocations

The memory allocations can be of different kinds. Linear memory can be allocated with `cudaMalloc` on the global memory heap. This memory is said linear in opposition to pitched or 2D memory. When one wants to store a matrix in memory the width of the matrix do not always fit the hardware requirements of the memory. Such a ill-sized matrix can cause a lot of cash misses and therefore low performances.

To avoid that the *CUDA* API provides another function `cudaMallocPitch()` that allocates memory with some padding at the end of lines for it to fit with the hardware layout of the memory. For images pitched memory seems to be ideal. Note that even 3D memory segments can be allocated with `cudaMalloc3D`.

3.3.2 Data transfers

Data transfers have to be constantly considered in a *CUDA* program. Simple index shifts or statement permutations can dramatically slow down an algorithm. The closer to the chip you are the faster the memory is. Thus the thread local memory is fast and can be used without too many considerations.

Shared memory is a bit slower than the thread local memory, plus the amount available is limited. It often impacts on blocks size as, if each thread of a block needs n slots in a shared array of values, the maximum amount of shared memory introduces an upper bound on the blocks size.

The slower memory is the global memory that contains all the data to process. Thus one needs to think twice before accessing it, because each thread of a grid will need to access a particular element of this memory. Memory accesses have to be coalesced to experience the best performances avoiding bus obstruction and cache misses.

The highest memory bandwidth will be achieved by the global memory only when memory accesses are coalesced so the hardware can fetch the data in the smallest number of operations.

The following paragraph from the *CUDA Best Practises Guide* explains the simplest method to do coalesced memory accesses:

The first and simplest case of coalescing can be achieved by any CUDA-enabled device: the k-th thread accesses the k-th word in a segment; the exception is that not all threads need to participate.

[Credits: *nvidia*]

3.3.3 Flow control instructions

As written before, *SIMT instructions specify the execution and branching behaviour of a single thread.* It suggests that flow control statements can be dangerous. For instance, a simple `if` statement can create different execution paths for some consecutive threads in half a warp and then cause the different paths to be executed sequentially instead of in parallel.

The following paragraph from the *CUDA Best Practises Guide* gives more details about flow control instructions use:

Any flow control instruction (if, switch, do, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths. If this happens, the different execution paths must be serialised, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path.

[Credits: *nvidia*]

3.3.4 Texture memory

Texture memory is not properly a kind of memory, it is a method of access using specific hardware to fetch data from the global memory. As its name suggests, it was originally created to fetch textures applied to polygons in graphics. A memory segment allocated on the global heap has to be bound to the texture memory using `cudaBindTexture()` (costless operation) and can then be accessed using `tex1D()`, `tex2D()` or `tex3D()` specific core functions.

Texture memory uses a cache of values that reduces singularly the number of global memory accesses. Moreover, for a simple memory access texture memory fetches can reduce the number of cache misses from hundreds to only one. Even the out-of-bounds cases are addressed by the texture memory. The default mode is `cudaAddressModeClamp` which means that if a value is requested out of bounds, the closer existing value is returned. It allows suppression of flow control instructions that may slow down an algorithm.

The following paragraph from the *CUDA Programming Guide* gives more details about texture memory:

The texture memory space resides in device memory and is cached in texture cache, so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimised for 2D spatial locality, so threads of the same warp that read texture addresses that are close together in 2D will achieve best performance. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.

[Credits: *nvidia*]

3.4 Characteristics of our GPU

The graphic card used for all our experiments is a *GeForce 9400 GT*. This is, even at this moment, a very old device as it was first launched the 27 August of 2008. It embeds one GPU with only 16 stream processors, 512 MB of DDR2 global memory at around 13GB/s. Current models such as *GeForce GTX 480* embed super fast GDDR5 memory at around 177 GB/s. A call to `cudaGetDeviceProperties()` gives us the detailed characteristics in figure 3.7.

```
--- DEVICE 0 ---
name                GeForce 9400 GT     |    major.major              1.1
totalGlobalMem      536150016 bytes     |    clockRate                1400000
sharedMemPerBlock   16384 bytes         |    textureAlignment         256 bytes
regsPerBlock        8192                |    deviceOverlap            1
warpSize            32                  |    multiProcessorCount      2
memPitch            2147483647 bytes    |    kernelExecTimeoutEnabled 1
maxThreadsPerBlock  512                 |    integrated               0
maxThreadsDim       512 x 512 x 64      |    canMapHostMemory         0
maxGridSize         65535 x 65535 x 1   |    computeMode              0
totalConstMem       65536 bytes         |    concurrentKernels        0
```

Figure 3.7: Characteristics of GPU used for experiments

The following characteristics are particularly interesting:

totalGlobalMem: `536150016`
global memory available on device, here approximatively 512 MB

sharedMemPerBlock: `16384`
shared memory available per block, here 16 KB

warpSize: `32`
smallest unit of computation, in threads

maxGridSize: `65535 x 65535 x 1`
maximal size for a grid in blocks

maxThreadsDim: `512 x 512 x 64`
maximal size for a block in threads

minor, major: `1, 1`
compute capability of the device and thus operations allowed or not, here 1.1

3.5 Effective speedups

To illustrate the speedup provided by a GPU we take a simple element-wise multiplication kernel:

```
1  template<typename T>
2  __global__
3  void MatEltMul(T* A,
4                 T* B,
5                 T* C,
6                 uint2 size,
7                 uint3 pitch,
8                 uint2 margins)
9  {
10   int i = blockIdx.y * blockDim.y + threadIdx.y;
11   int j = blockIdx.x * blockDim.x + threadIdx.x;
12
13   if (i < size.y && j < size.x){
14     i += margins.y;
15     j += margins.x;
16
```

```
17      L_ELEM(A, i, j, pitch.x) =
18        L_ELEM(B, i, j, pitch.y) *
19        L_ELEM(C, i, j, pitch.z);
20    }
21  }
```

3.5.1 Time versus image size

Let us define I^1, I^2 and I^3 some random square images of length $size$. If we measure the time taken to do $I^1_{i,j} = I^2_{i,j} * I^3_{i,j}$ for $i, j = 1 \dots size$, with average on 16 iterations, we obtain the plot in figure 3.8. It is easy to see that the CPU time increases a lot faster than the GPU time which means less parallelisation. Although the GPU time is not linear, we can see that it increases slowly with size of data due to overheads in kernel calls.

3.5.2 Bandwidth versus image size

Algorithms in *CUDA* , according to the best practices guide, should be benchmarked using the actual bandwidth (here in MegaPixels per second). We can see on figure 3.9 that for too small sets of data the best GPU bandwidth is not achieved while it stays constant for CPU. For an image of 2000×2000 pixels, the best bandwidth is achieved and is around 3 times higher than the CPU's one.

3.5.3 Speedup versus image size

Finally the speedup is plotted in figure 3.10. We can see that a good speedup is not achieved for really small sets of data because of the overhead introduced. It is not always better to do small computations on the CPU though, the memory transfers times can avoid one to use a faster CPU algorithm.

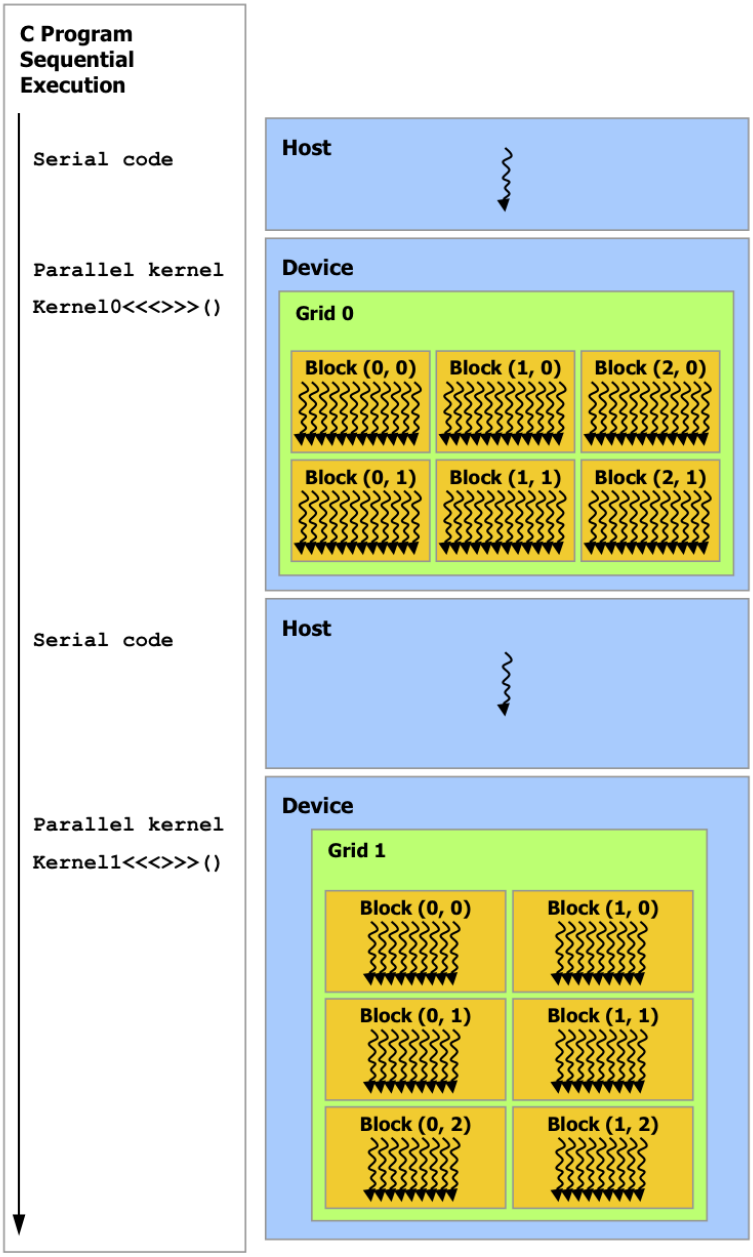


Figure 3.6: *CUDA* : Heterogeneous programming

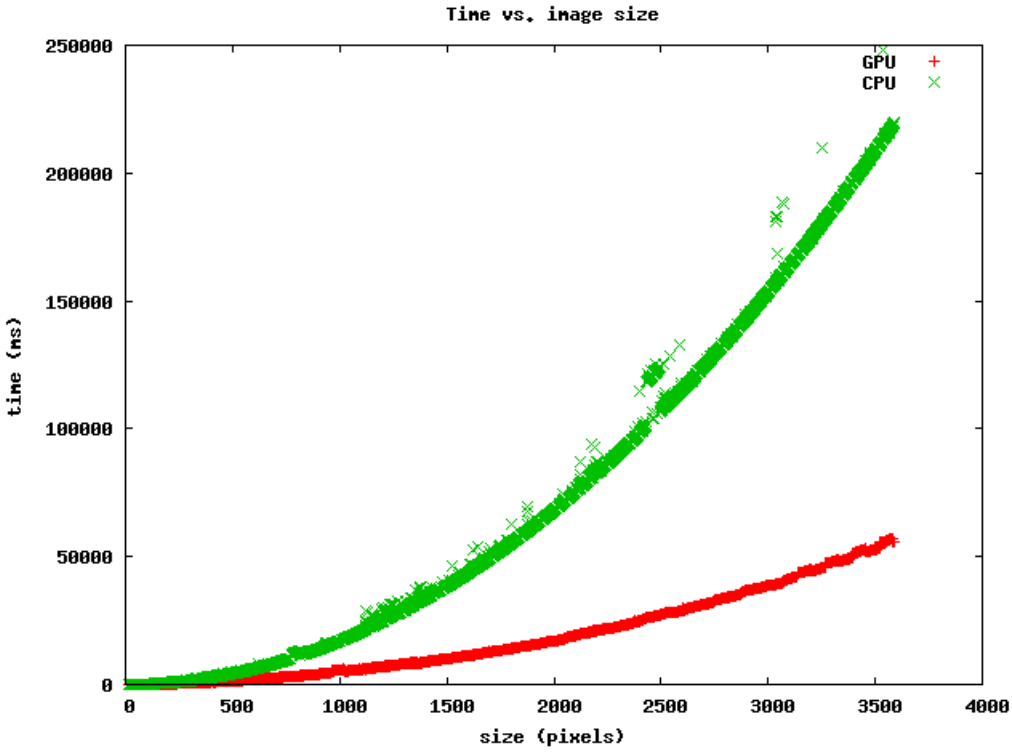


Figure 3.8: Time of computation versus image size for element-wise matrix multiplication: CPU (green), GPU (red)

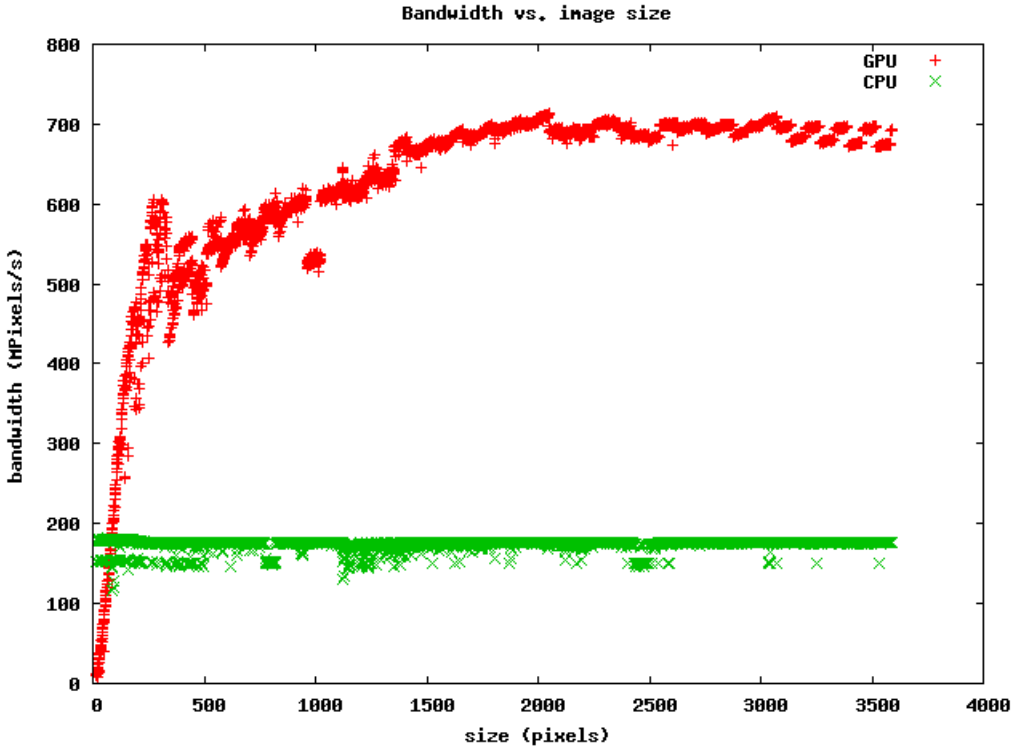


Figure 3.9: Bandwidth versus image size for element-wise matrix multiplication: CPU (green), GPU (red)

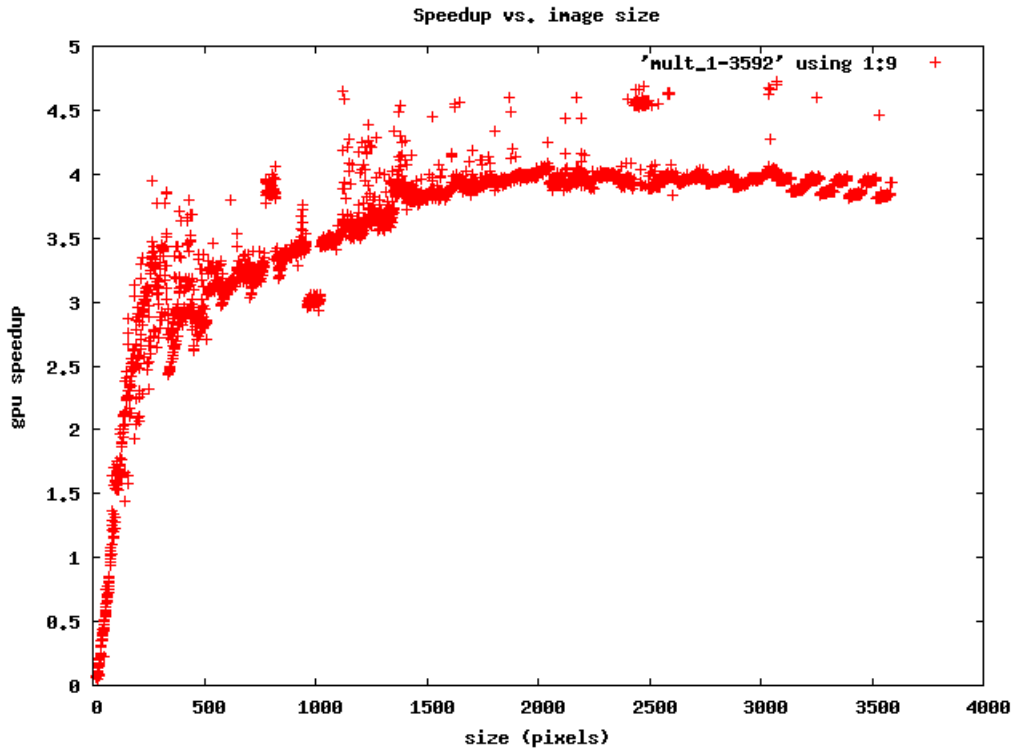


Figure 3.10: Speedup versus image size for element-wise matrix multiplication

Chapter 4

Implementation details

4.1 Data structures

As we have seen in the previous chapter there are many ways to allocate memory. In addition there are also multiple ways to represent an image in memory. There is no layout better than another, they are all used at a specific part of the algorithm in order to experience the best performances.

