Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Optical Flow Computation on Compute Unified Device Architecture

Examensarbete utfört i Bildbehandling
vid Tekniska högskolan i Linköping
av

Erik Ringaby

LiTH-ISY-EX--08/4043--SE

Linköping 2008

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Optical Flow Computation on Compute Unified Device Architecture

Examensarbete utfört i Bildbehandling
vid Tekniska högskolan i Linköping
av

Erik Ringaby

LiTH-ISY-EX--08/4043--SE

Handledare: **Johan Hedborg**
ISY, Linköpings universitet

Examinator: **Michael Felsberg**
ISY, Linköpings universitet

Linköping, 6 November, 2008

Avdelning, Institution Division, Department Computer Vision Laboratory Department of Electrical Engineering Linköpings universitet SE-581 83 Linköping, Sweden	**Datum** Date 2008-11-06

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ ____________	**Rapporttyp** Report category ☐ Licentiatavhandling ☒ Examensarbete ☐ C-uppsats ☐ D-uppsats ☐ Övrig rapport ☐ ____________	**ISBN** — **ISRN** LiTH-ISY-EX--08/4043--SE **Serietitel och serienummer** Title of series, numbering **ISSN** —

URL för elektronisk version

http://www.cvl.isy.liu.se
http://www.ep.liu.se/

Titel Title	Optiskt flödeberäkning med CUDA Optical Flow Computation on Compute Unified Device Architecture
Författare Author	Erik Ringaby

Sammanfattning
Abstract

There has been a rapid progress of the graphics processor the last years, much because of the demands from computer games on speed and image quality. Because of the graphics processor's special architecture it is much faster at solving parallel problems than the normal processor. Due to its increasing programmability it is possible to use it for other tasks than it was originally designed for.

Even though graphics processors have been programmable for some time, it has been quite difficult to learn how to use them. CUDA enables the programmer to use C-code, with a few extensions, to program NVIDIA's graphics processor and completely skip the traditional programming models. This thesis investigates if the graphics processor can be used for calculations without knowledge of how the hardware mechanisms work. An image processing algorithm calculating the optical flow has been implemented. The result shows that it is rather easy to implement programs using CUDA, but some knowledge of how the graphics processor works is required to achieve high performance.

Nyckelord
Keywords optical flow, GPU, GPGPU, CUDA

Abstract

There has been a rapid progress of the graphics processor the last years, much because of the demands from computer games on speed and image quality. Because of the graphics processor's special architecture it is much faster at solving parallel problems than the normal processor. Due to its increasing programmability it is possible to use it for other tasks than it was originally designed for.

Even though graphics processors have been programmable for some time, it has been quite difficult to learn how to use them. CUDA enables the programmer to use C-code, with a few extensions, to program NVIDIA's graphics processor and completely skip the traditional programming models. This thesis investigates if the graphics processor can be used for calculations without knowledge of how the hardware mechanisms work. An image processing algorithm calculating the optical flow has been implemented. The result shows that it is rather easy to implement programs using CUDA, but some knowledge of how the graphics processor works is required to achieve high performance.

Sammanfattning

Grafikprocessorn har utvecklats mycket de senaste åren, bland annat tack vare högre krav på bildkvalité och hastighet hos datorspel. Grafikprocessorns speciella arkitektur gör att den är mycket snabbare än den vanliga processorn på att lösa parallella problem med mycket data. Eftersom grafikprocessorn har blivit mer och mer programmerbar har det öppnat upp för att använda den till andra uppgifter än vad den ursprungligen var menad för.

Även om grafikprocessorn har varit programmerbar ett tag så har inlärningströskeln varit hög. Med hjälp av CUDA kan man med vanlig C-kod och några få tillägg programmera NVIDIAs grafikprocessorer och helt skippa de gamla programmeringsmodellerna. I detta examensarbete undersöks om det är möjligt att göra beräkningar på grafikprocessorn, utan att vara allt för insatt i hårdvarans mekanismer. En bildbehandlingsalgoritm som räknar ut det optiska flödet i en bildsekvens har implementerats. Resultatet visar att det ganska lätt går att implementera program i CUDA, men för att få effektiva program krävs fortfarande en hel del kunskap om hur grafikprocessorn fungerar.

Acknowledgments

The work has been performed at Computer Vision Laboratory, Linköping University and I would especially like to thank my supervisor Johan Hedborg for coding advices and overall assistance.

I would also like to express my gratitude to my examiner Michael Felsberg for his input and guidance.

Last but not least, I would like to thank Anne Liljedahl for her great support and proofreading the report.

Contents

List of Figures

Chapter 1

Introduction

This chapter gives a short introduction to the thesis. It describes the background, purpose and goal of the thesis, and also a list of abbreviations and the structure of this report.

1.1 Problem statement

To use pure two-dimensional analysis of image sequences can be very difficult if the temporal change is not taken into account. Objects that look similar to the background can be impossible to detect if they are not moving. If the object is moving relative to the static scene, it can be detected by calculating the optical flow. In a dense optical flow it is desired to get a motion field that describes the motion in every pixel between two image frames. There are several different methods to calculate the optical flow and in general the better result you want, the longer the execution time.

In many applications it is desirable to use the image processing in real time which puts some demands on the algorithm. Many image processing problems can be solved in a parallel way and that suits the GPU (Graphics processing unit) very well because it has an extremely parallel architecture. Earlier the GPU was difficult to use for general tasks but it is easier now, especially since the introduction of CUDA (Compute Unified Device Architecture) [1], where the standard C language with some simple extensions is used. A fast shared memory and scattered memory writes are also available which were not possible with the old methods. The common thing is not to compute just the optical flow, but to use it as a pre-step to image mosaicing, object tracking, compression etc. This forces the algorithm to be even faster since there are other computations that have to be done simultaneously. To obtain high speed in image processing is a difficult problem and it is necessary to use all the parallel processors in the GPU.

1.2 Purpose and goal of the thesis work

One goal of the thesis work is to make a GPU implementation of an image processing technique on CUDA. The technique will calculate the optical flow in an image sequence. The purpose of the thesis work is also to compare the speed with an existing (non-CUDA) implementation and to hopefully achieve good performance.

1.3 Glossary and abbreviations

- **API** - Application Programming Interface
- **CPU** - Central Processing Unit
- **CUDA** - Compute Unified Device Architecture, a technology developed by NVIDIA used to program their GPUs
- **DRAM** - Dynamic random access memory
- **GPU** - Graphics Processing Unit, the processor on a graphics card
- **GPGPU** - General-purpose computing on graphics processing units, use the GPU for other things than graphics
- **Image patch** - A small part of an image
- **Kernel** - A GPU program
- **SIMT** - Single Instruction, Multiple Thread
- **Streaming multiprocessor** - NVIDIA's GPUs contains many streaming multiprocessors and each consists of eight scalar processors
- **Thread** - A way for a program to split itself to many simultaneously running tasks

1.4 Report structure

Chapter 2: Theory
This chapter describes the theory of motion and optical flow in section 2.1 to 2.3. Section 2.4 includes information of the GPU, GPGPU and the development of these two. The last two sections (2.5 and 2.6) explains what CUDA is and how to use it.

Chapter 3: Implementation
The third chapter explains how the algorithm has been implemented on a GPU.

Chapter 4: Evaluation
In the last chapter, the results are presented in section 4.1 and is concluded with a discussion and possible future work.

Chapter 2

Theory

This chapter begins with the theory of motion and optical flow, which is followed by a description of the GPU and CUDA.

2.1 Motion

Because of technical differences and because of the large amount of storage space and computational power that were required for motion analysis, it used to be separated from general image processing. Nowadays the techniques for motion analysis no longer differ from those used in general image processing and there is no need for specialized equipment for the analysis. It is even possible to perform on a normal personal computer. Image sequence analysis enables one to recognize and analyze dynamic processes which can be used in many scientific and engineering applications such as study of flow, biological growth, industrial processes, traffic, autonomous vehicles and robots. [2]

2.1.1 Gray value changes

It can be very difficult to use pure two-dimensional analysis of image sequences if the temporal changes is not accounted for. Objects that look similar to the background in an image can be impossible to detect and segment if they are not moving. Human and animals' vision have special mechanisms to detect moving objects, and predators may only be able to see a camouflaged pray if it is moving. [3] Motion is therefore often associated with changes between images in a sequence. Figure 2.1 shows two frames extracted from a sequence. It is possible to see that the dark car on the road has moved, otherwise the images look the same. If one image is subtracted from the other, the motion will be discovered, like figure 2.2 shows. The image has been inverted and zoomed to display the interesting parts of the motion. The white parts of the image is where no movement has occurred and it is now possible to see that the car to the right also has moved a bit.

Figure 2.3 shows two images from the same scene, but this time the camera has been panned. A subtraction will look like figure 2.4. The whole scene has been

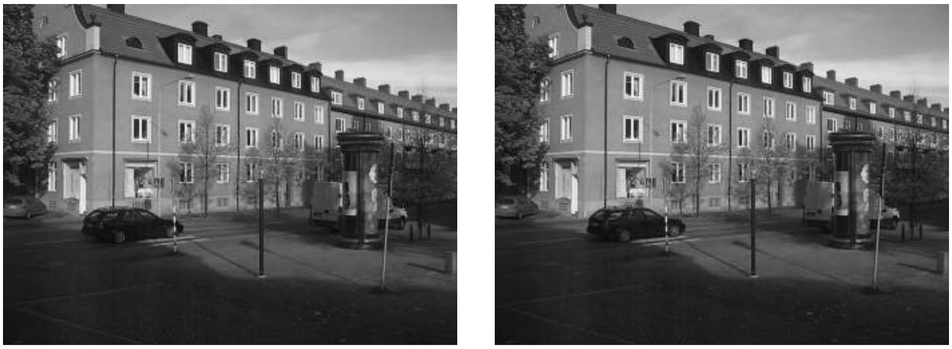


Figure 2.1: Motion in a sequence

Figure 2.2: Gray value changes caused by motion

moved but there are dark areas where the spatial gray value changes are small and the movement can not be detected there. This shows that motion might result in a temporal gray value change but it does not necessarily has to. Figure 2.5 shows

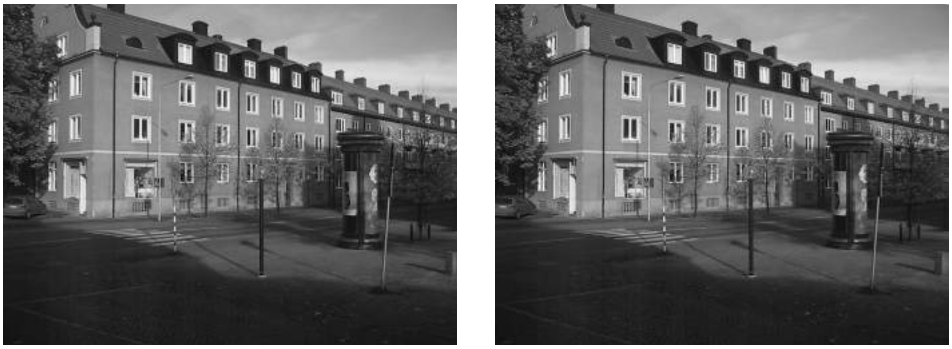


Figure 2.3: Translation of the scene

another scene where no actual movement has occurred. The difference image, figure 2.6, is however not completely dark. A lamp outside the image has been lit, causing an illumination change. Temporal gray value changes in sequences can originate from other sources than just motion. [2]

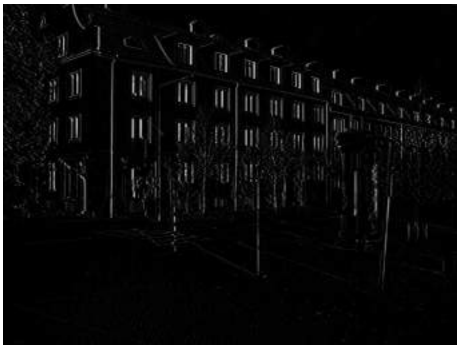


Figure 2.4: Undetected movements

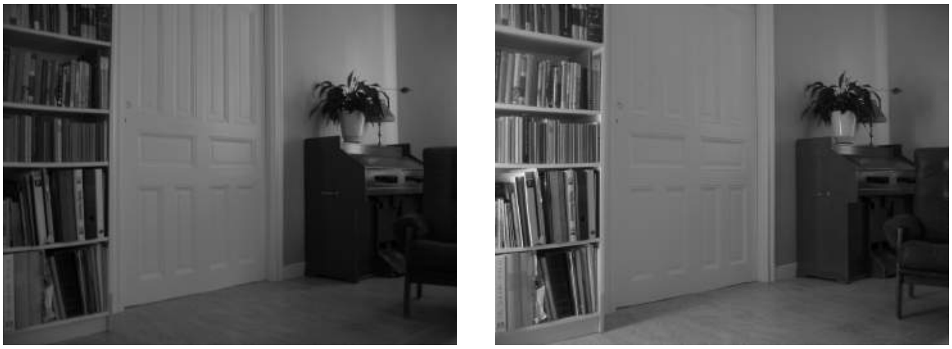


Figure 2.5: Illumination change

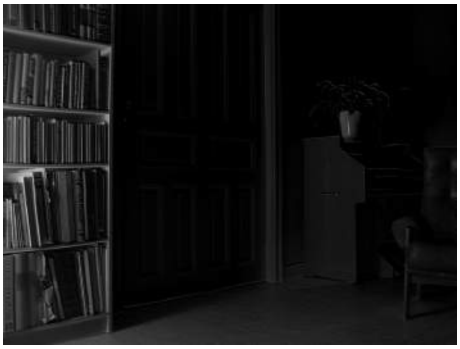


Figure 2.6: Gray value changes caused by illumination change

The difficulty in motion estimation is that you only have the apparent motion, not the real motion field. For example if a sphere with a homogeneous non-textured surface like in figure 2.7 rotates around an axis through the center and the illumination is constant, the flow will be zero even though it is moving. If the illumination source moves, it will cause an apparent optical flow field without motion of the sphere. [4] This does however not always need to be a bad thing. One example is video compression where the real motion field is of no interest, just the information in the image frames.

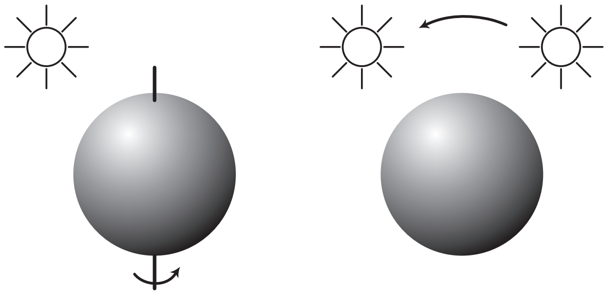


Figure 2.7: Non-textured sphere

2.1.2 Aperture problem

The gray value changes described in 2.1.1 can easily be derived with local derivative operators and can be resembled by an aperture which slides over the image. This operator only sees a small part of the image which could make the aperture problem occur. Figure 2.8 illustrates this. The solid line represents a part of the first image and the dashed line shows how it has moved in the next image. This motion between the images can be described by a displacement vector, but in the left image only the normal component can be determined. For an unambiguous solution a corner like in the image to the right is required. [5]

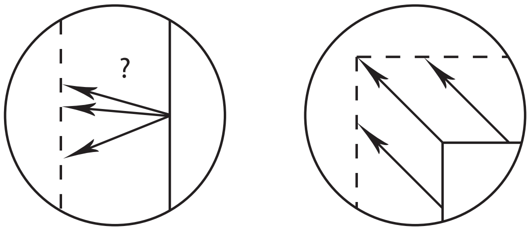


Figure 2.8: The aperture problem.

2.2 Optical flow

If the real 3D motion in the world is projected onto the image plane it is called the motion field. The motion field is however not accessible, so you have to calculate an approximation based on image data, called *optical flow*.

The optical flow can be described as the apparent motion of the brightness pattern, induced by the relative motion between the scene and the camera. [6] The optical flow is in other words based on the visual perception and if it is determined from two consecutive images it appears as a displacement vector in each pixel from one image to the other. This is called a dense optical flow and if it is not calculated in every pixel it is a sparse optical flow. The flow does not have to be calculated on gray value changes, it can also be done on color channels.

If we assume that the brightness of a patch does not change during motion and that the motion is a translation it can be expressed by:

$$f(x+\Delta x, y+\Delta y, t+\Delta t) = f(x,y,t) \tag{2.1}$$

where the displacement vector $(\Delta x, \Delta y)$ describes how the image function $f(x,y,t)$ has moved after the time Δt. Expand the left side of (2.1) in a Taylor series, neglecting higher order terms, and you will have:

$$f(x,y,t) + \Delta x\frac{\partial f}{\partial x} + \Delta y\frac{\partial f}{\partial y} + \Delta t\frac{\partial f}{\partial t} = f(x,y,t) \tag{2.2}$$

Divide with Δt and $u = \frac{\Delta x}{\Delta t}$, $v = \frac{\Delta y}{\Delta t}$:

$$u\frac{\partial f}{\partial x} + v\frac{\partial f}{\partial y} + \frac{\partial f}{\partial t} = 0 \tag{2.3}$$

Equation 2.3 is often called optical flow equation or brightness constancy constraint equation. The derivatives can be approximated but there are two unknowns (u and v) and only one equation. To obtain an unambiguous solution more information or constraints have to be added.

There are many methods to estimate the optical flow, for example methods using differential, tensor, correlation and phase techniques. Examples on algorithms that compute the optical flow is Lucas-Kanade [7], Horn & Schunck [8], Black & Anandan [9] and LP Registration [10].

2.3 Lucas-Kanade

The Lucas-Kanade algorithm was first presented in 1981 in "An iterative Image Registration Technique with an Application to Stereo Vision", [7]. The technique does not need any information from the image before execution, and works well in real time applications. It uses the spatial intensity gradient in the images to find matches in a number of iterations. In the gradient search it minimizes the Euclidean distance between the corresponding patches in the two images. The Lucas-Kanade technique starts with an initial estimate of these distances which is denoted with h or disparity below. The h values is updated in every iteration to get a better estimate of the disparity each time.

The one dimensional case is easier to illustrate than the two dimensional, but they are analogous. The problem is to find the best disparity (h) between the curves in figure 2.9 and the solution is based on the assumption that the functions $I(x)$ and $J(x)$ can be approximated linearly near x. The best h can be defined in different ways and here the best h is the disparity that minimizes (2.4).