4.1.1 Interlaced matrices

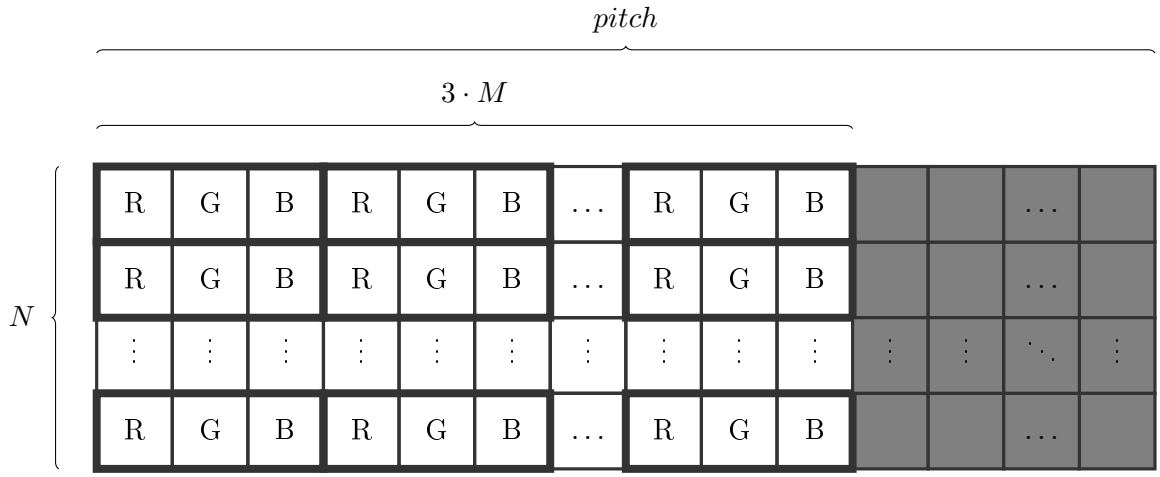


Figure 4.1: Interlaced matrices layout

Interlaced matrices (type `MatrixInterlaced`) were the first layout to come at mind as it is the layout used in many images format. The *OpenCV* library, used for format encoding and decoding, for instance uses this layout when decoding a `png` or `jpg` file from the disk.

For each pixel the red, green and blue values are stored consecutively. These pixel tuples are stored in a row-major scheme using 2D pitched memory. This layout is illustrated in figure 4.1. The formulae to find a given pixel in that layout is given by:

$$\begin{aligned} \forall(i,j,k) &\in \{1,\ldots,N\} \times \{1,\ldots,M\} \times \{1,2,3\} \\ M(i,j,k) &= \texttt{ptr[}i * pitch + j * 3 + k\texttt{]} \end{aligned} \tag{4.1}$$

It has the benefit that no processing has to be done on image after reading it from disk. It can be directly transferred from host to device with a simple `cudaMemcpy` and without using any, potentially slow, kernel to reorganise the values.

It can also be used to store flows as they are 2 channels floating point images. For the flows this layout is preferred because x- and y-components are always computed together in kernels. Thus it enables us to do coalesced memory accesses and therefore speeds-up the execution. It is not only used for the final flow but also in the whole solving process for *Horn & Schunck* and *CLG* methods.

```
1 template<typename T>
2 struct MatrixInterlaced {
3   T* ptr;
4   size_t pitch_bytes;
5   size_t width_bytes;
6   uint4 size;
7 };
```

This layout is implemented as a template structure for it to handle any type of values. Besides a pointer `ptr` to the allocated memory, the structure includes a `uint4` value `size` where `size.x` is the width, `size.y` the height, `size.z` the number of channels and `size.w` the pitch in number of elements. The pitch and width in bytes are also included as they are used in *CUDA* core functions such as `cudaMemcpy`.

4.1.2 Layers matrices

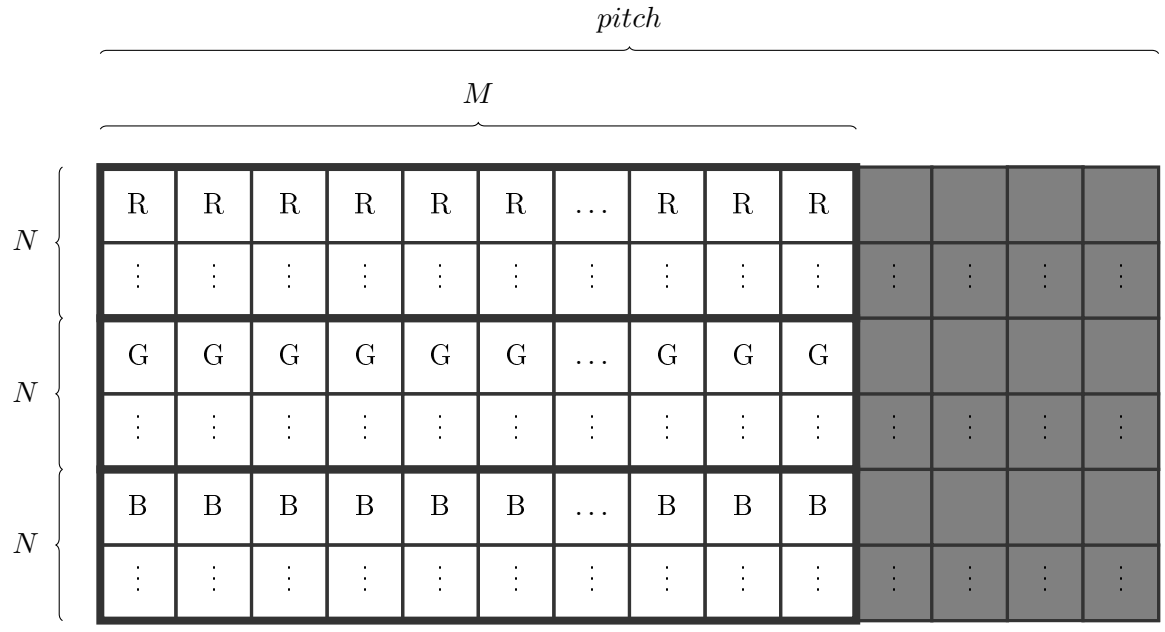


Figure 4.2: Layers matrices layout

We first implemented layers matrices (type `MatrixLayers`) because of performance problems we had in our convolution kernels. Despite the fact that a color convolution can be done separately on the different channels, at the beginning we tried to compute it for all the channels in the same kernel. But shared memory is limited and doing that involved changing the block sizes and other constants to

finally end up with a really slow algorithm. We then realised that it was easier and faster to execute a 1-layer kernel on each of the channels. To be able to achieve coalesced memory accesses this layout was implemented.

For each channel the values are stored in a row-major scheme using 2D pitched memory, and the channels are stored consecutively in memory. This layout is illustrated in figure 4.2. The formulae to find a given pixel in that layout is given by:

$$\forall(i,j,k) \in \{1,\ldots,N\} \times \{1,\ldots,M\} \times \{1,2,3\} \tag{4.2}$$

$$M(i,j,k) = \texttt{ptr[k - 1][}i * pitch + j\texttt{]} \tag{4.3}$$

This layout have the benefit that a k-layers image can easily be flattened into a 1-layer image doing coalesced memory accesses. Also note that for a 1-channel image this layout gives the same memory layout as the interlaced one. Nevertheless we prefer to use the interlaced one in that case.

```
template<typename T, int k>
struct MatrixLayers{
  T* ptr[k];
  uint3 size;
  int pitch[k];
  size_t pitch_bytes[k];
  size_t width_bytes;
};
```

This layout is implemented as a template structure for it to handle any type of values. A pointer to each layer allocated memory is stored in the array `ptr[]`. The structures includes a `uint4` value `size` where `size.x` is the width, `size.y` the height, `size.z` the number of channels and `size.w` the pitch in number of elements. The pitch and width in bytes are also included.

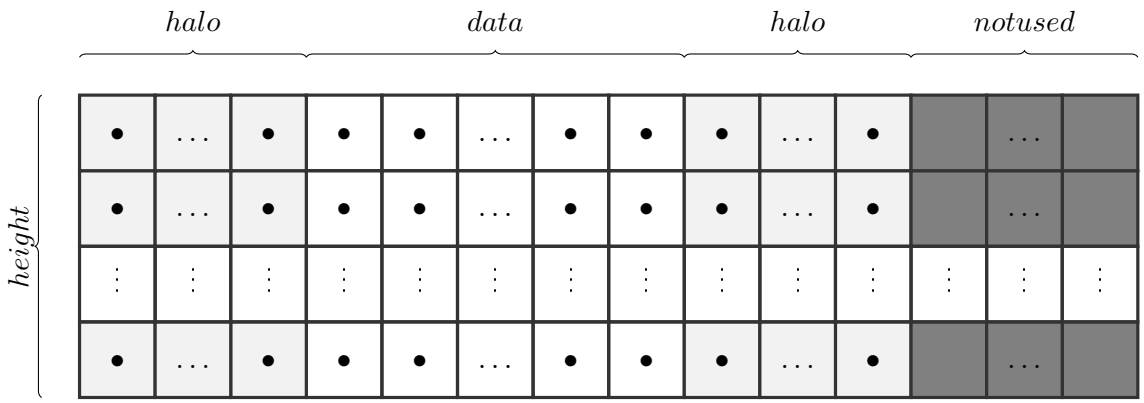
4.2 Advanced operations

The data structures being defined we detail in this section how we use them in various operators implemented.

4.2.1 Convolution

As we use convolution only for Gaussian smoothing and derivation, in order to save time, we can always operate separate convolution. We then need one kernel to convolve rows and another for columns. The *CUDA SDK* is shipped with some very interesting sample codes, one of them achieving parallel separate convolution. We took that code and modified it a bit, for instance we introduced texture memory accesses. We also modified it for *halo_steps* and *kernel_radius* to be dynamically valued. But let us explain step by step what does this kernel do. We explain here the row convolution kernel, the column one being only a transposition of it.

Each block has $ROWS\_BLOCSIZE\_X \times ROWS\_BLOCKSIZE\_Y$ threads, and one thread works on $ROWS\_RESULT\_STEPS$ values on its line. Each block is allocated a floating point array of shared memory with as many lines as the block will process and enough columns to fit $ROWS\_RESULT\_STEPS$ values plus two times the necessary halo (actually an upper bound MAX_HALO_SIZE on the halo size). The layout of this array is depicted in figure 4.3.



$$
\begin{aligned}
height &= ROWS\_BLOCKDIM\_Y \\
data &= ROWS\_RESULT\_STEP * ROWS\_BLOCKDIM\_X \\
halo &= halo\_steps * ROWS\_BLOCKDIM\_X \\
notused &= 2 * (MAX\_HALO\_STEPS - halo\_steps) * ROWS\_BLOCKDIM\_X
\end{aligned}
$$

Figure 4.3: Layout of block-shared array in convolution kernel

Then values need to be loaded from the image to the shared array. It is done in three parts:

- main data is loaded
- left halo is loaded
- right halo is loaded

For each of these parts, values are loaded by $ROWS\_RESULT\_STEPS$ successive steps of $ROWS\_BLOCKDIM\_X$ simultaneous memory transfers. At each step the thread k of the current block loads the value $k * ROWS\_BLOCKDIM\_X$. This is done that way for the memory accesses to be coalesced. This loading process is illustrated in figure 4.4.

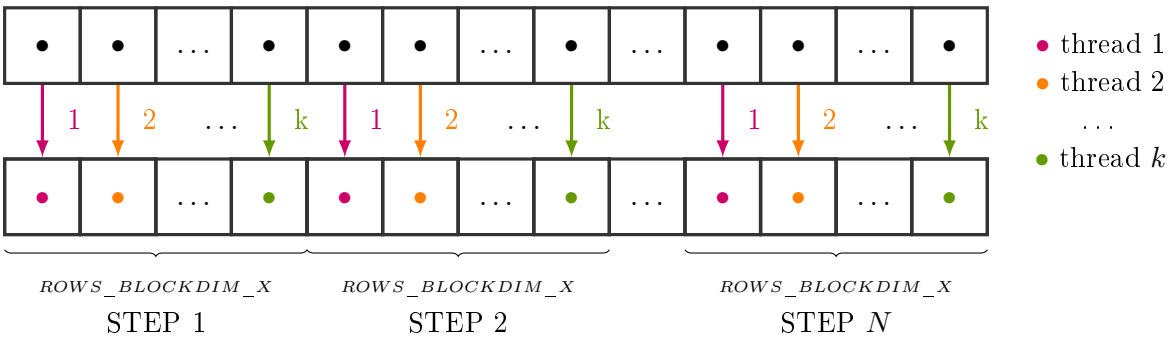


Figure 4.4: Coalesced image copy to block-shared memory in convolution kernel

Once the values loaded in shared memory, the convolution process can take place. For a given kernel K of radius r we have $K = (K_{-r}, \ldots, K_0, \ldots, K_r)^\mathsf{T}$ and each element I_i of the image is computed as:

$$I_i = \sum_{j=-r}^{r} K_j \cdot I_{i+j} \tag{4.4}$$

Here again the whole computation is done in *ROWS_RESULT_STEPS* successive steps, using the same scheduling as for loading. The figure 4.5 illustrates this process for a given line. Note that the same steps are used for computation as for the data loadings, the figure does not illustrate it.

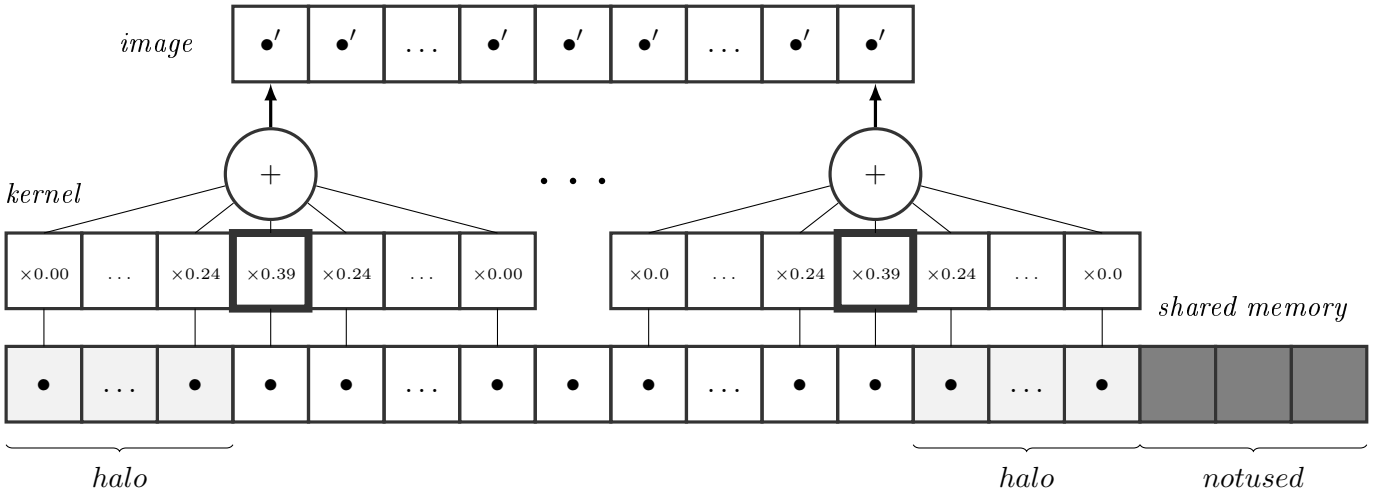


Figure 4.5: Parallel convolution process

Finally we give the modified version of the sample code that we use in the application :

```
__global__
void convolutionRowsKernel(
  float *d_Dst, float *d_Src, uint2 size, uint2 pitch, int halo_steps, int k_rad
){

  __shared__ float s_Data[ROWS_BLOCKDIM_Y]
                         [(ROWS_RESULT_STEPS + 2 * MAX_HALO_STEPS) * ROWS_BLOCKDIM_X];

  //Offset to the left halo edge
  const int baseX =
    (blockIdx.x * ROWS_RESULT_STEPS - halo_steps) * ROWS_BLOCKDIM_X + threadIdx.x;
  const int baseY =
    blockIdx.y * ROWS_BLOCKDIM_Y + threadIdx.y;

  //float *d_Src_old = d_Src;
  d_Src += (baseY * pitch.x + baseX);
  d_Dst += (baseY * pitch.y + baseX);

  //Load main data
  for(int i = halo_steps; i < halo_steps + ROWS_RESULT_STEPS; i++)
    s_Data[threadIdx.y] [threadIdx.x + i * ROWS_BLOCKDIM_X] =
      tex2D(texRef, baseX + i * ROWS_BLOCKDIM_X, baseY);

  //Load left halo
  for(int i = 0; i < halo_steps; i++)
    s_Data[threadIdx.y] [threadIdx.x + i * ROWS_BLOCKDIM_X] =
      tex2D(texRef, baseX + i * ROWS_BLOCKDIM_X, baseY);

```

```
29    for(int i = halo_steps + ROWS_RESULT_STEPS; i < 2 * halo_steps + ROWS_RESULT_STEPS; i++)
30      s_Data[threadIdx.y] [threadIdx.x + i * ROWS_BLOCKDIM_X] =
31        tex2D(texRef, baseX + i * ROWS_BLOCKDIM_X, baseY);
32
33    //Compute and store results
34
35    __syncthreads();
36
37    for(int i = halo_steps; i < halo_steps + ROWS_RESULT_STEPS; i++){
38      float sum = 0;
39
40      for(int j = -k_rad; j <= k_rad; j++){
41        sum += KERNEL[k_rad + j] *
42          s_Data[threadIdx.y] [threadIdx.x + i * ROWS_BLOCKDIM_X + j];
43      }
44
45      if(baseX + i * ROWS_BLOCKDIM_X < size.x)
46        d_Dst[i * ROWS_BLOCKDIM_X] = sum;
47    }
48  }
```

4.2.2 Gaussian smoothing

One 2D Gaussian kernel can be split in two 1D Gaussian kernels that can be used with the convolution code previously introduced. Obviously the bigger the σ is the slower the convolution will be because the halo size depends on σ. The maximum size of the halo being statically defined, an upper bound of σ_{max} exists.

We define the support of the Gaussian as $[-3\sigma; \sigma]$, this is conventionally done because it captures 99% of the energy of the Gaussian function. Discrete 1D Gaussian kernel is thus computed using the following formulae:

$$K_\sigma(i) = \exp\left(\frac{0.5}{\sigma^2} i^2\right) \Big/ \left(\sum_{j=i_{min}}^{i_{max}} K_\sigma(j)\right), \quad i = -\lfloor 3\sigma \rfloor, \ldots, \lfloor 3\sigma \rfloor \tag{4.5}$$

4.2.3 Derivation

The computation of the motion tensor involves first order derivative calculus.