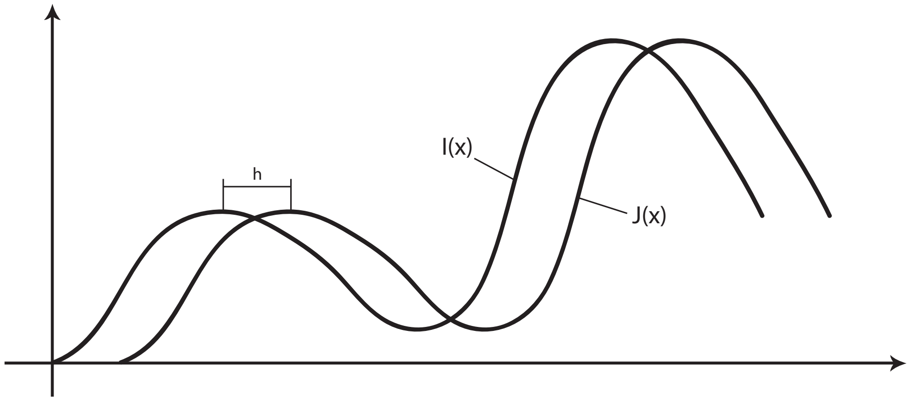


Figure 2.9: One dimensional case

$$\epsilon(h) = \int_w \Big(I(x+\frac{h}{2}) - J(x-\frac{h}{2})\Big)^2 dx \tag{2.4}$$

$I(x+\frac{h}{2})$, $J(x-\frac{h}{2})$ and their derivatives are Taylor expanded as:

$$I(x+\frac{h}{2}) = I(x) + \frac{h}{2}I'(x) + O(h^2)$$
$$J(x-\frac{h}{2}) = J(x) - \frac{h}{2}J'(x) + O(h^2)$$
$$I'(x+\frac{h}{2}) = I'(x) + \frac{h}{2}I''(x) + O(h^2)$$
$$J'(x-\frac{h}{2}) = J'(x) - \frac{h}{2}J''(x) + O(h^2)$$

Since the minimum of (2.4) is wanted, the derivative with respect to h is set to zero.

$$\begin{aligned}\frac{\partial\epsilon}{\partial h} &= \int_w \Big(I(x) + \tfrac{h}{2}I'(x) - J(x) + \tfrac{h}{2}J'(x) + O(h^2)\Big) \\ &\times \Big(I'(x) + \tfrac{h}{2}I''(x) + J'(x) - \tfrac{h}{2}J''(x) + O(h^2)\Big)dx \\ &= 0\end{aligned}$$

This results in the equation:

$$\int_w \Big(I(x) - J(x)\Big)\Big(I'(x) + J'(x)\Big)dx$$

$$= \frac{h}{2}\int_w \Big(I(x) - J(x)\Big)\Big(I''(x) - J''(x)\Big) + \Big(I'(x) + J'(x)\Big)^2 dx + O(h^2)$$

$(I(x) - J(x))(I''(x) - J''(x))$ can be omitted because the assumed linear approximation makes the term very small compared to $(I'(x) + J'(x))^2$ when h goes to zero. This together with denoting the derivatives $g(x) = I'(x) + J'(x)$ gives:

$$e = 2\int_w (I(x) - J(x))g(x)dx$$

$$Z = \int_w g(x)^2 dx$$

and the solution is $h = Z^{-1}e$.

The two dimensional discrete case is as follow

$$\begin{bmatrix} \sum\sum_w g_x^2 & \sum\sum_w g_x g_y \\ \sum\sum_w g_y g_x & \sum\sum_w g_y^2 \end{bmatrix}\begin{bmatrix} d_x \\ d_y \end{bmatrix} = 2\begin{bmatrix} \sum\sum_w (I-J)g_x \\ \sum\sum_w (I-J)g_y \end{bmatrix} \tag{2.5}$$

where g_x and g_y are the derivatives in the x and y directions. w is a small part of the image needed to compute the intensity gradient and is denoted as an image patch.

2.4 The GPU

GPU stands for Graphics Processing Unit and is one of the components on a graphics card. It handles all the computations and controls the card. Except the GPU, a graphics card consists of a video BIOS (Basic Input/Output System), memory, buss, outputs to computer displays and often a cooling system. RAMDAC stands for Random Access Memory Digital-to-Analog Converter and is used to convert the digital signals from the processor to the computer display, but the DAC-part of this component is slowly disappearing due to digital computer displays.

There are two types of manufactures, those who only manufacture IGPs (Integrated Graphics Processors) which are integrated on the motherboard and those who manufacture GPUs which can be used on dedicated expansion cards. Intel and VIA Technologies are two big IGP-manufactures and AMD/ATI, NVIDIA and Matrox manufactures GPUs and also complete graphics cards. The major task for a GPU is to visualize 3D graphics on a computer display. It also handles 2D operations but only a fraction of the memory is used for that.

When NVIDIA introduced CUDA, the GPU was seen more as a co-processor to the CPU rather than a graphics processor on its own. The new hardware that CUDA uses consists of what they call scalar streaming processors, instead of the traditional vertex and fragment processors that were used before. The traditional graphics hardware pipeline is not the same as the one in the CUDA-enabled cards, but is described shortly to make it easier to follow in the GPGPU development.

2.4.1 Traditional graphics hardware pipeline

A traditional GPU can often be described by a simple pipeline model when processing 3D graphics. It consists of a few linked computational steps where every step receives a data stream from the previous step, processes it and sends it to the next step. The geometrical objects are defined by vectors which describe points, lines, triangles and polygons, and each vector is associated with a few values such as color, normal vectors and texture coordinates. These combined are called a vertex and the main task for the vertex processor is to make different types of transformations on the vertices. Modern vertex processors are programmable so they can run programs that for example can change the position and color after the programmers choice. After this step, triangles, lines and points have been generated and projected to the image plane and those which are inside the display area must be rasterized. This is managed in the next step, where the objects are divided into fragments, which will represent pixels in the up-coming image, and the vertex values are interpolated. The fragments are sent to the fragment processor, which can perform advanced calculations and also supports texturing operations. Here the interpolation, texturing and coloring determines the final color of the fragment. The last step, raster operations, checks the fragments based on a number of tests and the pixel's color value is determined.

2.4.2 GPU history and GPGPU

General-purpose computing on graphics processing units (GPGPU) is a technique where the GPU is used for computations, other than graphics, instead of the CPU which normally would handle it. The GPUs were not programmable at all in the beginning but it has changed through newer generations.

The history of GPUs can be divided into a few generations where each generation delivered better performance and extended the programmability. Before the introduction of GPUs there existed some specialized graphics hardware developed by companies like Silicon Graphics and Evants & Sutherland. They developed many of the graphical concepts used today such as vertex transformation and texture mapping and these concepts are important in a historical point of view. Since the systems were expensive, they never reached the big market as today's GPUs.

The first generation consists of GPUs manufactured until 1998, including cards up to NVIDIA's TNT2, ATI's Rage and 3dfx's Voodoo3 and they were capable of rasterizing pre-transformed triangles and could apply one or two textures. These GPUs had a limited set of math operations and could not transform vertices of 3D objects, which instead were done on the CPU. The GPU had however taken over the task of updating individual pixels from the CPU.

The second generation, 1999-2000, includes NVIDIA's GeForce 256 and GeForce 2, ATI's Radeon 7500 and S3's Savage3D. These GPUs were capable of 3D vertex transformation and lighting, and both OpenGL and DirectX supported this. The set of math operations for combining textures and coloring pixels became larger in this generation and they were more configurable than the previous generation but not yet programmable.

2001 was the year of the third generation including NVIDIA's GeForce 3 and GeForce 4 Ti, Microsoft's Xbox and ATI's Radeon 8500. These GPUs provided vertex programmability and ability for configuration on pixel-level but not true pixel programmability.

The fourth generation GPUs including GeForce FX series and Radeon 9700 came 2002-2003. These provided both vertex and pixel-level programmability which made it possible to offload the CPU from complex vertex transformation and pixel-shading operations. These are the first GPUs that could be used for GPGPU and since then the development have made it easier to program the GPUs. [11]

Even though it was possible to use GPUs for GPGPU in the fourth generation it was quite difficult because the programmer had to use a graphics API (application programming interface) such as OpenGL or DirectX to do the low-level programming. Later on higher level programming environments such as Cg (C for graphics), HLSL (High Level Shader Language) and GLSL (OpenGL Shading Language) made it easier for GPGPU-programming but these generate code into the APIs, so programmers still need to know a lot of computer graphics to get high performance and the APIs limit what kind of applications that can be written.

NVIDIA's technology CUDA, together with new hardware, makes it possible to use standard C-code, with a few extensions, to program the GPU which is more described in section 2.5. Usually parallel software were developed for servers, data centers or clusters where the market is not so big, but with the usage of a GPU it reaches many customers. Graphics is however still the main business and the manufactures can not afford to make the cards to expensive, otherwise it will not make it on a competitive market.

2.4.3 GPU vs CPU

The CPU and GPU have different designs which is mirrored in how their performances have developed. The developers of the CPU have raised its clock frequency to improve the performance. Most CPUs were running hot while consuming a lot of power and it is difficult to improve much more in this way. The GPU is however based on another philosophy, more execution units are added to get better performance. Figure 2.11 shows how big the difference is in floating point operations per second (FLOP) and the bandwidth. There are mainly three reasons why the graphics hardware is advancing so fast. Firstly, the number of transistors that fit on a microchip is roughly doubled every 18 months, resulting in cheaper and faster computer hardware. Secondly, image generation is very parallel and graphics hardware designers can split up the problem into smaller tasks that are easier to tackle. The last reason, that combines the first two, is the human desire for visual stimulation and entertainment. The computer games are pushing the industry with higher demands on speed and image quality.

The CPU is designed to be a general purpose processor. The GPU is faster on many graphic tasks, because it has a specialized design. The GPU is very well-suited for compute-intensive and data-parallel computations, for example rendering. More transistors can be devoted to data processing instead of data caching and flow control, see figure 2.10. Since the same program is executed on many data elements in parallel the need for advanced flow control is lowered. The CPU is designed to have much data on the chip, that can be accessed with short latency. The GPU has many execution units instead and a small amount of on-chip data, which could force many DRAM-accesses. The GPU is however designed to tolerate DRAM latency by having a huge number of threads that can execute on the processors.

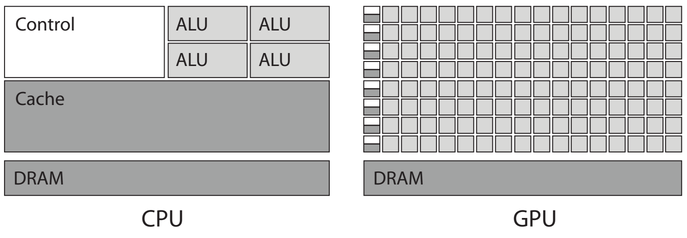


Figure 2.10: The GPU uses more transistors for data processing

The difference between GPU and CPU threads is that GPU threads are very lightweight and have little creation overhead. The hardware can generate the threads in just a few clock cycles. The GPU needs thousands of threads to achieve full efficiency while a multi-core CPU only needs a few. A CPU-thread is usually an abstraction of the processor, so when using a 866 processor each thread will have access to all the CPU registers and CPU memory space, and will have entire sets of separate registers. On the GPU, while using CUDA, all blocks that are assigned to a multiprocessor (described more in section 2.5.2) will dynamically share all the resources (register and shared memory) across all the threads that are running on the processor.

Not all kinds of programs can be successfully implemented on the GPU, but if an application require long execution time it is common there is a lot of data being processed. The GPU can not replace the CPU because it is not able to execute general-purpose programs like the CPU, but the GPU will be much faster than the CPU if the task is parallel in its nature and therefore suitable.

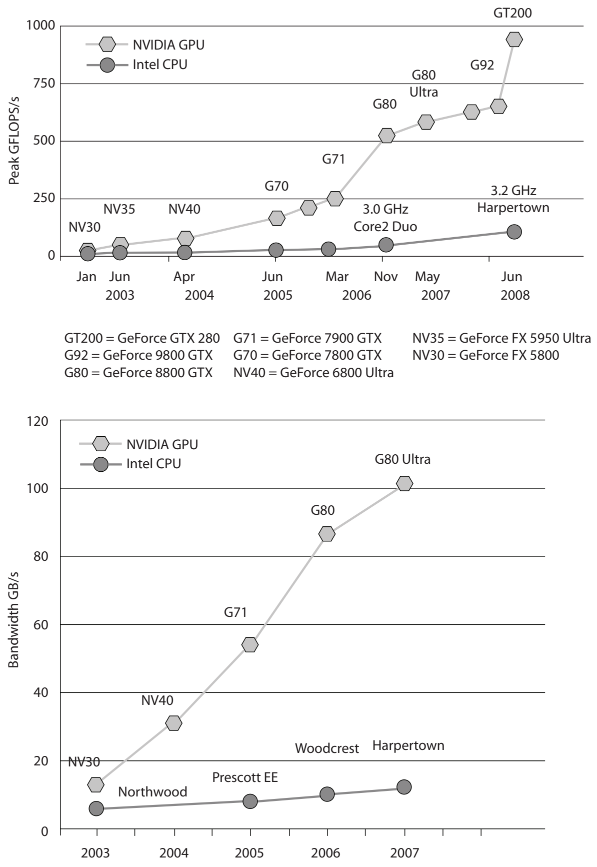


Figure 2.11: Floating point operations per second and memory bandwidth for the Intel CPU and NVIDIA GPU

2.5 CUDA

CUDA is an acronym for Compute Unified Device Architecture, which is a technology developed by NVIDIA. It is a parallel programming model and software environment designed to enable programmers and developers to write software for GPUs and CPUs, using the standard C language with some simple extensions. It sees the GPU as a co-processor to the CPU, with its own DRAM and is used to run many threads in parallel. The GPU is referred to as the *device* and the CPU as the *host*. CUDA is available for the GeForce 8 series and higher, the Tesla solutions and some Quatro cards, see [12] for a full list of supported devices.

These models have new hardware, which makes it possible to skip the programming through the traditional interface and the software layers have been redone so that C-like code can be used. This enables a more general purpose programming interface and also gives access to more regular CPU-instructions. The new hardware together with CUDA makes it possible to use scattered memory writes and a fast shared memory, see section 2.5.3 . [13]

The software is based on different layers, as shown in figure 2.12. These layers are the hardware CUDA driver, the CUDA runtime, and two higher-level mathematical libraries, CUBLAS and CUFFT which are described in [14] and [15]. [16] The CUDA runtime library is split into three components, a common, a host and a device component. The common component provides built-in vector types and a subset of the C runtime library in both host and device codes. The device component provides device-specific functions, for example some mathematical functions which are less accurate but faster than normal functions and a synchronization function which synchronizes all threads in a block. The host component provides, among others, functions that deal with device management (use multiple device systems), memory management and error handling. [17]

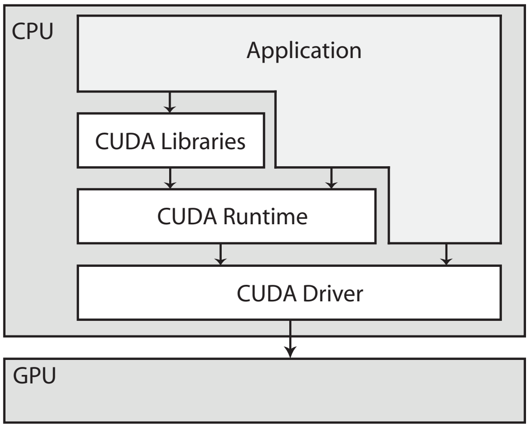


Figure 2.12: CUDA Software Stack

2.5.1 Thread batching

As described earlier, the GPU is very well suited for tasks where a large amount of data shall be processed in parallel. To do so a lot of threads are needed, and each and every one of these executes the same program, called a *kernel*. The threads are organized into a number of thread blocks and each thread gets an index, *threadIdx*, which begin at zero in the block and the index corresponds to the thread number within each block. The threadIdx can be 1-, 2- or 3-dimensional to help the addressing in a program. [16] All the threads within a block can be synchronized and exchange data through the shared memory. The shared memory is described more in section 2.5.3. Currently the maximum number of threads per block is 512. To get an efficient program a lot more threads than that will be needed and blocks of the same dimensionality and size can therefore be executed by the same kernel and batched into a grid of blocks. The threads in one block can however not communicate or synchronize with threads in another block, but this model make it easy to run the program on different devices without recompilation. If the program is run on a device with few parallel capabilities the blocks will be run more sequentially. The only way to synchronize between blocks is to write the data to global memory (see section 2.5.3), terminate the kernel and all the threads, and then restart the kernel and read the data back. [18] The block also receives an index like the threads, *blockIdx*, and can be 1- or 2-dimensional. The maximum size of each dimension of the grid is 65535. [19] Figure 2.13 shows how the threads can be organized in a program. Kernel 1 and kernel 2 are two different programs and grid 1 and grid 2 can have different number of blocks. The number of threads per block must be the same within a grid but can differ from other grids.

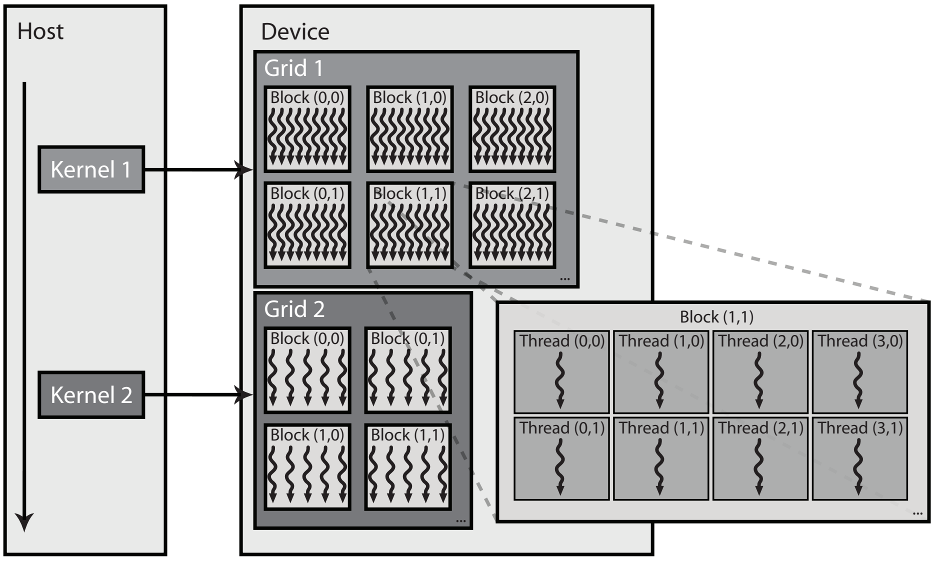


Figure 2.13: Organization of the threads

2.5.2 Hardware

The device contains a set of multithreaded streaming multiprocessors. The G80 (GeForce 8800 GTX, see figure 2.11 for more abbreviations) has 16 streaming multiprocessors and the GT200 has 30 [20], [21]. Each of these have a single instruction, multiple thread architecture, SIMT, and when a CUDA program invokes a kernel grid, the blocks are distributed to the multiprocessors with available execution capacity. Each multiprocessor currently consists of eight scalar processors, which executes the same instruction but operates on different data, two super function units, a multithreaded instruction unit and four different types of on-chip memory. The latter are local 32-bit registers (per processor), parallel data cache also called shared memory, read-only constant cache and read-only texture cache. The texture cache can be accessed via a texture unit. [16], [19] Figure 2.14 shows how a device can be built with many multiprocessors. For the G80, N in the figure corresponds to sixteen and M to eight, which gives a total of 128 processors. Figure 2.15 gives a closer view, where two streaming multiprocessors (SM) are grouped together. The scalar processors (SP), super function units (SFU) are also visible. [22]

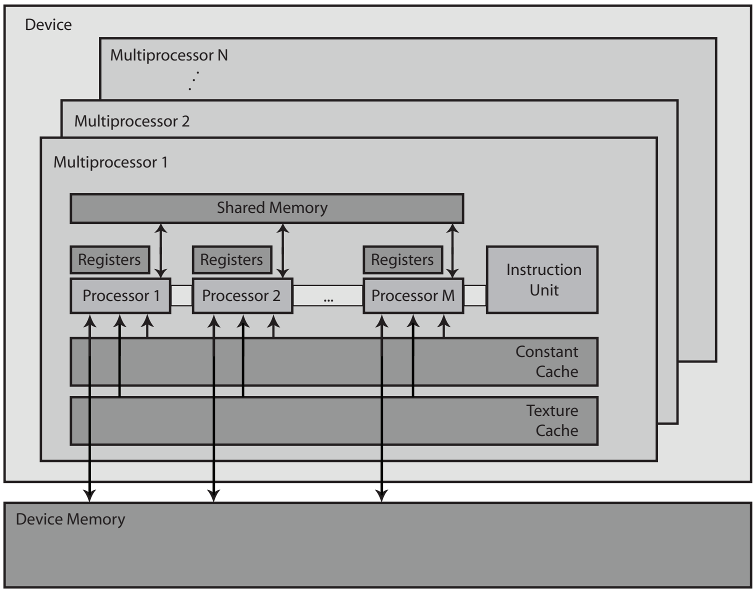


Figure 2.14: A device with many multiprocessors

The multiprocessors SIMT unit creates, manages, schedules and executes threads in groups of 32 parallel threads called *warps*. When blocks are assigned to a multiprocessor they are split into warps and then a warp that is ready to execute is selected and issues the next instruction to the active threads in the warp, and each thread is mapped to one scalar processing core.