Time derivative are only computed with a 2 points stencil implemented with a simple subtraction kernel:

$$I_t = I_2 - I_1 \tag{4.6}$$

Derivatives with respect to x and y are computed by convolution with fourth order precision kernel as detailed in appendix A.4:

$$K_x = \frac{1}{12}(1, -8, 0, 8, -1), \quad K_y = \frac{1}{12}\begin{pmatrix} & 1 \\ - & 8 \\ & 0 \\ & 8 \\ - & 1 \end{pmatrix} \tag{4.7}$$

Finally, for a pair of images from a sequence, the derivatives computed on both images are simply averaged:

$$I_x = \frac{1}{2}\left(K_x \star I_1 + K_x \star I_2\right) \tag{4.8}$$

$$I_y = \frac{1}{2}\left(K_y \star I_1 + K_y \star I_2\right) \tag{4.9}$$

4.2.4 Restriction and prolongation

Restriction and prolongation operators are key features of multigrid solvers (see [4][16]). The way these operators are implemented impacts a lot quality of the solution we end up with. They have to stay fast though. A perfect operator would consider each border case specifically, introducing a lot of `if` statements. Although we have seen in chapter 3 that those slow down execution a lot. As a result we tried to keep a good balance between accuracy and runtime.

4.2.5 Restriction

Restriction operator does a simple averaging on fine image values to compute the coarse image. These averages have to overlap in order to avoid introducing of high-frequent errors. We execute one thread per value $v_{i,j}$ to compute in the coarse image. Each thread use texture memory accesses to fetch the values $V_{k,l}$ from the fine image. The sum is weighted using the w coefficient matrix:

$$w = \begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix} \tag{4.10}$$

$$V = \begin{bmatrix} V_{2i-1,2j-1} & V_{2i-1,2j} & V_{2i-1,2j+1} \\ V_{2i,2j-1} & V_{2i,2j} & V_{2i,2j+1} \\ V_{2i+1,2j-1} & V_{2i+1,2j} & V_{2i+1,2j+1} \end{bmatrix} \tag{4.11}$$

$$v_{i,j} = \sum_{k=1}^{3}\sum_{l=1}^{3} w_{k,l} \cdot V_{k,l} \tag{4.12}$$

Obviously the problems are located at the borders. To experience highest speedup we let the texture memory functions do the job. The default mode for texture out-of-range fetches being `cudaAddressModeClamp`, everything behaves as if the border lines were replicated outside of the range. This was hard choice, but implementing a succession of `if` statements was definitely too slow. Some more improvements could be made at that point.

Thus, at the borders the coefficient matrix is changed:

$$w_N = w_W^{\mathsf{T}} \begin{bmatrix} 0 & 0 & 0 \\ 3/16 & 3/8 & 3/16 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}, \quad w_S = w_E^{\mathsf{T}} \begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 3/16 & 3/8 & 3/16 \\ 0 & 0 & 0 \end{bmatrix}, \quad w_{NW} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 9/16 & 3/16 \\ 0 & 3/16 & 1/16 \end{bmatrix}, \quad \ldots \tag{4.13}$$

The behaviour of the restriction operator is illustrated in figure 4.6.

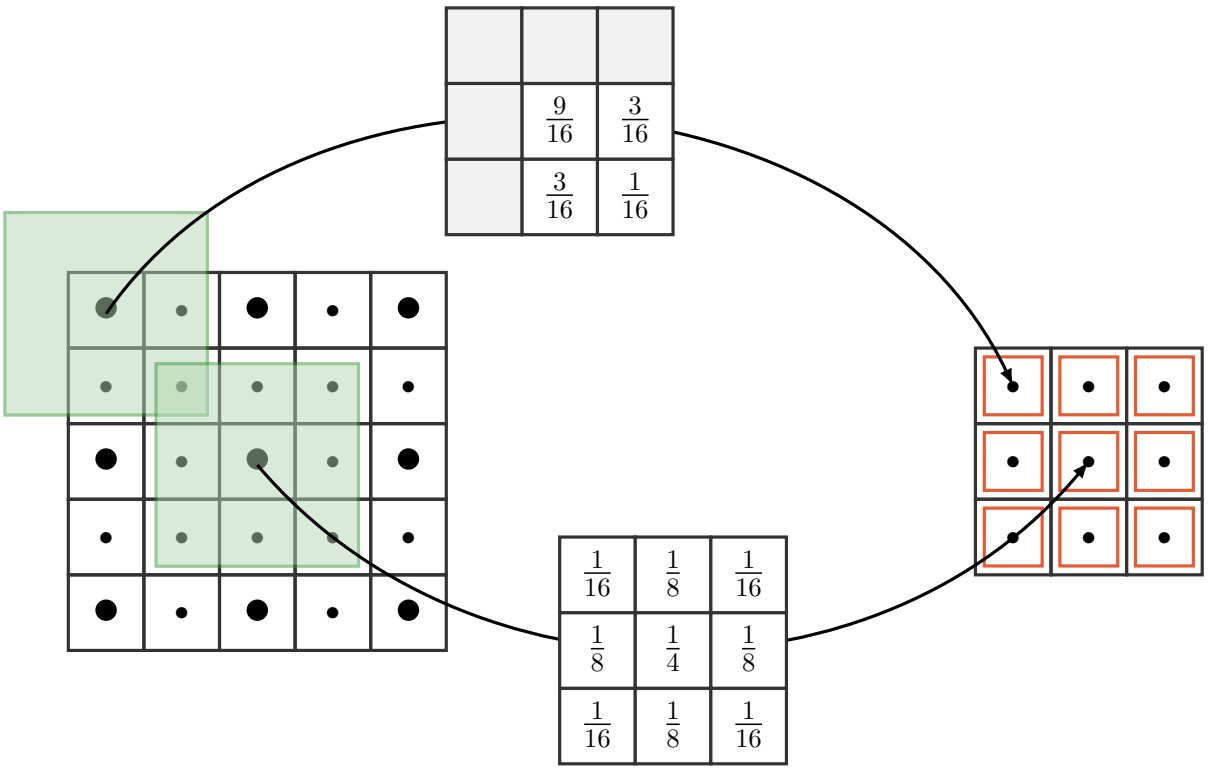


Figure 4.6: Parallel restriction operator: ***Red*** *squares surround coarse values written by a single thread.* ***Green*** *squares surround fine values used for the computation of the coarse value at the end of the arrow. On the arrows are represented the coefficient matrix used for that particular point. Note that green squares are only special cases, if we wanted to draw all of them the fine image would have been all green.*

Finally we give the source code for the restriction operator kernel:

```
__global__
void restriction(float* reduced, uint4 size_red, uint4 size){
  const int i_red = threadIdx.y + blockIdx.y * blockDim.y;
  const int j_red = threadIdx.x + blockIdx.x * blockDim.x;

  if(i_red < size_red.y && j_red < size_red.x){
    const int i = i_red * 2;
    const int j = j_red * 2;
    float sum;

    for(int k = 0; k < size_red.z; k++){
      sum = 0.25f *    tex2DI(texRef, j    , i    , k, size.z);
      sum += 0.125f *  tex2DI(texRef, j    , i + 1, k, size.z);
      sum += 0.125f *  tex2DI(texRef, j + 1, i    , k, size.z);
      sum += 0.125f *  tex2DI(texRef, j - 1, i    , k, size.z);
      sum += 0.125f *  tex2DI(texRef, j    , i - 1, k, size.z);
      sum += 0.0625f * tex2DI(texRef, j + 1, i + 1, k, size.z);
      sum += 0.0625f * tex2DI(texRef, j - 1, i - 1, k, size.z);
      sum += 0.0625f * tex2DI(texRef, j + 1, i - 1, k, size.z);
      sum += 0.0625f * tex2DI(texRef, j - 1, i + 1, k, size.z);

      I_ELEM(reduced, i_red, j_red, k, size_red) = sum;
    }
```

```
24   }
25 }
```

4.2.6 Prolongation

Prolongation operator is defined as the dual operation of the restriction one. Therefore it does a simple linear interpolation on coarse image values to compute the fine image. These interpolations have to overlap too in order to avoid introducing of high-frequent errors. What is practically done is that that each coarse value $v_{i,j}$ is copied to the fine value $V_{2i,2j}$, then we apply the following interpolation rules:

$$V_{2i+1,2j} = \frac{1}{2}(v_{i,j} + v_{i+1,j}) \tag{4.14}$$

$$V_{2i,2j+1} = \frac{1}{2}(v_{i,j} + v_{i,j+1}) \tag{4.15}$$

$$V_{2i+1,2j+1} = \frac{1}{4}(v_{i,j} + v_{i,j+1} + v_{i+1,j} + v_{i+1,j+1}) \tag{4.16}$$

The same remarks regarding the borders can be made about the prolongation operator as for the restriction one. But for the prolongation the problems only happen in right and bottom borders.

The behaviour of the prolongation operator is illustrated in figure 4.7.

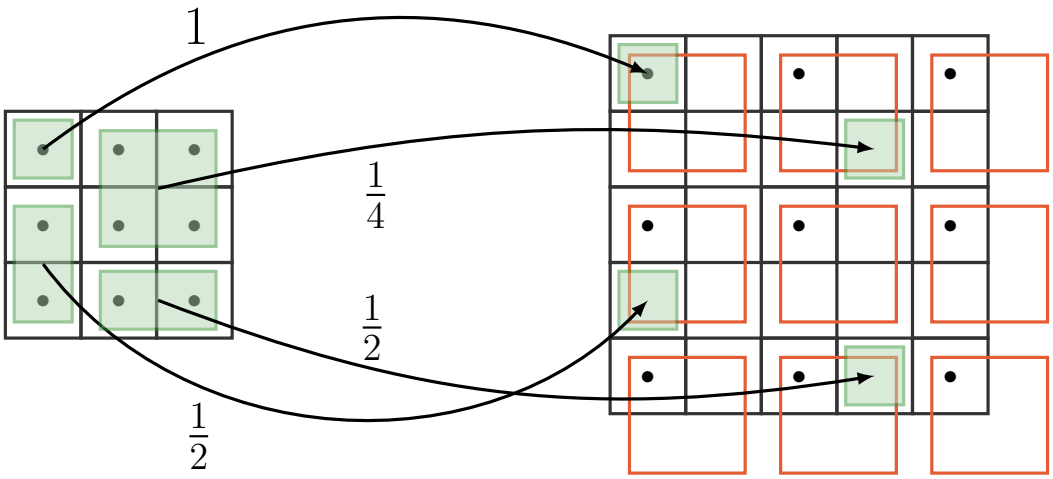


Figure 4.7: Parallel prolongation operator: ***Red*** *squares surround fine values written by a single thread.* ***Green*** *squares surround coarse values used for the computation of the coarse value at the end of the arrow. On the arrows are represented the weight used to sum the green neighbourhood. Note that green squares are only special cases, if we wanted to draw all of them the fine image would have been all green.*

Finally we give the source code for the prolongation operator kernel:

```
1 __global__
2 void prolongation(
3   float* prolongated,
4   uint4 size_pro,
5   uint4 size
6 ){
```

```
7   const int i = threadIdx.y + blockIdx.y * blockDim.y;
8   const int j = threadIdx.x + blockIdx.x * blockDim.x;
9
10  const int i_pro = i * 2;
11  const int j_pro = j * 2;
12
13  if(i_pro < size_pro.y && j_pro < size_pro.x){
14
15    for(int k = 0; k < size_pro.z; k++){
16      I_ELEM(prolongated, i_pro, j_pro, k, size_pro) = tex2DI2(texRef, j, i, k);
17
18      if(j_pro < size_pro.x - 1)
19        I_ELEM(prolongated, i_pro, j_pro + 1, k, size_pro) =
20          0.5f * tex2DI2(texRef, j, i, k) +
21          0.5f * tex2DI2(texRef, j + 1, i, k);
22
23      if(i_pro < size_pro.y - 1)
24        I_ELEM(prolongated, i_pro + 1, j_pro, k, size_pro) =
25          0.5f * tex2DI2(texRef, j, i, k) +
26          0.5f * tex2DI2(texRef, j, i + 1, k);
27
28      if(i_pro < size_pro.y - 1 && j_pro < size_pro.x - 1)
29        I_ELEM(prolongated, i_pro + 1, j_pro + 1, k, size_pro) =
30          0.25f * tex2DI2(texRef, j, i, k)           +
31          0.25f * tex2DI2(texRef, j, i + 1, k)       +
32          0.25f * tex2DI2(texRef, j + 1, i, k)       +
33          0.25f * tex2DI2(texRef, j + 1, i + 1, k);
34    }
35  }
36 }
```

4.3 Motion tensor computation

This section summarises and illustrates the steps necessary to the computation of the motion tensor, starting point of all implemented algorithms.

4.3.1 Memory transfers and data conversion

Firstly images need to be transfered from the hard disk drive to the computer RAM, this is done using *OpenCV* as this library can deal with a lot of formats. *OpenCV* loads images in RAM as `unsigned char` arrays, we need to transfer those to the GPU device memory. The `cudaMemcpy` function does that with the flag `cudaMemcpyHostToDevice`.

From this point we will only deal with floating point values, hence a conversion from `unsigned char` to `float` is made using a simple copy kernel. As the choice of *OpenCV* can be discussed, please note that a library which would return directly `float` arrays from image files would make the transfer four times slower.

4.3.2 Presmoothing

After that images need to be presmoothed as explained in 2.3. They are then convolved with the Gaussian kernel K_σ. The figure 4.8 shows two images of this sequence f_1, f_2, and their presmoothed versions.

Figure 4.8: Presmoothed images: f_1, $K_\sigma \star f_1$, f_2, $K_\sigma \star f_2$

4.3.3 Derivative products

Derivatives f_x, f_y and f_t are computed as explained in 4.2.3. We give no figure fore that because, as the values are really small, the images look quite dark.

Products between derivatives are computed with point-wise matrix multiplication kernel:

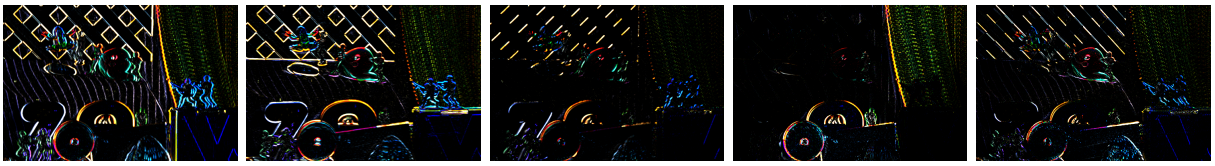


Figure 4.9: Derivatives products: f_x^2, f_y^2, $f_x f_y$, $f_x f_t$, $f_y f_t$

4.3.4 Postsmoothing

Then postsmoothing is done by separable convolution with a Gaussian kernel K_ρ:

Figure 4.10: Postsmoothed derivatives products: $K_\rho \star f_x^2$, $K_\rho \star f_y^2$, $K_\rho \star (f_x f_y)$, $K_\rho \star (f_x f_t)$, $K_\rho \star (f_y f_t)$ ($\rho = 6.0$)

4.3.5 Motion tensor color layers merging

For a color image, as you can see, the postsmoothed products are still 3-layers images. This represents the motion information found for each color channel of the image. In our case we want to use all the information available. Hence we sum the three channels of each image to get the final motion tensor. The kernel to do so is straightforward: each thread reads one value from each layer of the 3-channels image, sum them and stores the result a 1-channel one. Note that here again coalesced memory accesses are performed which results in a fast execution. The figure 4.11 shows the resulting sums.

Figure 4.11: Motion tensor components: J_{11}, J_{22}, J_{12}, J_{13}, J_{23} $(\rho = 6.0)$

4.3.6 Overall algorithm

We summarise the overall algorithm in the figure 4.12.

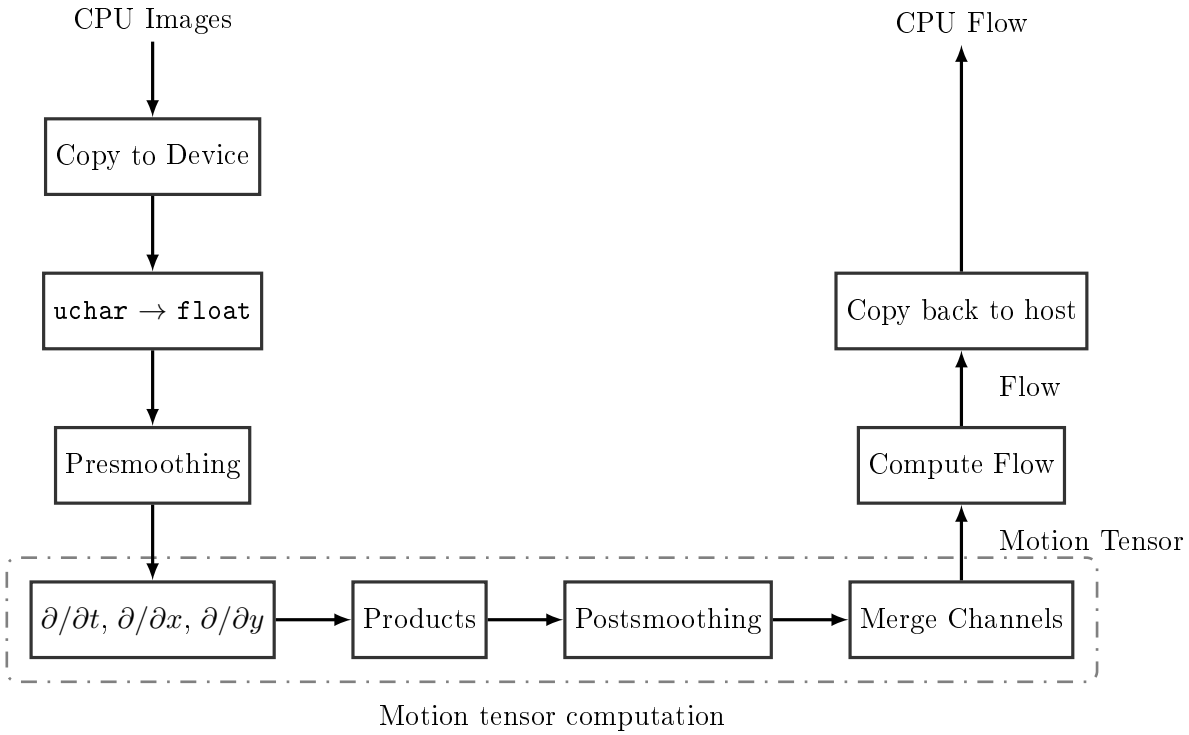


Figure 4.12: GPU optical flow overall algorithm: motion tensor computation

4.4 Solvers

This section describes the solvers implemented to compute optical flows using *Horn & Schunck* and *CLG* methods.