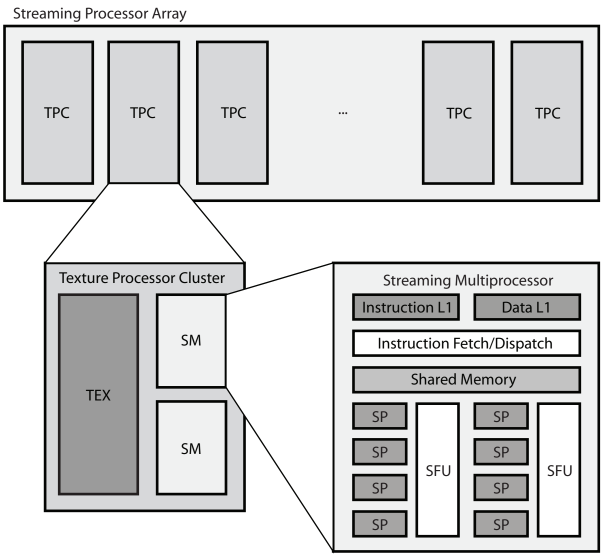


Figure 2.15: Hardware architecture

2.5.3 Memory model

With CUDA you can use gather and scatter memory operations which means that you can read and write in any location in DRAM, just like on a CPU. Figure 2.16 shows how the memory can be read and 2.17 how it can be written.

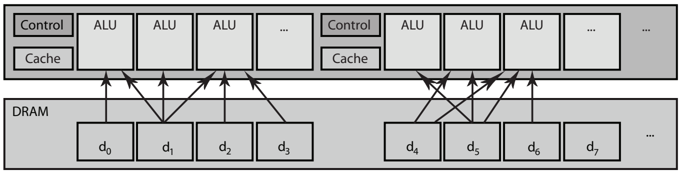


Figure 2.16: Gather

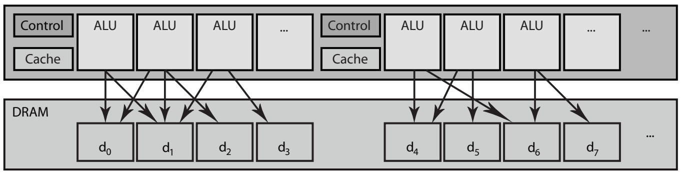


Figure 2.17: Scatter

A CUDA thread executed on the device has memory access through a private local memory, but even though it is called local it resides in DRAM and is slow. The shared memory is on the other hand very fast but if data is exchanged between blocks the global memory must be used. There are also two read-only memory spaces, the texture memory and the constant memory. See figure 2.18 for an overview of the memory model. [16]

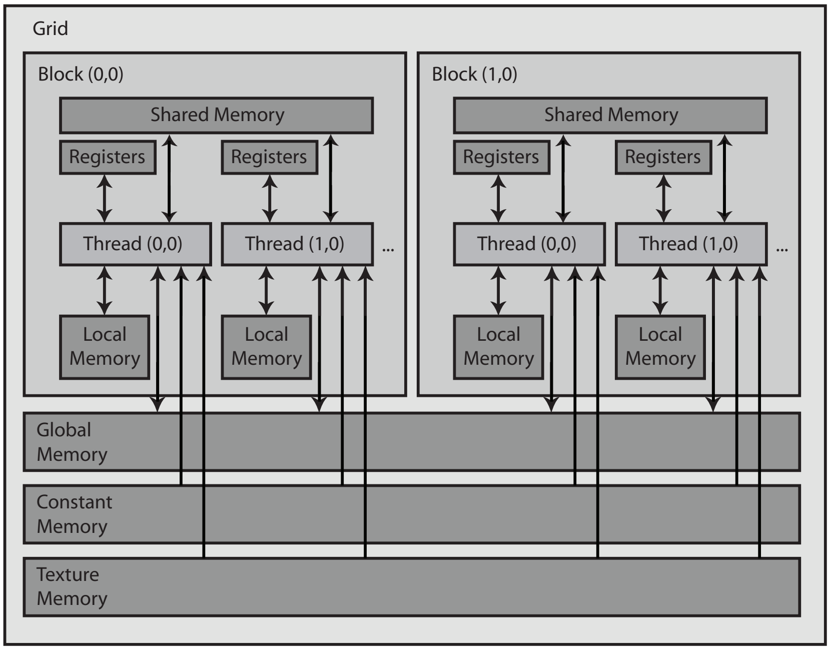


Figure 2.18: Memory Model

Shared memory

The shared memory is a parallel data cache with a very fast read and write access. The threads in a block can use the shared memory to exchange data with each other to minimize the DRAM access. Without this memory, different threads who uses the same data will have to access it one by one through the slow DRAM, see figure 2.19.

If the implemented algorithm is constructed in a way so that the threads will need data from a surrounding area or the same as other threads, the data need only to be read once from the DRAM into the shared memory and can then be accessed from there as shown in figure 2.20. Because the shared memory is accessible from all the threads, it requires a lot of wiring. This is expensive and the shared memory is limited to 16 kB. It is organized into sixteen banks to achieve high memory bandwidth but bank conflicts can arise. These banks can be accessed

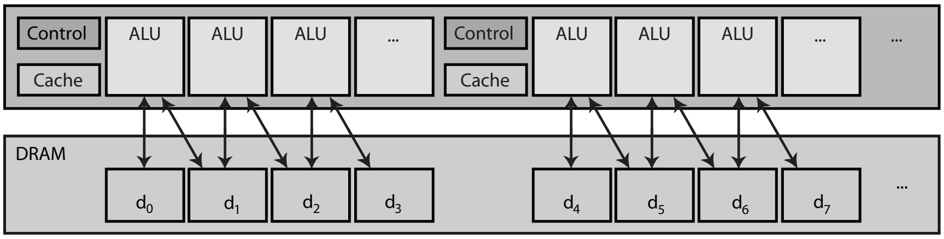


Figure 2.19: Without Shared memory

simultaneously and if no bank conflicts occur the shared memory is as fast as a register. If two or more memory requests fall into the same bank they have to be serialized, decreasing the effective bandwidth. [16]

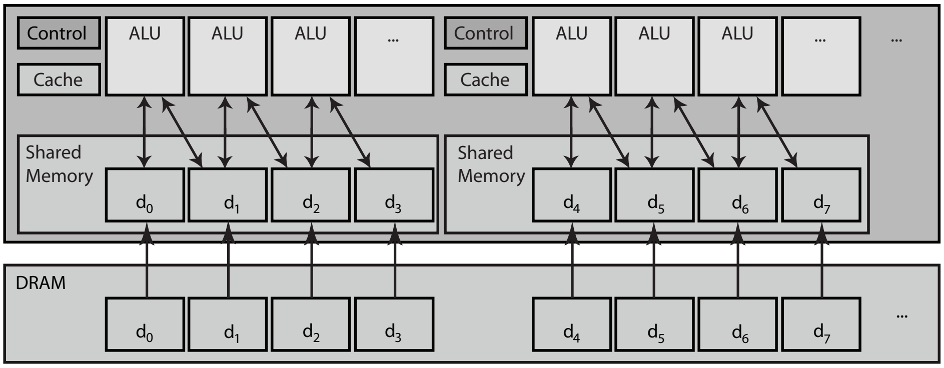


Figure 2.20: With Shared memory

Texture memory

CUDA supports some of the texturing hardware used by the GPU to access graphics from texture memory. A region of memory, called a texture, has to be bound before it can be used by a kernel, and will be read using device functions, texture fetches. The texture cache is shared by all the scalar processors and speeds up the readings from texture memory space, and is optimized for 2D spatial locality. The cache working set varies between 6 and 8 KB per multiprocessor. The texture reference has several attributes that can be set to be suitable for the current task. The dimensionality can be one or two dimensional and the coordinates can either be normalized (0 to 1) or in the same range as the texture dimension. The input and output data types of the texture fetches can be set and the addressing mode changes what will happen when the coordinates is out of range. Clamp is set when unnormalized coordinates are used and it means that coordinate values below 0 are set to 0 and values equal or greater than the texture size N in that dimension, are set to $N-1$. This can also be used together with normalized coordinates as well as the wrap mode. The wrap mode is good to use on textures with a periodic nature. Just the fractional part is used so 1.25 is treated the same as 0.25. Linear texture filtering is also available and performs low-precision interpolation between neighboring pixels. Normalized, addressMode, and filterMode may be directly modified in host code.

When binding textures, it can either be done on data allocated in linear memory or in a CUDA array. When linear memory is used, you can only use one dimensional addressing with non-normalized integer coordinates, texture filtering can not be used and the addressing mode is always set to return zero for out-of-range values. The result can however be written in the texture in contrast to using CUDA arrays, which only are readable.

Constant Memory

The constant memory is a read-only memory space and is cached just like the texture memory. It is most efficient when the CPU initializes some read only data which is accessed by many threads. The hardware is built so that all threads can access the same value at the same time. The constant memory is accessible from the whole grid and the total amount of memory is 64 KB.

2.6 Coding with CUDA

As described before in section 2.5, CUDA is a programming model which sees the GPU as a dedicated massive data parallel co-processor and the only trace of a graphics processor is the memory called texture memory. [18] Programs that are executed on the GPU, the kernels, generates a lot of threads which are organized as described in 2.5.1. One of the language extensions is the function type, which specifies where the function is executed and called:

Function type	Executed on	Only callable from
`__device__`	Device	Device
`__global__`	Device	Host
`__host__`	Host	Host

Table 2.1: CUDA function declarations

The `__global__` type is the function/program referred to as the kernel and must return `void`. Other restrictions that apply to `__device__` and `__global__` functions are that they do not support recursion, you cannot declare static variables inside their bodies and they cannot have a variable number of arguments. `__device__` functions cannot have their address taken. All `__global__` calls are asynchronous since CUDA 1.0, which means that it returns before the device has completed the execution and the host can continue its program.

2.6.1 Execution configuration

To be able to launch a kernel you have to specify a special execution configuration. It specifies the size of the grid and the thread blocks described in section 2.5.1.

Example 2.1: Function declaration and call

A function declared as
`__global__ void function(float* parameter);`
can be launched with
`function<<< gridDimension, blockDimension >>>(parameter);`
gridDimension and blockDimension have to be declared in advance with
```
dim3 gridDimension(200, 30);
dim3 blockDimension(8, 8, 4);
```
This configuration will launch a total of 1536000 threads in 6000 blocks, 256 threads per block.

In addition to gridDimension and blockDimension in example 2.1 the shared memory size and a associated stream can also be specified.

2.6.2 Additional language extensions

The function types `__device__` , `__global__` and `__host__` described in section 2.6 and the execution configuration described in the previous section 2.6.1 are parts of the extensions to the C language. Except these two extensions there also exist variable type qualifiers and five built in variables. Two of these variables, *threadIdx* and *blockIdx*, have been mentioned in section 2.5.1 and specifies the block and thread indices. There are three additional built in variables, namely *gridDim*, *blockDim* and *warpSize*. *gridDim* and *blockDim* specifies the grid and block dimensions and are together with *threadIdx* and *blockIdx* a good help in thread management. *warpSize* contains the number of threads in a warp. [19] These keywords are actually not real variables but hardware registers. [18]

The variable type qualifiers is used to specify the memory location on the device of a variable and these are `__device__` , `__constant__` and `__shared__` . The `__device__` qualifier declares that a variable resides on the device and can be used together with one of the other qualifiers, if not, the variable will reside in the global memory and be accessible from all threads within the grid and from the host. `__constant__` declares a variable that resides in constant memory and are accessible from all threads within the grid and from the host. `__shared__` declares a variable that resides in shared memory space of a thread block and is only accessible from the threads within the same block. Writes to the shared memory is only guaranteed to be visible to other threads after the execution of a `__syncthreads()` .

2.6.3 Choosing memory space

When implementing an algorithm you have to decide what kind of memory the data will reside in. It depends very much on the access pattern. The device memory (global and local memory) have a much higher latency and lower bandwidth than the on-chip, shared memory. The constant memory is as described in 2.5.3 good for read only structures, for example a constant value that will be used by a lot of threads. It will only be read from the device memory if it is a cache miss, otherwise from the constant cache. All threads in a half-warp, which is either the first or the second half of a warp, can access the value simultaneously and is as fast as a register. If different addresses are read the cost scales linearly.

The texture memory is optimized for 2D spatial locality as described in 2.5.3 and has many features such as hardware filtering and out of bound handling. The cache works well when accessing values near each other in a random way, but this memory is only readable while using CUDA arrays.

The global memory is not cached like the constant and texture memory and because of that, the access pattern is very important to get a high bandwidth. To get efficient reading and writing, the simultaneous memory accesses in a half-warp can be coalesced into a single transaction. The requirements to achieve this are dependent on the device's *compute capability*, and if they are fulfilled, the transaction will be coalesced even if some threads do not access the memory. The compute capability is defined by a major revision number and a minor revision number. Devices with the same major revision number have the same core architecture and the minor revision number corresponds to some improvements to the core architecture.

Devices with compute capability 1.0 and 1.1

The threads must access 32-bit, 64-bit or 128-bit words and they will result in, respectively, one 64-byte, one 128-byte or two 128-byte transactions. All the words must lie in the same segment with size equal to the transaction size, or twice the size in the 128-bit word case. The last requirement is that the thread must access the words in sequence. Figure 2.21 shows a coalesced memory access and figure 2.22 on page 26 a non-coalesced. Coalesced 32-bit access deliver the highest bandwidth and 128-bit lowest.

If the requirements are not fulfilled a separate memory transaction will be issued and performance is reduced.

Devices with compute capability 1.2

If all the threads in the half-warp lies in the same segment, the memory access will be coalesced. The size of the segment has to be 32 bytes if all the threads access 8-bit words, 64 bytes if there are 16-bit words and 128 bytes when the access is on 32-bit or 64-bit words. Unlike devices with compute capability 1.0 and 1.1, the threads do not need to access the words in a sequence. For higher compute capability, they can have a random access pattern including patterns where different threads access the same address, see figure 2.23 on page 27. If the threads in a half-warp access

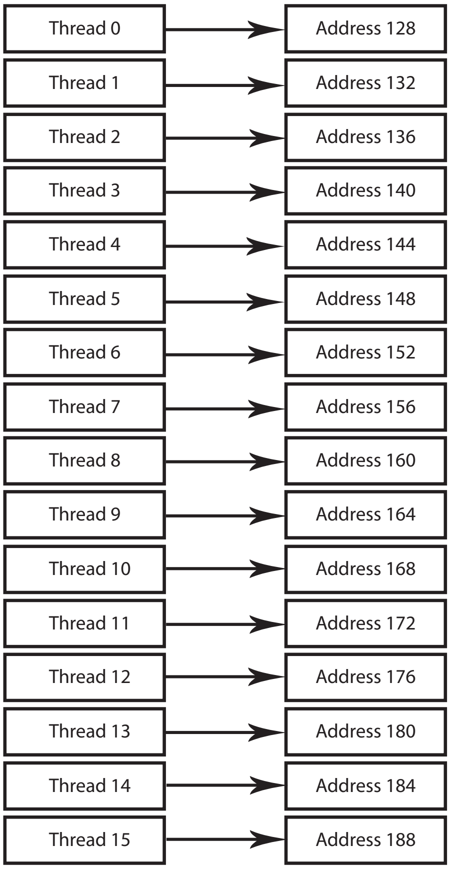


Figure 2.21: float memory access coalesced into one memory transaction

n different segments there will be n memory transactions. For devices with lower compute capability it would issue 16 transactions when two segments or more will be used.

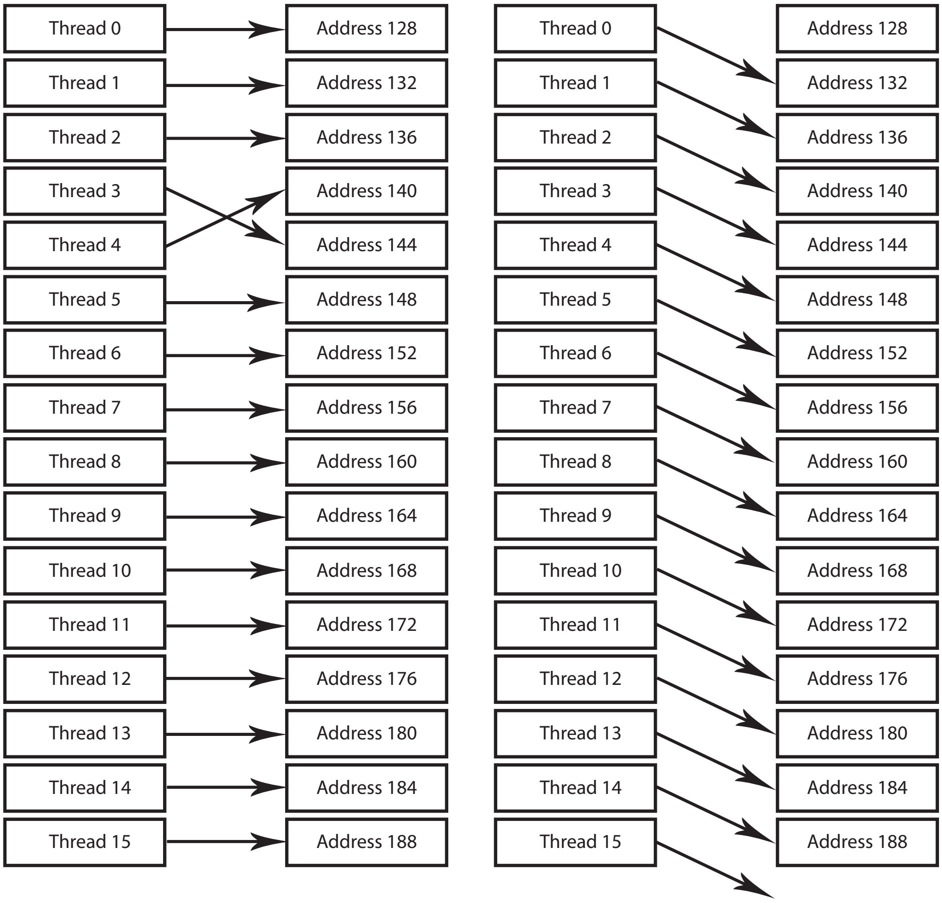


Figure 2.22: Examples of memory access patterns that are non-coalesced for devices of compute Capability 1.0 or 1.1 resulting in 16 memory transactions