4.4.1 Jacobi solver

The Jacobi solver implementation is really close to the definition given in 2.8.1 (see also [1]). At first the matrix introduced in 2.6 is computed. Obviously a sparse matrix representation is used as the matrix would need more than terabytes of memory to be densely stored. Only the different diagonals are then stored.

In the theoretical aspects, we use a matrix form for our systems $M\mathbf{x} = \mathbf{f}$. But we concretely implement vectors as 2-channels `MatrixInterpolated` and matrix diagonals as well. It makes sense because we already said that a flow is an image with all the constraints that come with – mainly topology – and it would make no sense to represent it as a linear vector. Thus in order to do coalesced memory accesses and save some computation time we represent matrix diagonals in that way as well.

In fact only the principal diagonal has to be computed. The other diagonals are only constant values or zero depending on the topology. To avoid use of flow control statements in the solver iteration kernel, we precompute masks (0 or 1) depending on the topology. Then, during the iterations, we only need to multiply the mask by the constant value.

The matrix computation kernel (see [3]) computes the following values:

the diagonal terms: used in the residual computation

the inverted diagonal terms: used in the solvers

the laplacian terms of the diagonal: used in the non linear solvers

the motion tensor terms of the diagonal: used in the nonlinear solvers as well

external diagonals masks: to save time in iterations

The diagonal being the sum of the laplacian and motion tensor terms, this computation does a bit of redundancy. This call is certainly not the most expensive of the application, nevertheless some of the aforementioned computations can be disabled if using only nonlinear or linear solvers. The right hand $\mathbf{f}$ is computed from the motion tensor with simple texture fetches in a kernel.

Finally the solver can be run on those data in two modes. The *error mode* sets a target error and iterates until this target error is reached. The *iterations mode* directly sets a number of iterations that will be executed. The solver being iterative, it involves the use of an old and new solution. Copying constantly the new solution into the old one before computing would be very inefficient. We therefore use a pair of vectors that we permute without needing to do any copy.

The source code of the Jacobi solver kernel is given there:

```
1  __global__
2  void jacobi_it(
3    float* x, uint4 size, float2 aH2inv, float sor_factor, float one_m_sor_factor
4  ){
5    const int i = threadIdx.y + blockIdx.y * blockDim.y;
6    const int j = threadIdx.x + blockIdx.x * blockDim.x;
7
8    if(i < size.y && j < size.x){
9      float2 sum, aii_inv, fi;
10     float im1 = -tex2DI2(mask_b_ref, j, i, 1);
11     float ip1 = -tex2DI2(mask_a_ref, j, i, 1);
12     float jm1 = -tex2DI2(mask_b_ref, j, i, 0);
13     float jp1 = -tex2DI2(mask_a_ref, j, i, 0);
14
15     aii_inv.x = tex2DI2(diag_inv_ref, j, i, 0);
16     aii_inv.y = tex2DI2(diag_inv_ref, j, i, 1);
17     fi.x = tex2DI2(f_ref, j, i, 0);
18     fi.y = tex2DI2(f_ref, j, i, 1);
19
```

```
    sum.x = tex2DI1(jxy_ref, j, i) *
      tex2DI2(x_old_ref, j, i, 1);
    sum.y = tex2DI1(jxy_ref, j, i) *
      tex2DI2(x_old_ref, j, i, 0);

    sum.x += ip1 * tex2DI2(x_old_ref, j, i + 1, 0);
    sum.y += ip1 * tex2DI2(x_old_ref, j, i + 1, 1);
    sum.x += jp1 * tex2DI2(x_old_ref, j + 1, i, 0);
    sum.y += jp1 * tex2DI2(x_old_ref, j + 1, i, 1);
    sum.x += jm1 * tex2DI2(x_old_ref, j - 1, i, 0);
    sum.y += jm1 * tex2DI2(x_old_ref, j - 1, i, 1);
    sum.x += im1 * tex2DI2(x_old_ref, j, i - 1, 0);
    sum.y += im1 * tex2DI2(x_old_ref, j, i - 1, 1);

    I_ELEM(x, i, j, 0, size) =
      sor_factor * (aii_inv.x * (fi.x - sum.x))     +
      one_m_sor_factor * tex2DI2(x_old_ref, j, i, 0);

    I_ELEM(x, i, j, 1, size) =
      sor_factor * (aii_inv.y * (fi.y - sum.y))     +
        one_m_sor_factor * tex2DI2(x_old_ref, j, i, 1);
  }
}
```

For the Yosemite sequence this solver converges in a few thousands of iterations. On the GPU this executes a lot faster than a CPU point-wise Gauss-Seidel but it is still poor performances compared to multigrid solvers. The figure 4.13 depicts the convergence of the solver for different number of iterations.

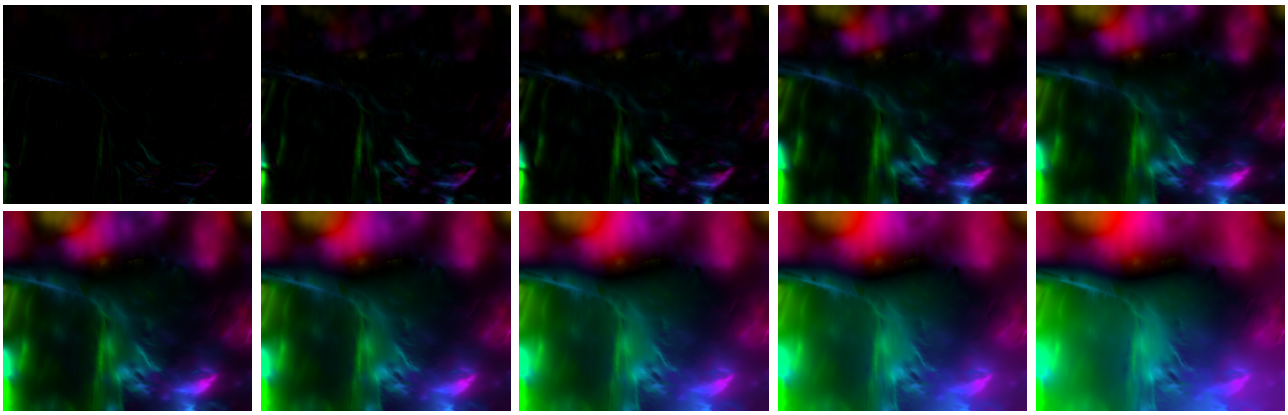


Figure 4.13: Convergence of the Jacobi solver for different number of iterations: 10, 50, 100, 300, 500, 700, 1000, 1500, 2000, 3000

4.4.2 Jacobi solver with lagged nonlinearities

The non linear Jacobi solver is really close to the linear one, they use a lot of function stubs in common. The difference is that operators have constantly to be updated with the new solution values. To this end we need to compute Ψ'_S and Ψ'_D from the freshly computed solution $\mathbf{x}$. The source code of the Ψ'_S computation kernel is given:

```
__global__
void psi_s_kernel(
  float* psi_s,
  uint4 size
```

```
){
  const int i = threadIdx.y + blockIdx.y * blockDim.y;
  const int j = threadIdx.x + blockIdx.x * blockDim.x;

  float2 d[2];
  float val;

  if(i < size.y && j < size.x){
    for(int k = 0; k < 2; k++){
      d[k].x =   1.0f * tex2DI2(x_ref, j - 2, i, k);
      d[k].x += -8.0f * tex2DI2(x_ref, j - 1, i, k);
      d[k].x +=  8.0f * tex2DI2(x_ref, j + 1, i, k);
      d[k].x += -1.0f * tex2DI2(x_ref, j + 2, i, k);

      d[k].y =   1.0f * tex2DI2(x_ref, j, i - 2, k);
      d[k].y += -8.0f * tex2DI2(x_ref, j, i - 1, k);
      d[k].y +=  8.0f * tex2DI2(x_ref, j, i + 1, k);
      d[k].y += -1.0f * tex2DI2(x_ref, j, i + 2, k);
    }

    val = d[0].x * d[0].x + d[0].y * d[0].y;
    val += d[1].x * d[1].x + d[1].y * d[1].y;
    val += EPSILON_S2;

    if(val != 0.0f)
      I_ELEM1(psi_s, i, j, size) =
        0.5f * (1.0f / sqrtf(val));
  }
}
```

And the one to compute Ψ'_D as well:

```
__global__
void psi_d_kernel(
  float* psi_d,
  uint4 size
){
  const int i = threadIdx.y + blockIdx.y * blockDim.y;
  const int j = threadIdx.x + blockIdx.x * blockDim.x;

  float u, v, val;

  if(i < size.y && j < size.x){
    u = tex2DI2(x_ref, j, i, 0);
    v = tex2DI2(x_ref, j, i, 1);
    val  = tex2DI1(jxx_ref, j, i) * u * u;
    val += tex2DI1(jyy_ref, j, i) * v * v;
    val += tex2DI1(jtt_ref, j, i);
    val += 2.0f * tex2DI1(jxy_ref, j, i) * u * v;
    val += 2.0f * tex2DI1(jxt_ref, j, i) * u;
    val += 2.0f * tex2DI1(jyt_ref, j, i) * v;
    val += EPSILON_D2;

    if(val != 0.0f)
      I_ELEM1(psi_d, i, j, size) =
        0.5f * (1.0f / sqrtf(val));
  }
}
```

During the iteration process, after each new solution has been computed, Ψ'_S and Ψ'_D are computed from it. The right hand $\mathbf{f}$ is updated using those two derivatives. The threads in the following iterations

use Ψ'_S and Ψ'_D to compute the matrix coefficients they need.

This solver converges a lot slower than the linear one. Moreover the time spent in each iteration is higher than for the linear one as additional computation is needed.

4.4.3 Multigrid solvers

Nonlinear and linear multigrid solvers are only a set high level functions using the Jacobi or nonlinear Jacobi kernels. At first the number of grid levels to allocate has to be defined. If no maximum depth is given, the application will go as deep as it can. For instance Yosemite sequence can be restricted down to a size of 2×1, figure 4.14 shows the size for each depth of grid and the corresponding grid steps.

depth	size	grid step
0	316×252	1×1
1	158×126	2×2
2	79×63	4×4
3	40×32	4×4
4	20×16	8×8
5	10×8	16×16
6	5×4	32×32
7	3×2	32×64
8	2×1	32×128

Figure 4.14: Yosemite sequence size and grid steps for each depth of a multigrid scheme

Then a restricted version of the motion tensor has to be computed for each grid level. Figure 4.15 shows images of restricted motion tensor component $\mathbf{J_{11}}$ for each depth.

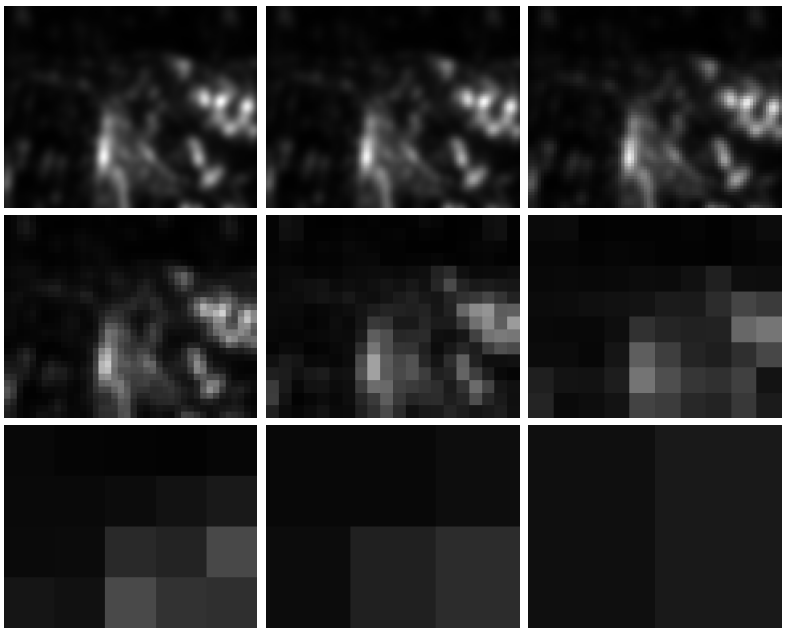


Figure 4.15: Yosemite sequence's J_{11} at each of the 9 multigrid levels from 316×252 to 2×1

The prolongation, restriction have already been explained and the V and W-Cycles are implemented straightforwardly to the pseudo codes given in 2.8. We finally come up with fully working linear and nonlinear solvers that, with the right parameters, compute a dense flow in less than $100ms$.

Chapter 5

Tools developed

All the functionalities described in chapter 4 are bundled in one main command-line application called `flow-compute` described in 5.1. But other tools have also been developed during this project in order to perform repetitive tasks and speed up the development phase. These tools are designed to cooperate and provide distinct and coherent sets of functionalities for our platform. The `flow-tool` command line tool was created to deal with different data representations of the flow. The `flow-studio` gui application was designed for someone to visually interact with the flow parameters and see in real time the flow computed. Finally some wrapper makefiles were created to make the command-line user interaction smoother and to easily compute flow of multiple sequences with predefined parameters. We detail in that chapter the functionalities and behaviour of these tools.

5.1 `flow-compute` command-line tool

The `flow-compute` executable is, together with the flow computation library, the main product of this project. This application was made to compute quickly the optical flow of some image sequences using the GPU and to provide simple and complete tuning on methods and parameters. It implements the algorithms listed in chapter 2. It also implements some extra features as *parameters file handling*, *advanced timing*, *detailed debugging trace* and *video handling* that are detailed here.

5.1.1 Parameter files

Parameters to use for a specific algorithm can differ from one sequence to another. Therefore we thought it could be convenient to store those parameters in a file coming along with the sequence. Using a simple CSV format the `flow-compute` application can be passed a `.params` file as argument to compute the flow with the file parameters. Obviously command-line parameters override parameters read in those files, which adds flexibility to this scheme. Finally the application creates a `.params` file in output. This output file should be kept with the binary flow file in order to be able to find out which parameters were used to compute it. This output file can be used again as input for another computation. The figure 5.1 summarises this behaviour.

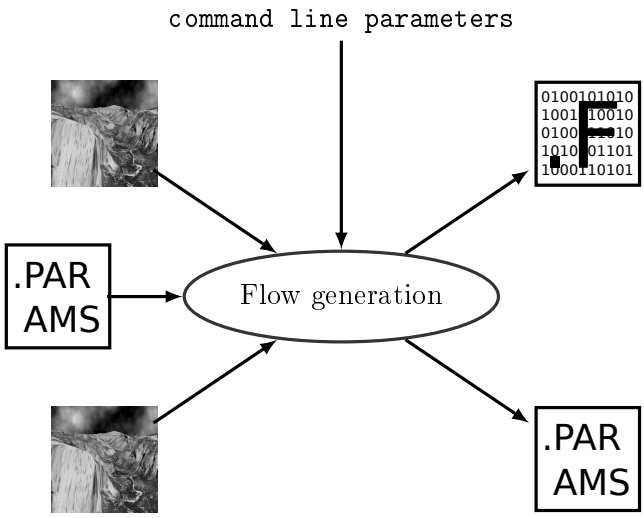


Figure 5.1: `flow-compute`: parameter files

5.1.2 Workflow generation

An image processing algorithm is sometimes hard to debug only by inspecting or printing the values. If the resulting image is wrong, and you do not know why, it can be useful to export one image for each step of computation. Our `flow-compute` application implements this feature generating on demand what we call *workflows*. Figure 5.2 shows a workflow for the Yosemite sequence.

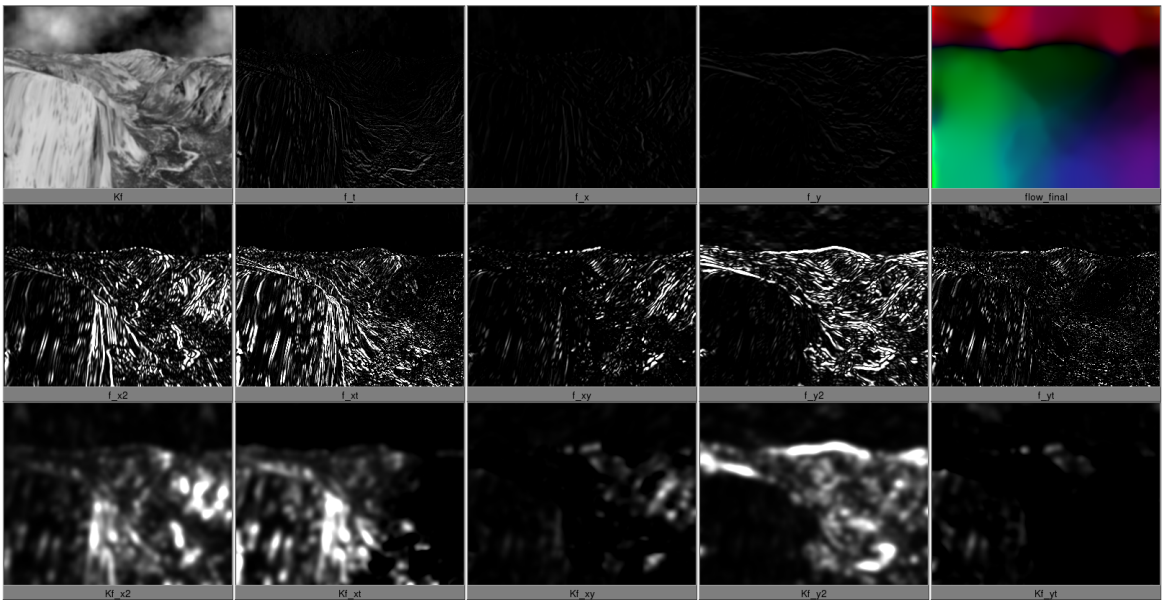


Figure 5.2: `flow-compute`: workflow generation

5.1.3 Advanced timing

Dealing with real-time situations means constantly doing a lot of optimisations. To this end, being able to do precise and flexible time measures in the program is essential. We want to know precisely the time spent in one function but also the frame-rate and the bandwidth in mega-pixels per second. Knowing these parameters for all parts of the algorithm allows to identify the bottlenecks and optimise what has to be. A good timer has to be non-intrusive which means that he should not slow down the part of code it is measuring. It has also to be non code-intrusive which means that it should not be passed as argument of every function.

The `flow-compute` application integrates a timer which is able to provide all these features. We only have to call the `TIMER_INSTACIATE` macro in one file of the project and after all we have to do is include the header in other files and call the timing macros. `TIMER_START("name")` macro starts a measure called *name* and `TIMER_STOP()` stops it.