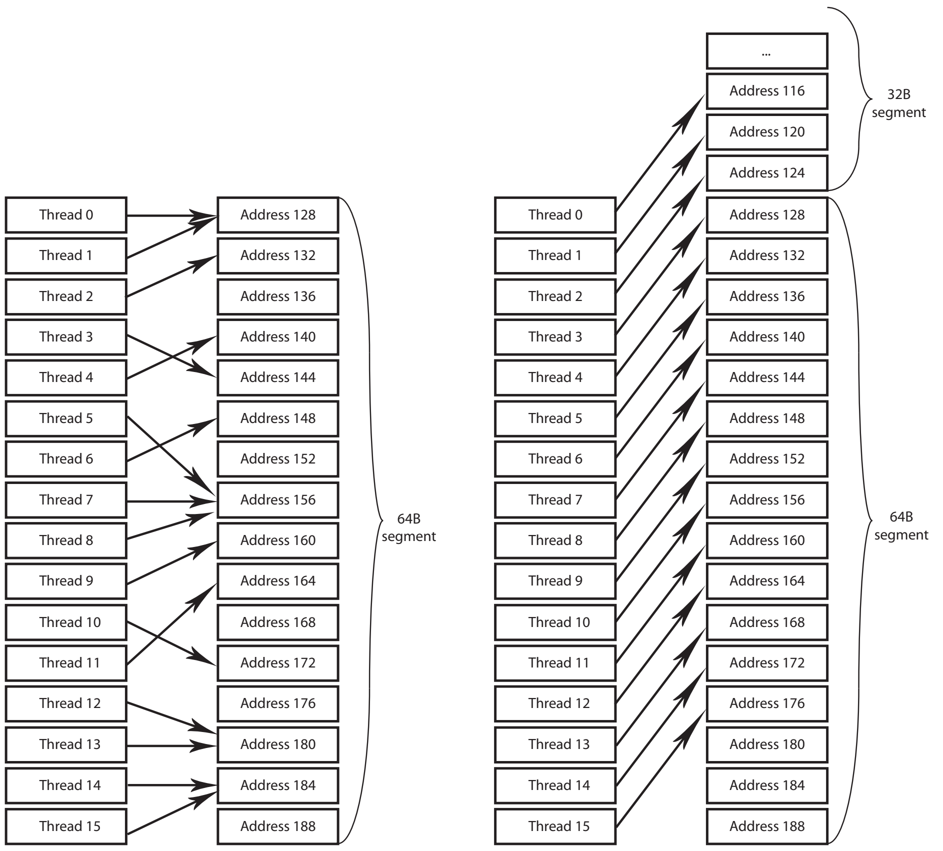


Figure 2.23: Compute Capability 1.2 and higher. Left: Random access pattern resulting in one memory transaction. Right: Misaligned access pattern resulting in two memory transactions.

Chapter 3

Implementation

The Lucas-Kanade algorithm has been implemented in CUDA on a Core 2 Duo E6600 2.4GHz CPU and a GeForce 8800 GTX GPU. The following section describes how this was done in more detail.

3.1 Algorithm

3.1.1 Overview

The Lucas-Kanade algorithm can be broken down into a few steps, see figure 3.1 for an overview. Each of these steps are done using image patches. If the resulting optical flow is not dense there are many different methods to choose the position of the patches. One example is the Harris operator [23] which can be used as a pre-step to decide where to compute the optical flow.

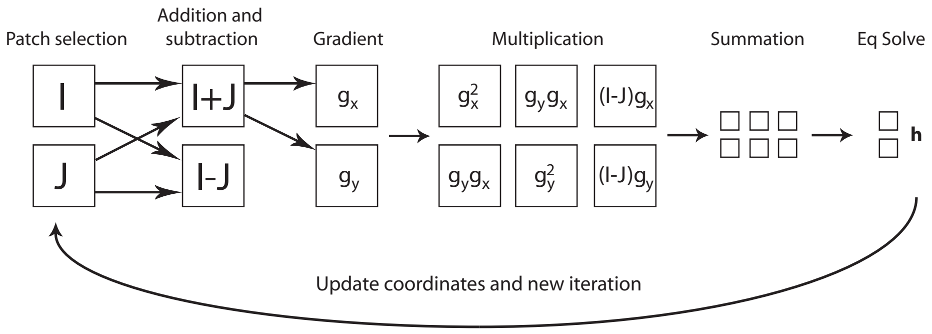


Figure 3.1: Algorithm overview

One patch from each image is extracted resulting in patch I and J. These are added together and subtracted into two new patches, $I + J$ and $I - J$. $I + J$ is then filtered in both directions with the sobel operator, creating an additional two patches, g_x and g_y. These two are together with $I - J$ combined into another six new patches, g_x^2, g_y^2, $g_y g_x$, $g_x g_y$, $(I - J)g_x$ and $(I - J)g_y$. $g_y g_x$ and $g_x g_y$ are actually the same and therefore only five new patches have to be calculated.

These patches are elements in (2.5) but before the equations can be solved, the values in each patch have to be summarized. The last step is to solve the equation system and obtain a new estimate for the disparity. The estimations are represented with floats and a sub-pixel accuracy. This is done on many patch pairs in parallel and the new coordinates for the patch positions are then used as input for the next iteration of the algorithm.

3.1.2 Sobel

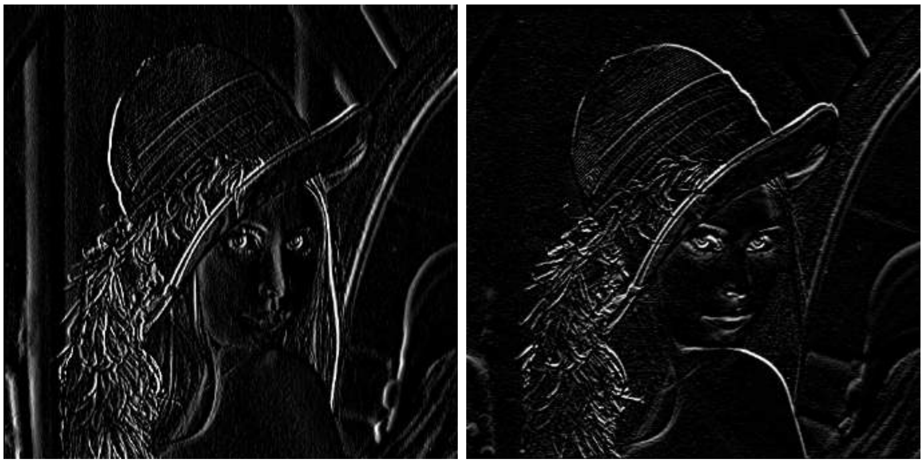
The sobel operator in the x-direction is defined by:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

and in the y-direction by:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

An example of an image convolved with these filters is presented in figure 3.2.



(a) Sobel-x (b) Sobel-y

Figure 3.2: Image filtered with the sobel operator

The algorithm creates one patch filtered in the x-direction called g_x above and another patch filtered in the y-direction, g_y, both created from the patch $I + J$. A straightforward way to implement this step could be for each thread to read its operands (pixels) from the global memory, make the calculation and then write the resulting pixel back to global memory. With this method it can be difficult to get efficient reading from global memory and above all, the bandwidth is wasted since the neighboring pixels use the same input data. One calculation needs six values and each of these, if they do not lie on the border, would have to be read six times, see figure 3.3. The capital letters represent the origins to six kernels and the lower-case letters represent which pixels corresponding filter reads data from. The black pixel needs to be read six times by all these filters. Since the bandwidth is low and the latency are high for global memory, most the time will be spent on memory transactions.

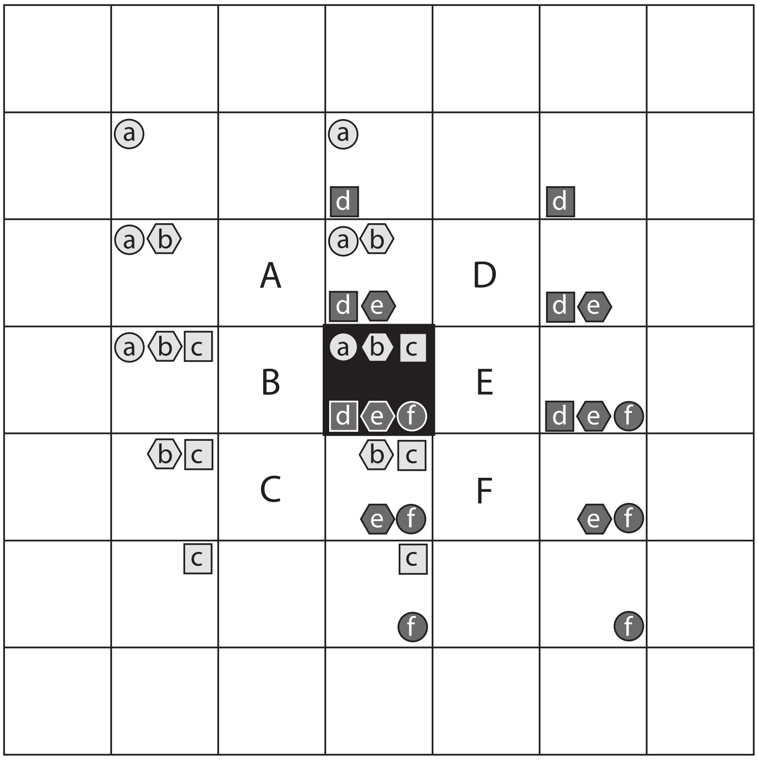


Figure 3.3: Data accesses with sobel

A more efficient way to implement this step is to make use of the shared memory. This can be done by making all the threads in a block read one value from the global memory and put it into the shared memory so the whole block resides there. This results in that one value just have to be read once from the global memory and the threads access all needed values from the fast shared memory.

One issue with this method is that a border problem between thread blocks will be introduced. The problem at the image border also exists in the method described above, where you have to decide how to treat accesses outside the image, but because of limitations in the number of threads per block and the size of shared memory a new problem arises. The current maximum number of threads per block is 512 and they have to read more than one value each to put the whole image in shared memory. The shared memory size is however 16 kB and if the image size is not very small it will not fit there. This forces many blocks to be used and pixels at the edge of one block will depend on pixels that are not in shared memory. This can be compensated by adding a border, extending the block with two rows and two columns. However, these new threads will be idle during the computational process, and they can complicate the coalescence of memory transactions. If the convolution kernel is large (sobel has radius one) there are a few more things that can improve the filtering speed, described in [24].