Using highly recursive function calls, one other feature that is useful in our case is multi-scale timing. Concretely we want to do some measures in a measure, down to a defined depth. It is then interesting to know in a precise function what takes the most of its execution time and what is fast. This useful feature is also implemented in `flow-compute` using the macros `TIMER_NEXT_LEVEL()` and `TIMER_PREV_LEVEL()`.

Time measures are based on CPU runtime with kernel calls. These kind of measures can be slightly different for two similar calls. Also the GPU, while processing data, can be faster as it warms up. To compensate these imprecisions and variations, our timer runs n times the same code and eventually averages the computed values. We give in figures 5.3 and 5.4 some sample execution traces.

Examples

Here is one example of simple timing:

```
TOTAL (100%) : [96.0593 ms, 10.41 fps, 0.83 MPx/s]
  |
  | change format (0%) : [0.423264 ms, 2362.59 fps, 188.14 MPx/s]
  | compute df/dt (0%) : [0.135981 ms, 7353.97 fps, 585.61 MPx/s]
  | gaussian presmoothing (2%) : [2.11773 ms, 472.20 fps, 37.60 MPx/s]
  | derivatives (2%) : [2.13298 ms, 468.83 fps, 37.33 MPx/s]
  | matrix multiplications (0%) : [0.758164 ms, 1318.98 fps, 105.03 MPx/s]
  | convolutions (14%) : [13.4958 ms, 74.10 fps, 5.90 MPx/s]
  | compute flow (80%) : [76.9954 ms, 12.99 fps, 1.03 MPx/s]
```

Figure 5.3: `flow-compute` simple timing

And there you can see an example of multi-scale timing:

```
TOTAL (100%) : [96.1734 ms, 10.40 fps, 0.83 MPx/s]
  |

  [ ... ]

  | compute flow (80%) : [77.1022 ms, 12.97 fps, 1.03 MPx/s]
  |   |
  |   | J restrict (2%) : [1.82903 ms, 546.74 fps, 43.54 MPx/s]
  |   | Matrices (6%) : [5.22847 ms, 191.26 fps, 15.23 MPx/s]
  |   | Vectors (1%) : [1.07363 ms, 931.42 fps, 74.17 MPx/s]
  |   | Solve (89%) : [68.64 ms, 14.57 fps, 1.16 MPx/s]
  |   |   |
  |   |   | Fill 0 (0%) : [0.01464 ms, 68306.01 fps, 5439.34 MPx/s]
  |   |   | FMG (99%) : [68.6234 ms, 14.57 fps, 1.16 MPx/s]
  |   |   |   |
  |   |   |   | VCycle_nl (1%) : [1.1899 ms, 840.41 fps, 66.92 MPx/s]
  |   |   |   |   |
  |   |   |   |   | Jacobi_nl (99%) : [1.18802 ms, 841.74 fps, 67.03 MPx/s]
  |   |   |   |   |
  |   |   |   | Prolongate (0%) : [0.029234 ms, 34206.75 fps, 2723.95 MPx/s]
  |   |   |   | FGM_nl (98%) : [67.4014 ms, 14.84 fps, 1.18 MPx/s]
  |   |   |   |   |
  |   |   |   |   | VCycle_nl (3%) : [2.02254 ms, 494.43 fps, 39.37 MPx/s]
  |   |   |   |   | Prolongate (0%) : [0.02905 ms, 34423.41 fps, 2741.20 MPx/s]
  |   |   |   |   | FGM_nl (96%) : [65.3473 ms, 15.30 fps, 1.22 MPx/s]
```

Figure 5.4: `flow-compute` multi-scale timing

5.1.4 Detailed debugging trace

Multigrid algorithms are somtimes hard to debug or even to use. Choosing good parameters is not always easy and a problem in the computation often results in a 0 flow. Being able for the different kinds of cycle to have a visual information on what happened in definitely a plus. We provide such a feature in the workflow mode of `flow-compute`. The detail of mathematical operations computed is given and the depth is represented with indentation. A debugging trace of a three depth levels full-multigrid solver using V-Cycles is given in appendix B.1 and another one using W-Cycles is given in appendix B.2.

5.1.5 Video handling

Optical flow computation in real-time would not mean anything without video handling. The time of computation – below a certain limit – is important only if the data arrives in the form of a continuous stream with a given frame rate. For the computation to be fast, the `flow-compute` application had to integrate the concept of data streams by for example avoiding data allocation and deallocation inside the computing loop. Memory is allocated once at the beginning and, for a given image size, can then be used to compute many frames. *OpenCV* is used to open the file descriptor `/dev/videoN` of the Linux kernel to be able to retrieve images from a camera connected the computer. To prove it, the figure 5.5 shows the computed flow while the author of this document is moving.

Figure 5.5: Video handling in `flow-compute`

5.2 `flow-tool` command-line tool

As far as we know, two kinds of representation have been used for optical flows: the color one and the vector field one. At a lower level, flow values – different from flow representations – can be stored in different formats in a binary file. Middlebury university uses the `.flo` extension while Otago university uses `.pcm` extension and Yosemite sequence uses `.F` extension. One will need to be able to compute an error measure between a flow computed with our algorithms and for example a `.flo` ground-truth. Finally image sequences can be in several image formats. As a consequence we needed a tool to abstract those various representations and formats. We then implemented `flow-tool`.

Having tried our algorithm firstly using the Yosemite sequence, we use the `.F` flow format as a central format in our application. It is also because this format is really close to the memory layout of the flows we compute, which are themselves really close to the structure imposed by the GPU architecture. Our tool is able to convert from `.pcm` and `.flo` to `.F`. It can compute the average errors – angular or end point – between two `.F` files and it can provide the representation of a `.F` file into an image. The

image representations can be of three kinds: colour representation, colour representation with legend around and vector plot representation. The figure 5.6 summarises the functionalities of `flow-tool`.

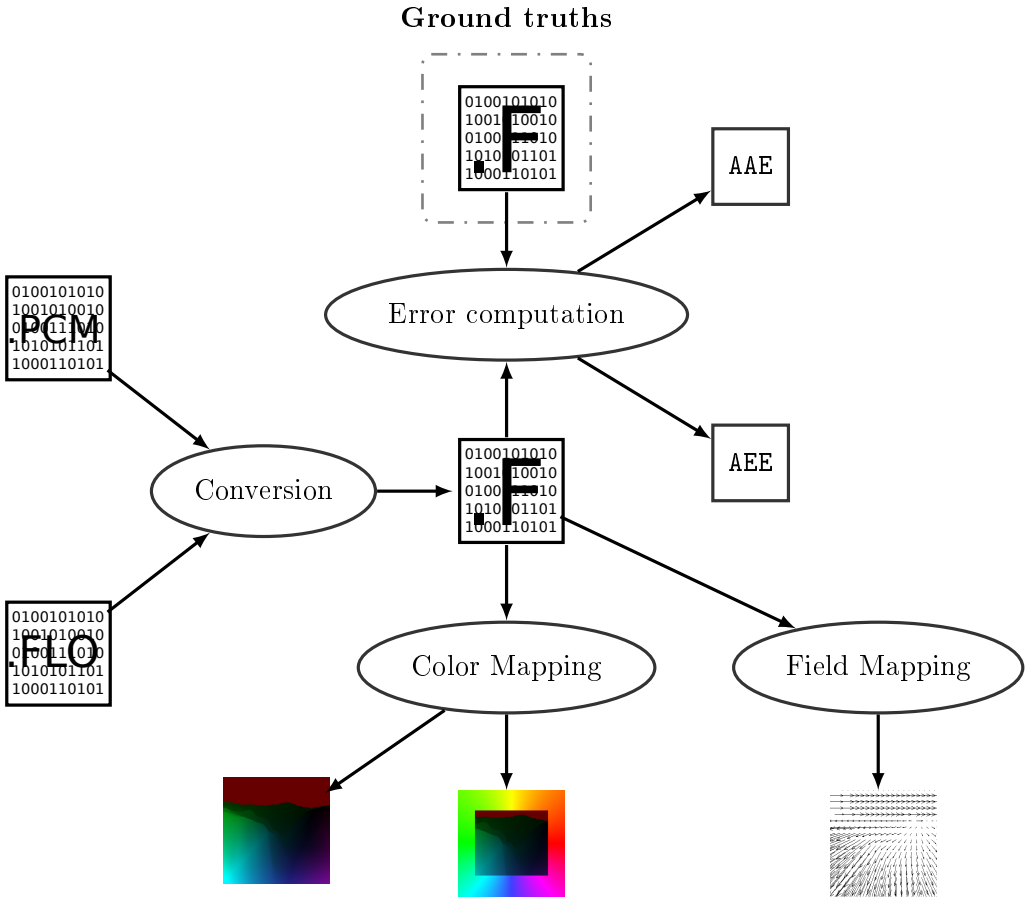


Figure 5.6: `flow-tool` conversion chain flowchart

5.3 Flow generation using `GNU Make`

`GNU Make` is often thought to be destined only to code compilation with dependences. But it is a lot more general than that. It can be used for all tasks that involves interaction between targets and dependencies. We wanted a way to easily test our algorithms on a set of sequences, generate flow images and appropriate montages between them. We then implemented a set of makefiles, using *ImageMagick* command-line tools, to generate these targets letting `GNU Make` handle the dependencies for us. We explain here some use cases of these generation tools.

5.3.1 Sequences

At first we want to have a coherent (in terms of format) set of sequences with ground truth and parameter files. This has to be done preserving the initial files somewhere just in case the central format used does not fit the requirements. In our project tree, the `sequences/` directory was created for that. In the `src/` subdirectory you can put all the sequences you have in every format with ground-truths in also every formats. You can also put default parameter files. Then a simple call to `make`, in

the `sequences/` directory, converts all the sequences in the central format you chose, converts all the ground truths in the `.F` format, generates color and vector field representation for of them and creates symbolic links to your parameter files (or blank files in case there is not one) in the `dst/` directory. Obviously if a given sequence is inserted or another modified, only the necessary (re)computations are done.

5.3.2 Advanced flow generation

Once sequences generated, we want to face them with our algorithms. Then, in the `flow-compute` source code directory (where one is prone to make some changes on the algorithms), a simple calls to `make flows` and `make stats` compute everything possible for each of these sequences:

- binary flow files in `.F` format
- color and vector field representations
- errors (if ground truh) in the form of `.aae` and `.aee` files (figure 5.7)
- montages, mixes and blends between images (figures 5.9, 5.10, 5.11)
- statistics files `.stats` with the detailed timing of the execution (figure 5.8)

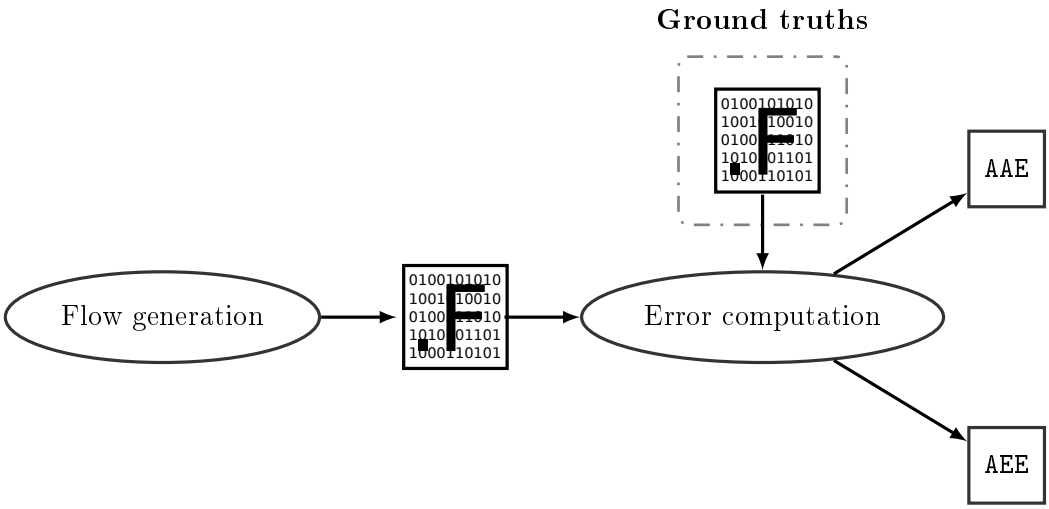


Figure 5.7: `flow-tool`: error computation

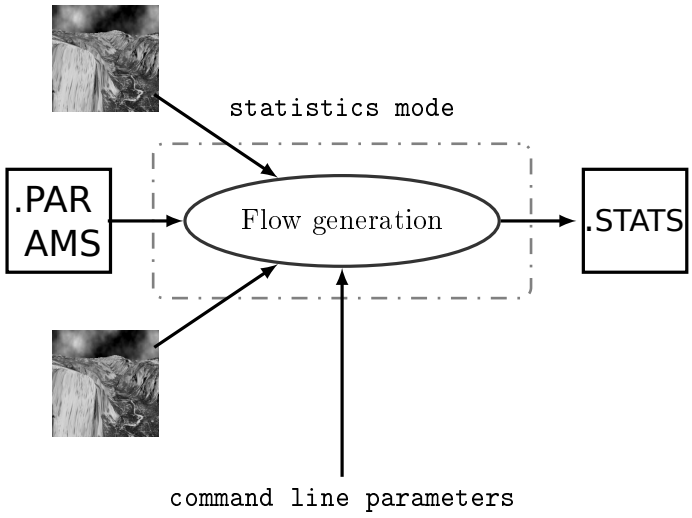


Figure 5.8: `flow-tool`: stats file generation

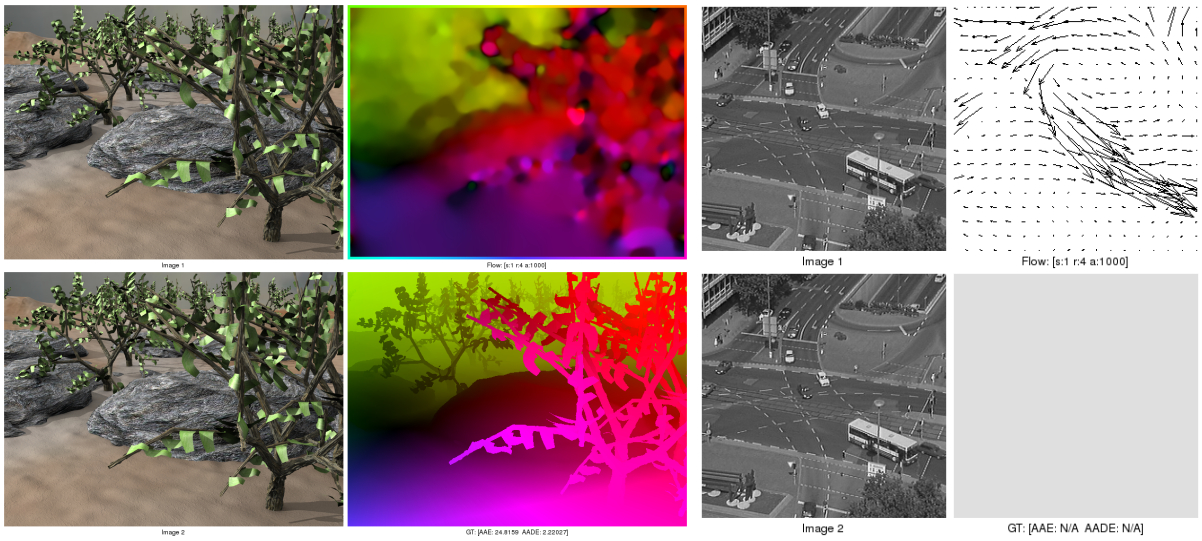


Figure 5.9: Ground truth comparison montages: **(left)** color representation, **(right)** vector field representation

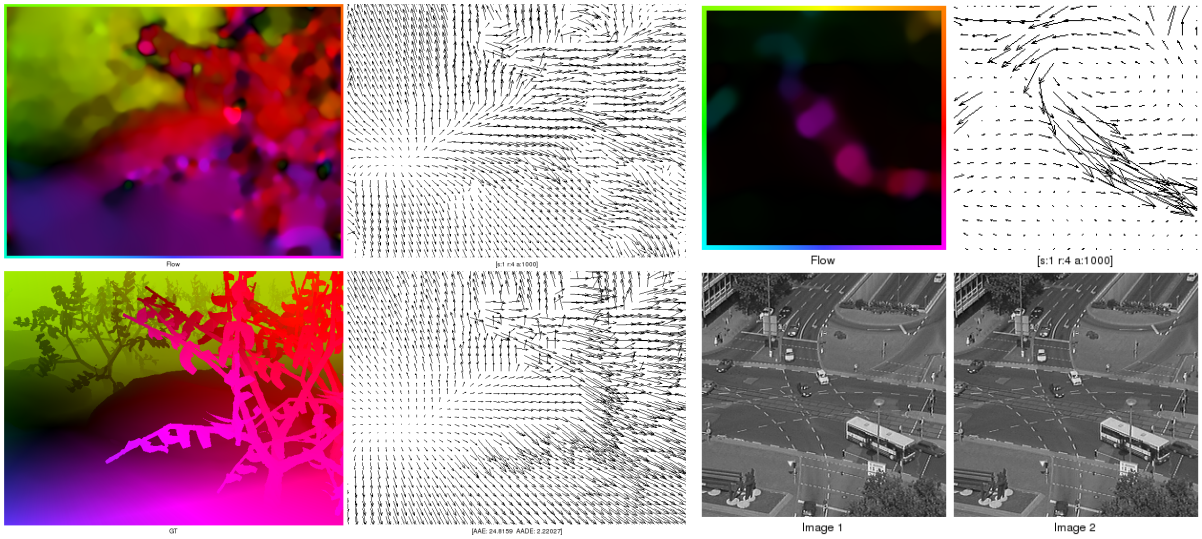


Figure 5.10: Mixes between color and flow field representation: **(left)** ground truth available, **(right)** no ground truth available using sequence images instead

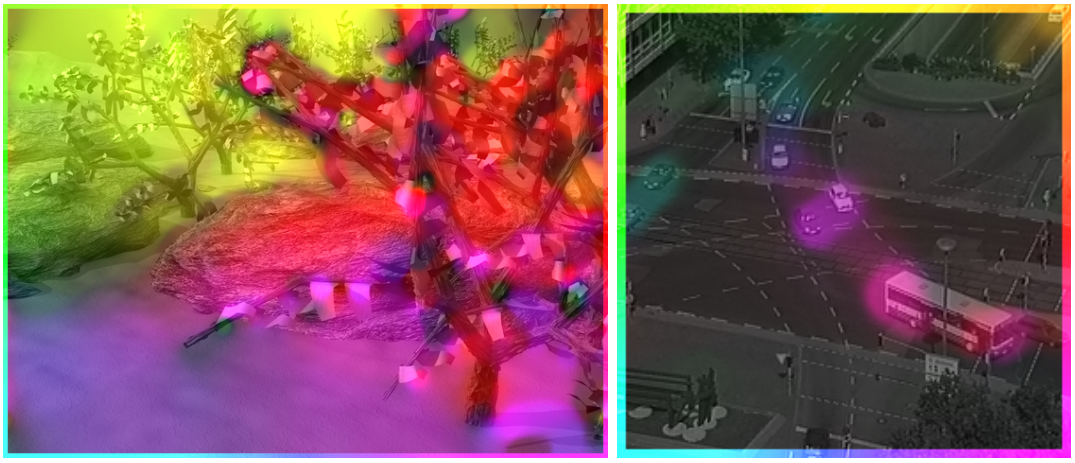


Figure 5.11: Blends between sequence image and color flow representation: **(left)** fixed objects / moving camera, **(right)** fixed camera / moving objects

5.4 `flow-studio` gui application

Setting all the parameters in command-line can be somehow boring and inefficient. Some tasks are repetitively done when researching on optical flow: testing an algorithm with a given sequence trying different parameters, testing many sequences with the same parameters for an algorithm, finding the best parameters for a given sequence... Being able to do that in a graphical user interface would be a great thing. Starting from this idea we implemented from scratch such a user interface using *Qt* as graphical toolkit. Figure 5.12 is a screen-shot of this application that showed during the demo of our project.

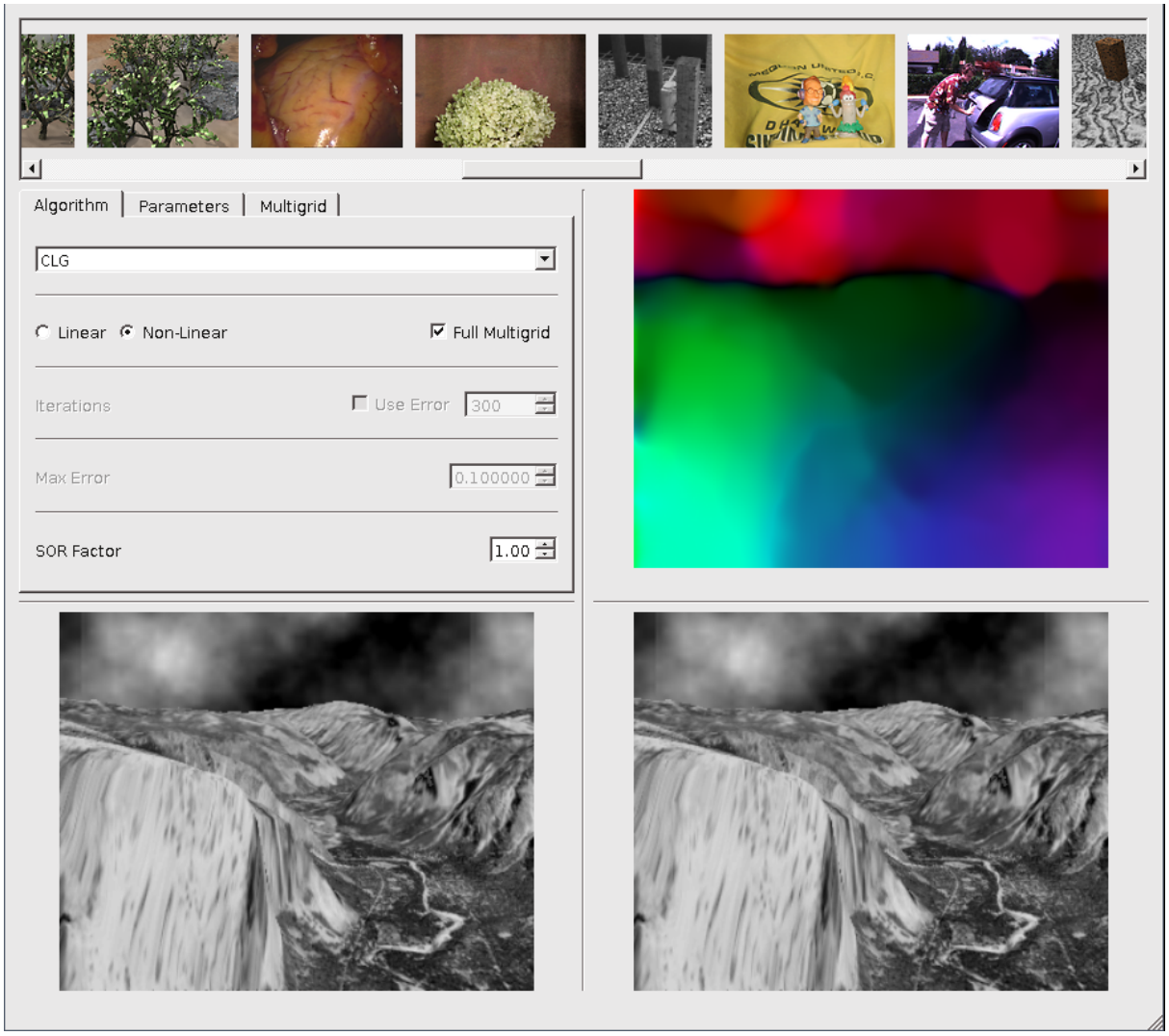


Figure 5.12: `flow-studio`: an interactive gui based flow generation tool

The features implemented are:

- selection of sequence between those generated in the `sequences/` directory
- change of basic parameters such as σ, ρ and α
- selection of the algorithm between those implemented
- enable/disable nonlinearity
- change number of iterations for solvers
- advanced control of the multigrid solver

This is only a proof of concept as it misses some functionalities and is not, for the moment, really stable. As a result the code of this application will not be submitted with the report, nevertheless some further versions can be implemented improving stability and adding more features.

Chapter 6

Results

In this chapter we present our results. We tried to select some representative outputs of our algorithms to give rise to the different aspects of methods implemented. We will discuss both the accuracy and efficiency of implemented methods and try to identify some problems. We use the grey-scale Yosemite sequence with clouds as principal benchmark. The clouds of this sequence form a fully textured region but are hard to analyse as many wrong matchings are possible. The angle of the motion in this region is constant and it is therefore easy to visually find out what is wrong. All the flows computed in 6.1 are using one cycle of full-multigrid solver with two steps of pre and postsmoothing.

6.1 Accuracy

Accuracy is important in optical flow computation as we want to get values as close as possible to the real ones. Despite the fact that accuracy is, in this project, less important than performances we need to comment and criticise the quality of the solutions computed with all implemented algorithms. This has to be done for application users to know which accuracy they can get with the different methods.

6.1.1 *Lucas & Kanade* method

Lucas & Kanade method provides average results for a really simple implementation. In figure 6.1 we present one of the best results in terms of accuracy that our implementation of this algorithm achieved. The parameters are here set to $\sigma = 0.65$ and $\rho = 6.3$.

The flow computed has a poor visual quality and does not really look similar to the ground truth. We can see that too much smoothing is introduced and that there are some important discontinuities in the solution. We can also find some discontinuities in the clouds which should have constant angle. Looking at the legend we can approximatively evaluate the amplitude of the error in this region to $\pm 60°$. The average angular error is $16.44°$ high and the average endpoint error equals 0.81 pixels. This is really high compared to results provided by other researchers. The efficiency fits perfectly real-time applications as we can compute, for that size and that values of the parameters, 38.63 images per second.

6.1.2 *Horn & Schunck* method

Horn & Schunck method introduces global information and is supposed to yield more smooth the solutions. In figure 6.2 we present one of the best results in terms of accuracy that our implementation of this algorithm achieved. The parameters are there set to $\sigma = 0.65$ and $\alpha = 730$.

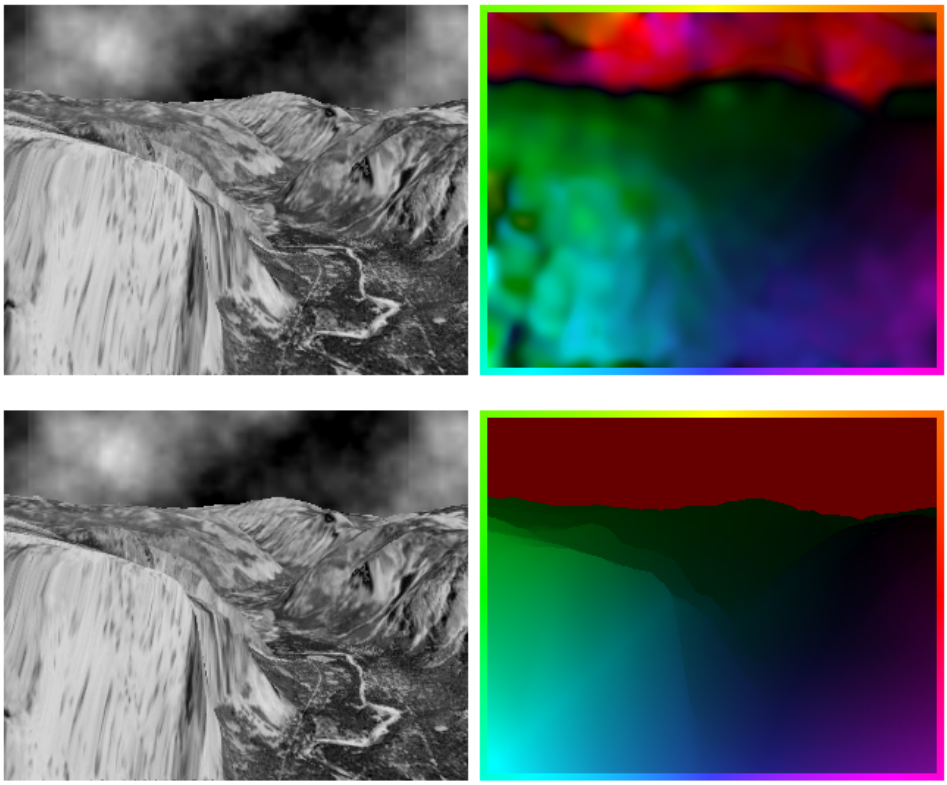


Figure 6.1: *Lucas & Kanade* algorithm result for Yosemite sequence with clouds ($\sigma = 0.65$, $\rho = 6.3$)

The flow computed seems worst than the one computed with the *Lucas & Kanade* method. It is not worst in terms of accuracy though. The noise introduced by this method on the flow is due to the non presmoothed motion tensor that is used. Although, ignoring this noise, the solution seems more smooth and closer to the ground-truth. The transitions more preserved than with the *Lucas & Kanade* method. The clouds are more reliably computed too, as we can approximatively evaluate the amplitude of the error in that region to $\pm 30°$. The average angular error is $15.00°$ high and the average endpoint error equals 0.72 pixels. This a good result compared to *Lucas & Kanade* method but it is still poor compared to other researchers results. The efficiency is twice worst than for the *Lucas & Kanade* method as the computation is done at 22.73 images per second. However it is high enough for most of real-time applications.

6.1.3 *CLG* method

CLG method combines elegantly advantages of both *Lucas & Kanade* and *Horn & Schunck* methods. In figure 6.3 we present one of the best results in terms of accuracy that our implementation of this algorithm achieved. The parameters are set to $\sigma = 0.8$, $\rho = 3.5$ and $\alpha = 470$.

Our implementation of the *CLG* method is, for the linear version, a bit disappointing as it yields worst results as the *Horn & Schunck* 's one. Obviously we are not criticising the *CLG* method here but **our** implementation as this method has provided really good results for other researchers. The solution has less discontinuities than with the *Lucas & Kanade* method. This is due to the global information introduced by the smoothness term as in the *Horn & Schunck* method. The computation

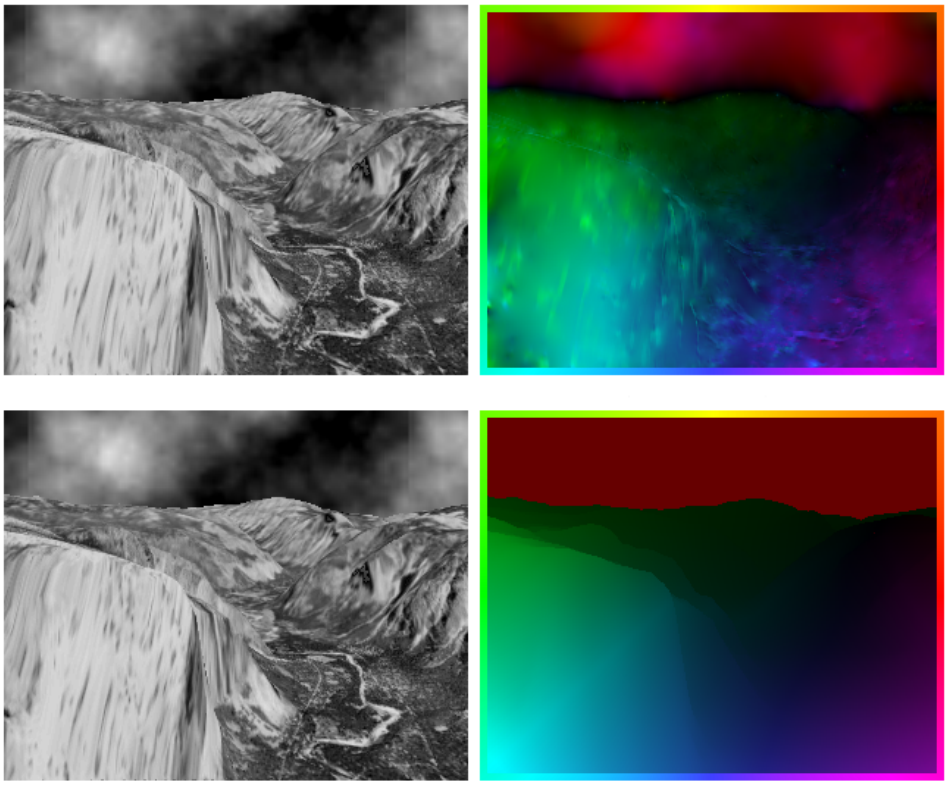


Figure 6.2: *Horn & Schunck* algorithm result for Yosemite sequence with clouds ($\sigma = 0.65$, $\alpha = 730$)

of the clouds region is better than in the *Horn & Schunck* method too. The average angular error is 15.49° high which is better than with the *Lucas & Kanade* method but unfortunately worst than the *Horn & Schunck* 's one. The average endpoint error is 0.74 pixels high. Finally speaking about efficiency the time of computation is the worst seen so far: the computation is done at 17.55 images per second. Here again it fits most of real-time applications.

6.1.4 Nonlinear *CLG* method

Last but not least, the nonlinear *CLG* improves the results recomputing locally the operators over solving iterations. In figure 6.4 we present one of the best results in terms of accuracy achieved by our application during this project. The parameters are there set to $\sigma = 0.7$, $\rho = 1$ and $\alpha = 200$.

The flow computed looks visually really good compared to the previous implementations. We can clearly see the main advantage of nonlinear methods is that they introduce less smoothing. The solution has not much noise and is locally smooth but the discontinuities are well preserved. Figure 6.5 emphasises this phenomenon using another sequence. We can look closer at the edge between the mountain and the clouds. We clearly see that the black region is thin when for other methods it was large because of smoothing. All these visual estimations are confirmed by the average angular error which is 14.53° high and the average endpoint error that is 0.73 pixels high. This is better than all the methods previously introduced and therefore legitimates our implementation of the *CLG* method. However this result is still poor compared to original papers and the error computed could be a lot smaller with some further improvements. Finally the efficiency is the poorest experienced so far as

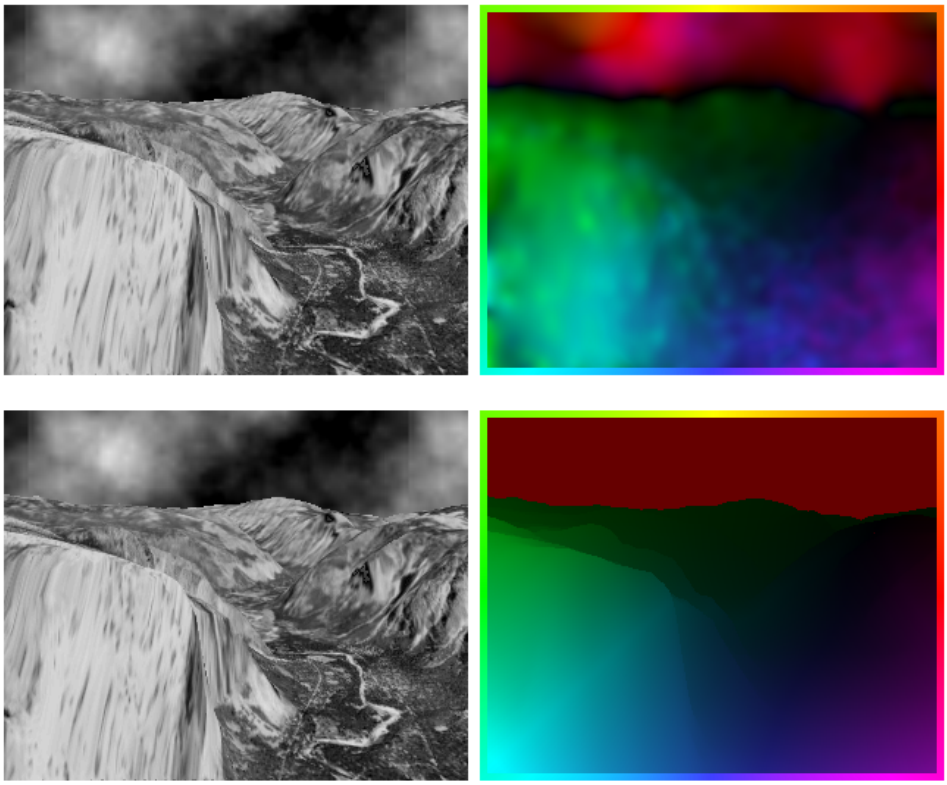


Figure 6.3: *CLG* algorithm result for Yosemite sequence with clouds ($\sigma = 0.8$, $\rho = 3.5$, $\alpha = 470$)

flows are computed at 11.30 images per second. This is normal because the amount of computation to be done at each iteration is higher.

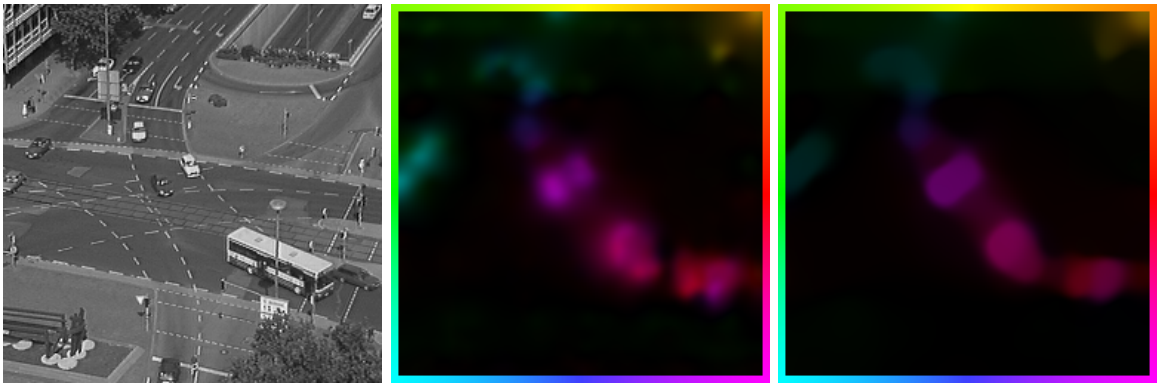


Figure 6.5: Nonlinear methods: discontinuities preserved

6.2 Performances

Good performances are one of the key objectives of our project. We are, for instance, more focused on efficiency than on accuracy to be able to handle real-time situations. Obviously performances inti-

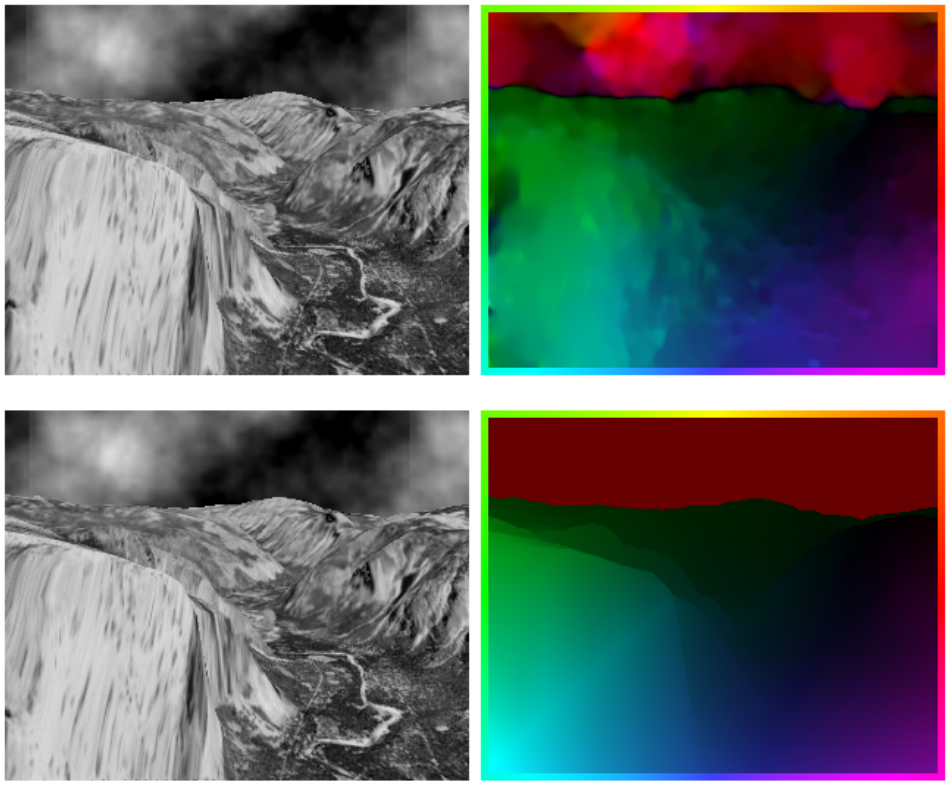


Figure 6.4: Nonlinear *CLG* algorithm result for Yosemite sequence with clouds ($\sigma = 0.8$, $\rho = 3.5$, $\alpha = 470$)

mately depend on the kind of application we target. It is not necessary to compute an over-accurate flow or to compute a flow 10 times faster than what is needed. Moreover the size of images, the number of color channels, the accuracy needed, the tolerance to noise, the need of smoothness and many other factors impact on the choice of one algorithm and its parameters.

6.2.1 Performances of algorithms for a single application

At first we wanted to compare performances of our different algorithms for the same hypothetical application. Using the Yosemite sequence with clouds we benchmarked all of them to rank their performances for similar parameters $\sigma = 0.5$, $\rho = 1.0$ and $\alpha = 1000$. *Lucas & Kanade* method relying more on pre and postsmoothing that other methods, its parameters are different to reach the same target accuracy: $\sigma = 1.0$ and $\rho = 2.0$. The figure 6.6 tabulates the time of computation, frame rate and bandwidth of all implemented algorithms. This figure also focuses on the high amount of time saved by full-multigrid solvers.

	LK	HS		CLG			
				L		NL	
		J(3000)	FMG(2,2)	J(3000)	FMG(2,2)	JNL(3000)	FMG(2,2)
t (ms)	15.4	3421	43.8	3414	50.1	8480	88.6
fr (fps)	65.02	0.29	22.82	0.29	19.95	0.12	11.29
B $(MPx \cdot s^{-1})$	5.18	0.02	1.82	0.02	1.59	0.01	0.90

Figure 6.6: Performances of algorithms for a single application

6.2.2 Performances of each algorithm for different applications

Then we thought it would be interesting to benchmark a given algorithm for different hypothetical applications. Here size and number of color channels to process are varied for the same algorithm. For *Lucas & Kanade* method parameters are set to $\sigma = 0.8$, $\rho = 6.0$). For *Horn & Schunck* , *CLG* and nonlinear *CLG* methods, they are set to $\sigma = 0.8$, $\rho = 3.0$ and $\alpha = 1000$. The results of this benchmark are tabulated in figures 6.7, 6.8 and 6.9.

	Colour			Greyscale		
Size	t(ms)	fr(fps)	B$(MPx \cdot s^{-1})$	t(ms)	fr(fps)	B$(MPx \cdot s^{-1})$
16×16	3.35601	297.97	0.23	1.25434	797.23	0.20
32×32	3.9714	251.80	0.77	1.50192	665.81	0.68
64×64	6.69617	149.34	1.84	2.3989	416.86	1.71
128×128	15.4237	64.84	3.19	5.369	186.25	3.05
256×256	55.729	17.94	3.53	19.0667	52.45	3.44
512×512	217.773	4.59	3.61	73.9292	13.53	3.55

Figure 6.7: Performances of *Lucas & Kanade* algorithm for different applications

	Colour			Greyscale		
Size	t(ms)	fr(fps)	B$(MPx \cdot s^{-1})$	t(ms)	fr(fps)	B$(MPx \cdot s^{-1})$
16×16	7.17457	139.38	0.11	5.42596	184.30	0.05
32×32	9.77188	102.33	0.31	7.57711	131.98	0.14
64×64	14.6655	68.19	0.84	11.3415	88.17	0.36
128×128	26.6134	37.58	1.85	19.4209	51.49	0.84
256×256	70.9636	14.09	2.77	45.2031	22.12	1.45
512×512	240.577	4.16	3.27	140.108	7.14	1.87

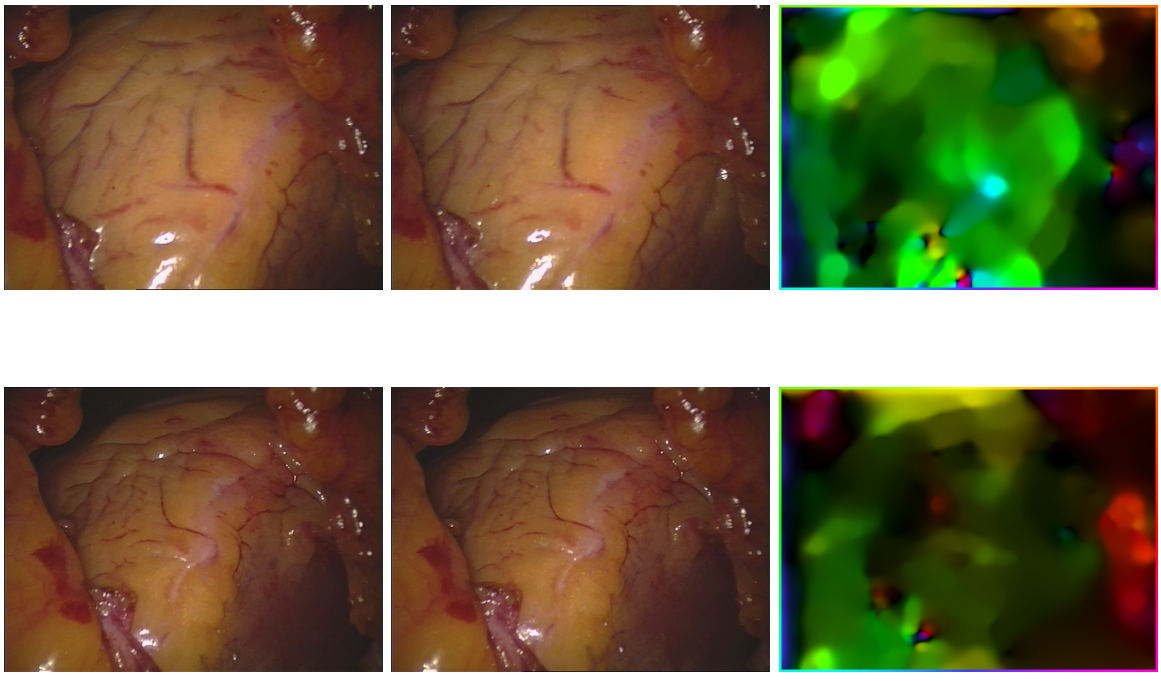
Figure 6.8: Performances of *CLG* algorithm for different applications

	Colour			Greyscale		
Size	t(ms)	fr(fps)	B($MPx \cdot s^{-1}$)	t(ms)	fr(fps)	B($MPx \cdot s^{-1}$)
16×16	13.4829	74.17	0.06	11.6441	85.88	0.02
32×32	18.7918	53.21	0.16	16.7841	59.58	0.06
64×64	27.3959	36.50	0.45	23.9993	41.67	0.17
128×128	45.1346	22.16	1.09	37.8763	26.40	0.43
256×256	101.498	9.85	1.94	75.6003	13.23	0.87
512×512	307.492	3.25	2.56	206.889	4.83	1.27

Figure 6.9: Performances of nonlinear *CLG* algorithm for different applications

6.3 Application

Our target application is biomedical images flow computation. The choice of the algorithms to implement was made considering this target application and the assumptions injected in equations too. To legitimate these choices, we tried our best algorithm (nonlinear *CLG* method) on a medical sequence. We can see in this sequence a heart moving inside the chest of a person as the blood arrives and leaves. This sequence is really hard to handle because the wet surface introduces specularities. Moreover only the veins provide information (i.e. gradient) on this surface. The motion is hard to compute on these non-textured locations. The motion is also special, sometimes we can see the heart rotating, shrinking or growing. Nevertheless all these problems can be managed by the nonlinear *CLG* algorithm. The figure 6.10 shows that the flow computed on the heart surface – ignoring the borders – has not a lot of discontinuities.



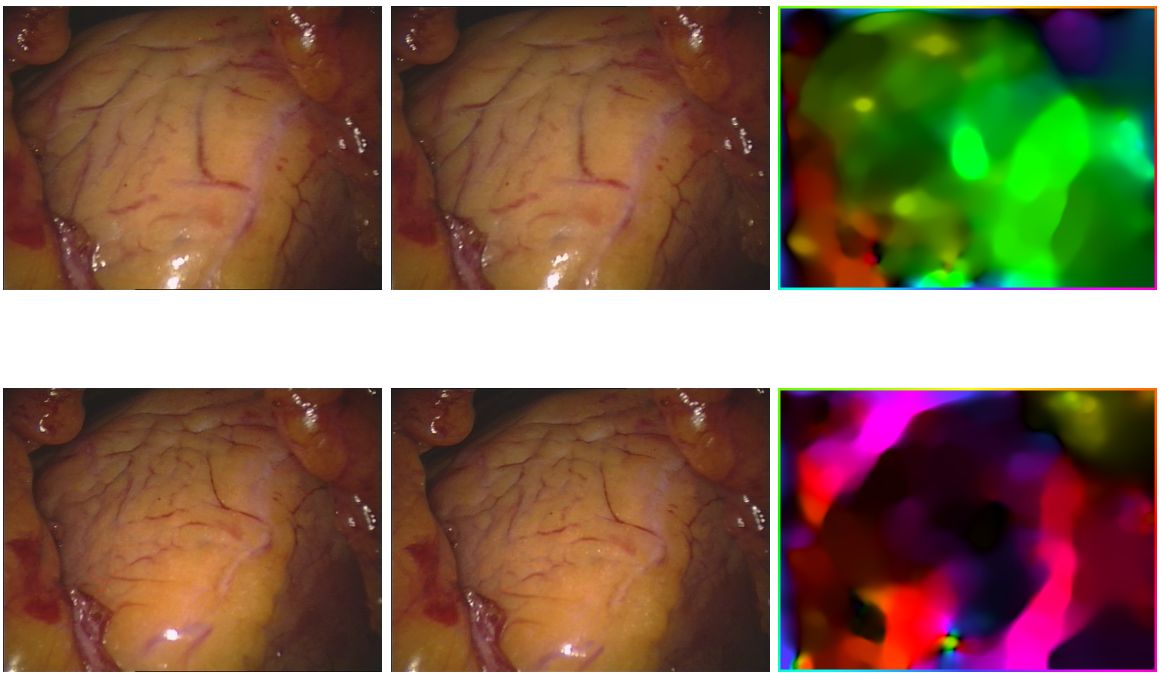


Figure 6.10: Example of biomedical images optical flow computation: heart sequence

Conclusion

At first we can discuss the objectives of this project. Have we met them ? Our library for real-time optical flow computation gives good results in terms of efficiency – between 10 and more than 60 frames per second depending on the algorithm used – and average results in terms of accuracy – 14.53° of average angular error at best for Yosemite sequence.

On top of this library are built `flow-compute` and `flow-tool` – fully working tools – that are combined in makefiles wrappers for an automatic flow generation accessible by someone with a small computing background. Finally the `flow-studio` prototype application gives an insight of a graphical user interface for flow computation.

This set of tools form an advanced prototype of platform for real-time optical flow computation as we wanted to develop. Obviously all the software needed to use these tools is free and open-source: `gcc`, `GNU Make`, *ImageMagick* and *Qt* which is not open-source but free to use for non-commercial software. This prototype, with some further work, could then be released as free software destined to the scientific community. We can then consider that the objectives we targeted are met. However we formulate in this conclusion some potential improvements to be made.

The main problem, compared to other publications, is definitely accuracy. Many things could be done to improve average errors of computed solutions. The fact that *Horn & Schunck* method yields better results than the linear *CLG* method gives us a clue on what could be improved. Both methods are only different for the postsmoothing step. We think that the reason why the *Horn & Schunck* method has better results than the other one is that our convolution kernels introduce error. But how could convolution introduce error ? All is about side effects and borders processing...

To be able to achieve best performances we use the *CUDA* default clamped behaviour for out-of-range values. This means that convolutions at the borders typically act as if the border values were infinitely repeated. More advanced convolution algorithms, notably the one implemented in *OpenCV* library, use reflection schemes at the borders such as *border 101*. Moreover, as the same problem occurs in prolongation and restriction operators of full-multigrid solvers, such an improvement would also increase their quality. Finally, the entire flow computation for all algorithms involving a lot of smoothing operations, we can assume that the error generated at the borders spreads widely over the whole computed flow. We could then expect significantly better results in terms of accuracy implementing time-efficient reflection schemes.

The constraints chosen for the energy functionals and the assumptions made one the flow are only basic ones. Other publications use more advanced constraints. Flow driven anisotropic regularisers[9], theory of warping[5] and many other theoretical developments around optical flow could be introduced in our algorithms to reach higher accuracies. Optical flow being an active research field, new methods of computation and theoretical models are constantly developed. Methods such as Brox et al. method[5], complementary optical flow, anisotropic Huber-L1, TV-L1 and many others can provide

highly accurate results with sometimes less than 2° average angular error for the Yosemite sequence. Thus implementing new methods could also impact positively on our results.

Talking about efficiency, results could be improved by doing some further optimisations on the *CUDA* kernels developed. Nonlinear full-multigrid solver and, part of it, Jacobi solver with non-lagged linearities were implemented at the very end of the project. They provide average performances but could easily be optimised with simple code transformations.

The GPU used for our experiments is really basic and rather outdated. This is a good thing as all scientists do not dispose of a brand new GPU with high bandwidth memory and a lot of stream processors. It would be sad, in the case of a tool to help the scientific community, to experience good performances only on an expensive GPU. But the good results we obtained in terms of efficiency could be increased tenfold on a GPU such as the one on the *nvidia GeForce GTX 480* graphic card. We are looking forward to try our library on such a powerful GPU are curious to see by how many it improves the results.

To conclude on this project we can say that we do hope some further work will be done on our prototype to maybe see someday `flow-platform` available for the scientific community...

Appendix A

Notations and legends

A.1 Operators

Let us consider the following matrix:

$$A = \begin{pmatrix} A_{0,0} & \dots & A_{0,M} \\ \vdots & \ddots & \vdots \\ A_{N,0} & \dots & A_{N,M} \end{pmatrix} \tag{A.1}$$

The **flat** operator converts a matrix in a vector using a row-major layout:

$$\mathbf{flat}A = (A_{0,0}, \quad A_{0,1}, \quad \dots, \quad A_{N,M-1}, \quad A_{N,M})^{\mathsf{T}} \tag{A.2}$$

The pointwise product between two vectors is defined as:

$$\mathbf{c} = \mathbf{a} \bullet \mathbf{b} \Leftrightarrow \forall i \in [1..N] \quad c_i = a_i b_i \tag{A.3}$$

The pointwise product between two matrices is defined as:

$$C = A \bullet B \Leftrightarrow \forall (i,j) \in [1..N] \times [1..M] \quad C_{i,j} = A_{i,j} B_{i,j} \tag{A.4}$$

A.2 Motion tensor

We represent the motion tensor J that holds the information of the first order derivatives in a tensor structure:

$$J = \begin{pmatrix} \mathbf{J_{11}} & \mathbf{J_{12}} & \mathbf{J_{13}} \\ \mathbf{J_{21}} & \mathbf{J_{22}} & \mathbf{J_{23}} \\ \mathbf{J_{31}} & \mathbf{J_{32}} & \mathbf{J_{33}} \end{pmatrix} \tag{A.5}$$

Introducing $u_1 = x$, $u_2 = y$ and $u_3 = z$, each component is a vector such as:

$$\forall (i,j) \in \{1,2,3\}^2 \quad \mathbf{J_{ij}} \quad = \quad \mathbf{flat} \left[\frac{\partial \mathbf{f}}{\partial u_i} \bullet \frac{\partial \mathbf{f}}{\partial u_j} \right] \tag{A.6}$$

A.3 Flow vectors

Most of times we will denote the flow as a vector of solutions according to the x- and y- directions:

$$\mathbf{u} = (u, v)^{\mathsf{T}} \tag{A.7}$$

The component u and v a actually matrices as, for instance, we call the x- value at the (i, j) location $u_{i,j}$.

In some other calculus, specifically the solvers calculus, it is more convenient to represent flows as one big linear vector defined as:

$$\mathbf{F} = \begin{pmatrix} \mathbf{flat}F_x \\ \mathbf{flat}F_y \end{pmatrix} \tag{A.8}$$

A.4 First order derivatives

First order derivatives of an image pair are computed with fourth order precision as:

$$[f_x]_{i,j} = \frac{1}{2}\sum_{n=0}^{1}\left(\frac{f_{i,j-2,t+n} - 8f_{i,j-1,t+n} + 8f_{i,j+1,t+n} - f_{i,j+2,t+n}}{12h_x}\right) \tag{A.9}$$

$$[f_y]_{i,j} = \frac{1}{2}\sum_{n=0}^{1}\left(\frac{f_{i-2,j,t+n} - 8f_{i-1,j,t+n} + 8f_{i+1,j,t+n} - f_{i+2,j,t+n}}{12h_y}\right) \tag{A.10}$$

$$[f_t]_{i,j} = \frac{f_{i,j,t+1} - f_{i,j,t}}{h_t} \tag{A.11}$$

A.5 Laplacian operator

The laplacian operator Δ can be discretised as a weighted sum of the neighbours of a point. For instance, h being the step size of a grid:

$$\Delta f_{i,j} \approx \frac{f_{i,j-1} + f_{i,j+1} - 2f_{i,j}}{h_x^2} + \frac{f_{i-1,j} + f_{i+1,j} - 2f_{i,j}}{h_y^2} \tag{A.12}$$

The laplacian operator for a row-major image vector is given by:

$$L_{2\times 2} = \left(\begin{array}{cc|cc} 2 & -1 & -1 & 0 \\ -1 & 2 & 0 & -1 \\ \hline -1 & 0 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{array}\right), \quad L_{3\times 3} = \left(\begin{array}{ccc|ccc|ccc} 2 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 3 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & -1 \\ \hline 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{array}\right), \quad \ldots \tag{A.13}$$

A.6 Color flow representation legend

In this document we represent flows with either a field of vectors or a color code. The legend of the color code is given in figure A.1.

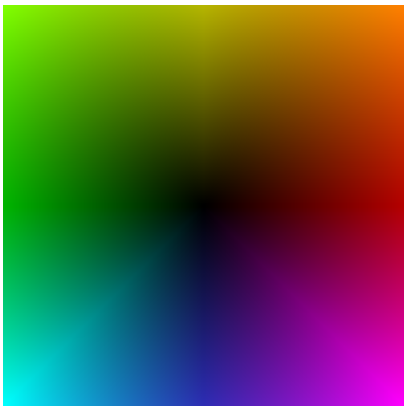


Figure A.1: Color flow representation legend: magnitude as intensity of color and angle as color

Appendix B

Full-multigrid solvers debugging

This appendix present some debugging traces of our full-multigrid solvers algorithms.

B.1 V-Cycles

```
./flow-compute -g -i -f -d2 -v0 -q -w ../sequences/dst/yos.0.png ../sequences/dst/yos.1.png

    {depth: 2, cycle: 1/1}

    S_NL[depth: 2, n: 10] | A2 x2 = f2 --- bottom
    -> [0] 1.000000e+00
    -> [9] 1.092513e-01

         //
        //
       //   [depth: 2] -> [depth: 1]   |   x1 = P[2->1] x2
      //
    \\ //
     \V/

  {depth: 1, cycle: 1/1}

  V_NL[presmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 2.724886e-02
  -> [1] 2.264541e-02
       \
        \  r1 = f1 - A1 x1
         \  f2 = R[1->2] r1 + A1( R[1->2] x2 )
         _\/
    S_NL[depth: 2, n: 10] | A2 x2 = f2 --- bottom
    -> [0] 9.913190e-02
    -> [9] 6.526698e-02
         /
        /    e1 = P[2->1] ( x2 - R[1->2] x1 ), |e1| = 0.000000e+00
      \/_
  V_NL[postsmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 1.638443e-02
```

```
  -> [1] 1.148990e-02

       //
      //
     //   [depth: 1] -> [depth: 0]   |   x0 = P[1->0] x1
    //
  \\ //
   \V/

{depth: 0, cycle: 1/1}

V_NL[presmooth, depth: 0, n: 2]   |   A0 x0 = f0
-> [0] 8.469992e-03
-> [1] 4.275628e-03
     \
      \  r0 = f0 - A0 x0
       \  f1 = R[0->1] r0 + A0( R[0->1] x1 )
       _\/
  V_NL[presmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 1.951636e-02
  -> [1] 1.189268e-02
       \
        \  r1 = f1 - A1 x1
         \  f2 = R[1->2] r1 + A1( R[1->2] x2 )
         _\/
    S_NL[depth: 2, n: 10] | A2 x2 = f2 --- bottom
    -> [0] 6.634983e-02
    -> [9] 5.703199e-02
         /
        /    e1 = P[2->1] ( x2 - R[1->2] x1 ), |e1| = 1.278900e-01
      \/_
  V_NL[postsmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 1.436005e-02
  -> [1] 8.303368e-03
       /
      /    e0 = P[1->0] ( x1 - R[0->1] x0 ), |e0| = 0.000000e+00
    \/_
V_NL[postsmooth, depth: 0, n: 2]   |   A0 x0 = f0
-> [0] 8.549634e-03
-> [1] 4.519083e-03
```

B.2 W-Cycles

```
./flow-compute -g -i -f -d2 -v1 -q -w ../sequences/dst/yos.0.png ../sequences/dst/yos.1.png

    {depth: 2, cycle: 1/1}

    S_NL[depth: 2, n: 10] | A2 x2 = f2 --- bottom
    -> [0] 1.000000e+00
    -> [9] 1.092513e-01
```

```
         //
        //
       //   [depth: 2] -> [depth: 1]   |   x1 = P[2->1] x2
      //
    \\ //
     \V/

  {depth: 1, cycle: 1/1}

  W_NL[presmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 2.724886e-02
  -> [1] 2.264541e-02
       \
        \  r1 = f1 - A1 x1
         \  f2 = R[1->2] r1 + A1( R[1->2] x2 )
         _\/
    S_NL[depth: 2, n: 10] | A2 x2 = f2 --- bottom
    -> [0] 9.913190e-02
    -> [9] 6.526698e-02
         /
        /    e1 = P[2->1] ( x2 - R[1->2] x1 ), |e1| = 0.000000e+00
      \/_
  W_NL[postsmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 1.638443e-02
  -> [1] 1.148990e-02

       //
      //
     //   [depth: 1] -> [depth: 0]   |   x0 = P[1->0] x1
    //
  \\ //
   \V/

{depth: 0, cycle: 1/1}

W_NL[presmooth, depth: 0, n: 2]   |   A0 x0 = f0
-> [0] 8.469992e-03
-> [1] 4.275628e-03
     \
      \  r0 = f0 - A0 x0
       \  f1 = R[0->1] r0 + A0( R[0->1] x1 )
       _\/
  W_NL[presmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 1.951636e-02
  -> [1] 1.189268e-02
       \
        \  r1 = f1 - A1 x1
         \  f2 = R[1->2] r1 + A1( R[1->2] x2 )
         _\/
```

```
    S_NL[depth: 2, n: 10] | A2 x2 = f2 --- bottom
    -> [0] 6.634983e-02
    -> [9] 5.703199e-02
         /
        /    e1 = P[2->1] ( x2 - R[1->2] x1 ), |e1| = 1.278900e-01
      \/_
  W_NL[postsmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 1.436005e-02
  -> [1] 8.303368e-03
  W_NL[presmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 7.741360e-03
  -> [1] 7.532462e-03
      \
       \  r1 = f1 - A1 x1
        \  f2 = R[1->2] r1 + A1( R[1->2] x2 )
         _\/
    S_NL[depth: 2, n: 10] | A2 x2 = f2 --- bottom
    -> [0] 6.093034e-02
    -> [9] 6.049969e-02
         /
        /    e1 = P[2->1] ( x2 - R[1->2] x1 ), |e1| = 1.918802e-01
      \/_
  W_NL[postsmooth, depth: 1, n: 2]   |   A1 x1 = f1
  -> [0] 1.534350e-02
  -> [1] 7.971475e-03
      /
     /    e0 = P[1->0] ( x1 - R[0->1] x0 ), |e0| = 0.000000e+00
    \/_
W_NL[postsmooth, depth: 0, n: 2]   |   A0 x0 = f0
-> [0] 1.053220e-02
-> [1] 5.414092e-03
```

Appendix C

Sequences credits

We acknowledge in this appendix the authors of the sequences used to write this report.

C.1 Yosemite sequences

We use the *Yosemite sequence without clouds* in chapter 2. Also in most of our flow figures we use the *Yosemite sequence with clouds*. These sequences were originally created by Lynn Quam who we credit for that.

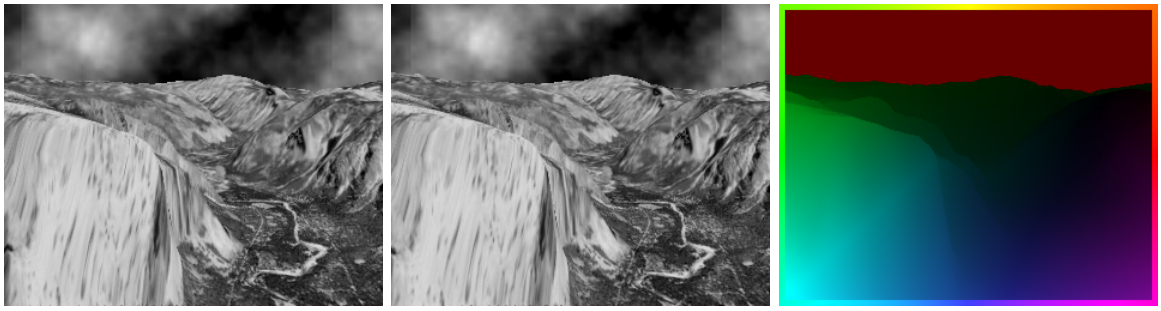


Figure C.1: Yosemite sequence with clouds [Credits: Lynn Quam]

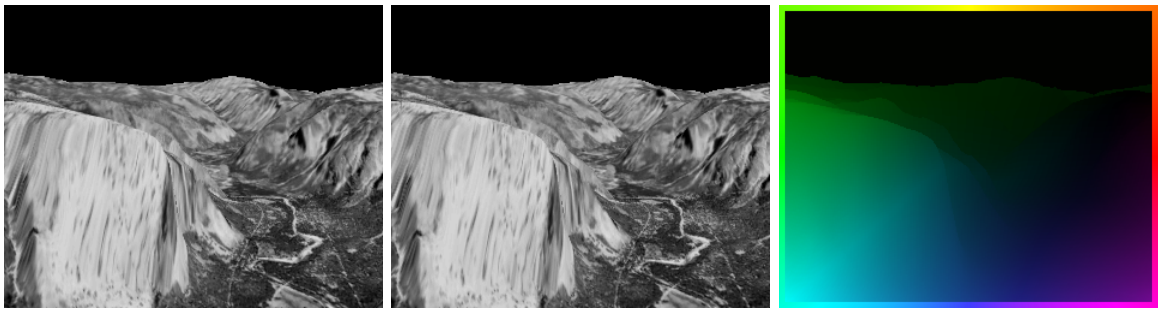


Figure C.2: Yosemite sequence without clouds [Credits: Lynn Quam]

C.2 Urban3 sequence

We use in chapter 1 the *Urban3 sequence*. This sequence is provided by Middleburry university as part of the database and evaluation methodology for optical flow `http://vision.middlebury.edu/flow/`.

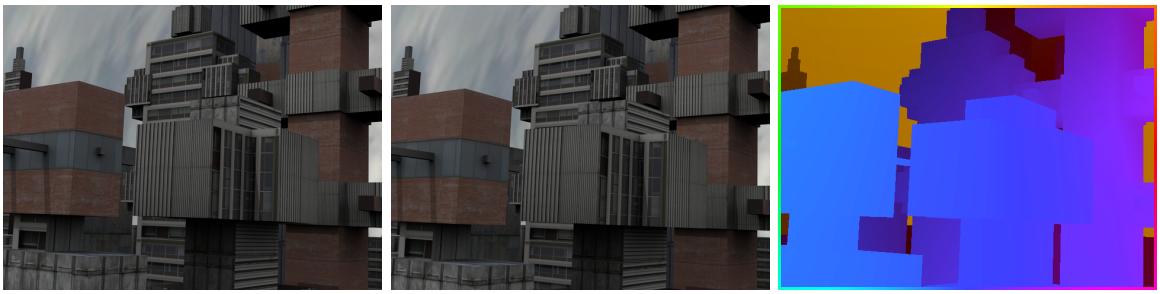


Figure C.3: Urban3 sequence [Credits: Middleburry university]

C.3 Army sequence

We use in chapter 4 the *Army sequence*. This sequence is provided by Middleburry university as part of the database and evaluation methodology for optical flow `http://vision.middlebury.edu/flow/`.

Figure C.4: Army sequence [Credits: Middleburry university]

C.4 Flowerpots sequence

The *Flowerpots sequence* is the stereovision sequence used in chapter 1. This sequence is provided by Middleburry university as part of the database and evaluation methodology for stereovision `http://vision.middlebury.edu/stereo/`.

Figure C.5: Flowerpots stereovision sequence [Credits: Middleburry university]

C.5 Ettlinger-Tor sequence

Ettlinger-Tor sequence, used in chapters 1 and 6, is a traffic intersection sequence recorded at the Ettlinger-Tor in Karlsruhe by a stationary camera. This sequence can be downloaded on H.-H. Nagel's image sequence server at `http://i21www.ira.uka.de/image_sequences/`.

Figure C.6: Ettlinger-Tor sequence [Credits: H.-H Nagel]

Bibliography

[1] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos. Comparing cuda and opengl implementations for a jacobi iteration. In *High Performance Computing Simulation, 2009. HPCS '09. International Conference on*, pages 22 –32, 21-24 2009.

[2] Simon Baker, Stefan Roth, Daniel Scharstein, Michael J. Black, J.P. Lewis, and Richard Szeliski. A database and evaluation methodology for optical flow. *Computer Vision, IEEE International Conference on*, 0:1–8, 2007.

[3] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[4] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[5] Thomas Brox, Andrés Bruhn, Nils Papenberg, and Joachim Weickert. High accuracy optical flow estimation based on a theory for warping. pages 25–36. Springer, 2004.

[6] A. Bruhn, J. Weickert, C. Feddern, T. Kohlberger, and C. Schnörr. Variational optic flow computation in real-time. *IEEE Trans. Image Proc.*, 14(5):608–615, 2005.

[7] Andres Bruhn and Joachim Weickert. Towards ultimate motion estimation: Combining highest accuracy with real-time performance. In *ICCV '05: Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, pages 749–755, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Andrés Bruhn, Joachim Weickert, Timo Kohlberger, and Christoph Schnörr. A multigrid platform for real-time motion computation with discontinuity-preserving variational methods. *Int. J. Comput. Vision*, 70(3):257–277, 2006.

[9] Andrés Bruhn. *Variational Optic Flow Computation - Accurate Modelling and Efficient Numerics*. PhD thesis, Universität des Saarlandes, 2006.

[10] Andrés Bruhn and Joachim Weickert. A multigrid platform for real-time motion computation with discontinuity-preserving variational methods. *International Journal of Computer Vision*, 70:257–277, 2006.

[11] Andrés Bruhn, Joachim Weickert, Timo Kohlberger, and Christoph Schnörr. Discontinuity-preserving computation of variational optic flow in real-time. In *Scale-Space and PDE Methods in Computer Vision, volume 3459 of Lecture Notes in Computer Science*, pages 279–290. Springer, 2005.

[12] Andrés Bruhn, Joachim Weickert, and Christoph Schnörr. Combining the advantages of local and global optic flow methods. In *In Pattern Recognition, L.Van Gool, (Ed)*, pages 454–462. Springer, 2002.

[13] Martin Burger. Review and description of "computational methods for inverse problems", by curtis r. vogel. siam, philadelphia, pa, 2002. *Math. Comput.*, 72(243):1574–1575, 2003.

[14] Tony F. Chan and Pep Mulet. On the convergence of the lagged diffusivity fixed point method in total variation image restoration. *SIAM J. Numer. Anal.*, 36(2):354–367, 1999.

[15] Frohn-Schauf, Claudia, Henn, Stefan, Witsch, and Kristian. Nonlinear multigrid methods for total variation image denoising. *Computing and Visualization in Science*, 7(3-4):199–206, October 2004.

[16] Pascal Gwosdek, Andrés Bruhn, and Joachim Weickert. High performance parallel optical flow algorithms on the sony playstation 3. *Vision, Modeling and Visualisation*, 2008.

[17] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. Technical report, Cambridge, MA, USA, 1980.

[18] 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*, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[19] NVIDIA. Cuda best practises guide. Technical report, NVIDIA Corporation, 2010.

[20] NVIDIA. Cuda programming guide. Technical report, NVIDIA Corporation, 2010.