A third method to implement the filtering is to make use of the texture memory with its cache. It gives a more straightforward implementation and is not dependent on the shared memory. If the kernel size is large the shared memory method tends to be faster but since the kernel size of the sobel operator is small, the use of the texture memory is a more efficient technique. Another great advantage using texture memory is that the border problem at the edge of the image can be handled by the hardware by setting the appropriate addressing mode, see section 2.5.3.

3.1.3 Sum

There are a few ways to implement parallel reduction and the paper "Optimizing Parallel Reduction in CUDA" [25], describes how this can be done efficiently in different versions. One condition to get high performance programs in a GPU implementation is, as described in section 2.4, that the data size is large. In this part of the algorithm the data needed to be summarized is pretty large, but it is divided between many patches. There are a big number of patches, with each patch being quite small.

The first part of the algorithm outputs three patches g_x, g_y and $I-J$ per every point that is going to be tracked. From these three patches, five new (g_x^2, g_y^2, g_xg_y, $(I-J)g_x$ and $(I-J)g_y$) is desired but the equation system solver requires scalars so these must be summarized. The multiplication and the summation are done at the same time by first reading the g_x, g_y and $I-J$ patches into the shared memory. In the shared memory the values in every patch are multiplied and reduced to scalars in parallel by multiple threads. Even though the summation is done per patch, all patches of the same type is packed together in the memory. An improvement compared with ordinary GPGPU is that with CUDA more than four output arguments can be used. This makes it possible to compute all the five sums (g_x^2, g_y^2, g_xg_y, $(I-J)g_x$ and $(I-J)g_y$) in the same kernel and write the result to global memory.

3.1.4 Equation solver

As described in section 2.3 the algorithm tries to find the disparity h, which minimizes (2.4), by solving the equation system

$$\underbrace{\begin{bmatrix} \sum\sum_w g_x^2 & \sum\sum_w g_x g_y \\ \sum\sum_w g_y g_x & \sum\sum_w g_y^2 \end{bmatrix}}_{\mathbf{Z}} \underbrace{\begin{bmatrix} d_x \\ d_y \end{bmatrix}}_{\mathbf{h}} = 2 \underbrace{\begin{bmatrix} \sum\sum_w (I-J) g_x \\ \sum\sum_w (I-J) g_y \end{bmatrix}}_{\mathbf{e}}.$$

After the summation, **Z** will consist of four values and h can be obtained by

$$\mathbf{h} = \mathbf{Z}^{-1}\mathbf{e} \tag{3.1}$$

if the inverse exists. If not, the h value will be incorrect, and it can be due to a textureless part of the image. If a sparse field is calculated, the interest points will probably be near a trackable area but there may be an affine transformation between the images that gives an incorrect value of the disparity. An attempt to identify these values is described in section 4.1.

(3.1) has been analytically solved creating two functions:

$$d_x = 2\frac{\sum g_y^2 \sum (I-J) g_x - \sum g_x g_y \sum (I-J) g_y}{\sum g_x^2 \sum g_y^2 - (\sum g_x g_y)^2}$$

$$d_y = 2\frac{\sum g_x^2 \sum (I-J) g_y - \sum g_x g_y \sum (I-J) g_x}{\sum g_x^2 \sum g_y^2 - (\sum g_x g_y)^2}$$

where d_x and d_y are the coordinates for how patch J has moved. These values can be calculated in parallel for as many patches as the hardware allows, by using many threads.

3.2 Device emulation

When implementing algorithms and programs, debugging is a very good tool. It is used for finding and reducing the number of bugs/error in the program. For CUDA, the programming environment does not support debugging code running on the device, but you can set a device emulation mode that can be used for the same purpose. The code is executed on the host and the native debugging support can be used as if it was a host program, enabling the programmer to use breakpoints, inspection etc. It is also possible to use host specific functions, like `printf()` , on the device which is not callable from the device in normal mode. This also applies to data, the device functions can call functions and access data on the host, and the other way around (host to device) is also true. The runtime is also able to detect deadlock situations caused by improper usage of `__syncthreads`.

Since it is an emulation and not a simulation of the device, there are some differences between running in emulation mode and on the actual device. Simultaneous accesses of the same memory location by multiple threads could produce different results. This is because emulation mode threads are executed sequentially or at least much less simultaneously than on the device. Even though it is possible to reference data on the host from the device and vice versa and the correct result is obtained in emulation mode, it will probably generate an error when executed in device mode. There may also be a difference in floating point results because functions and the storage of values will be done at a higher precision. This can be adjusted by forcing the precision on the host, but the values can still slightly differ because of different compilers or architectures.

Chapter 4

Evaluation

In this chapter the results are presented together with a discussion of the thesis and possible future work.

4.1 Results

The Lucas-Kanade algorithm has been implemented in CUDA and tested with gray scale images. These are provided by Middlebury Vision [26] together with what they call the ground-truth flow, which can be used to evaluate the performance of optical flow algorithms. They have different techniques to create these datasets with ground-truths and one is based on hidden fluorescent texture with computer-controlled lighting and motion stages for camera and scene. Ultra violet light makes the high-frequency fluorescent texture visible for accurate tracking. They also provide ground-truths from realistic synthetic sequences and real stereo imagery of rigid scenes. The setups have higher spatial and temporal resolution than the data-sets. [27]

There are two common ways to illustrate the optical flow and one of these is to write vectors for every patch, representing the disparity. Figure 4.1 shows how a rectangle has moved to the right and the background up to the left. However, if the flow field are large and dense, it can be difficult to see the result.

Another method to visualize the result can be to use a color representation. Depending on the disparity's angle, a color flow image can be created. Using a color scheme like in figure 4.2(a), the corresponding image to figure 4.1 would look like figure 4.2(b).

The algorithm does not give perfect results and it can be interesting trying to detect incorrect matches. One solution can be to look at the length of the displacement vector and compare it to its neighbors, or if the maximum displacement is known, act accordingly. It can however be hard to decide the threshold for what is accepted as a match or not, and another method is to run the algorithm again, starting at the estimated coordinates in the second image and see if it goes back to the original coordinates in the first image. If the second run matches the "start patch" it is regarded as correct, but if the new displacement does not return to

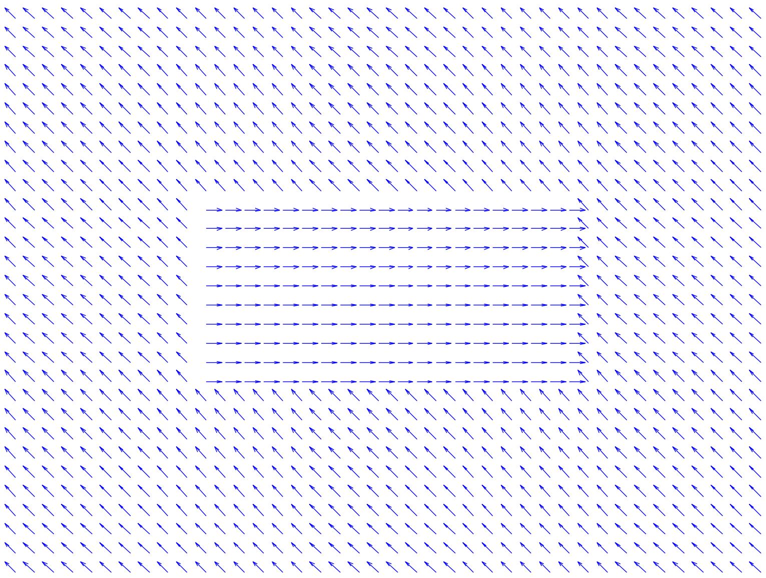


Figure 4.1: Vector representation

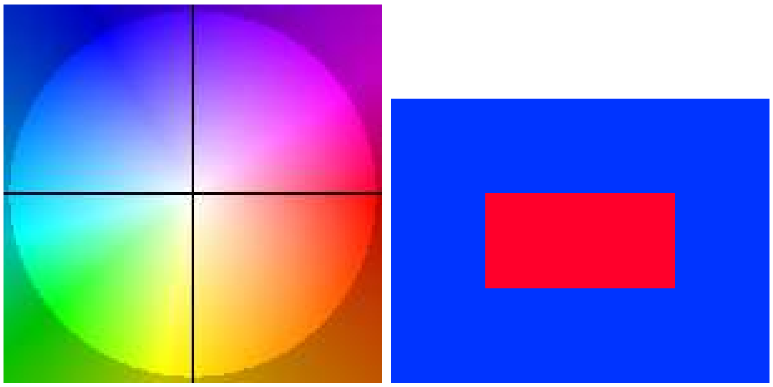

(a) Image test pattern (b) Result with color representation

Figure 4.2: Color representation

the original position the flow is regarded as undefined and is coded as black in the result images.

Figure 4.4(a) shows Middlebury's ground-truth image for the Rubber Whale sequence and 4.4(b) is the result from the Lucas-Kanade algorithm. The result is fairly good, with most error at the area with the circular object. That object has made a rotation between the frames in the source sequence, which is poorly handled in the Lucas-Kanade algorithm.

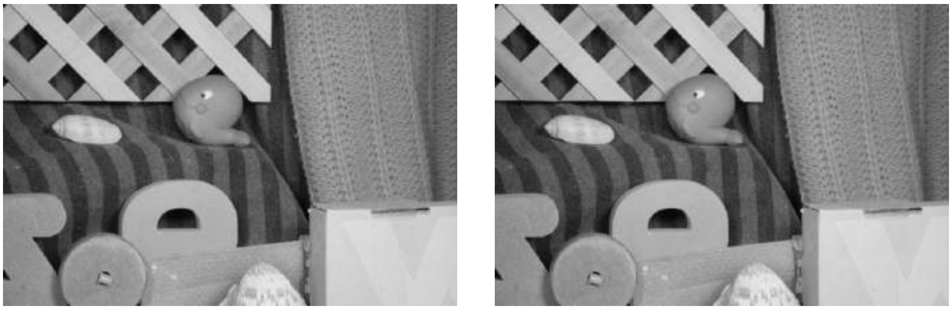
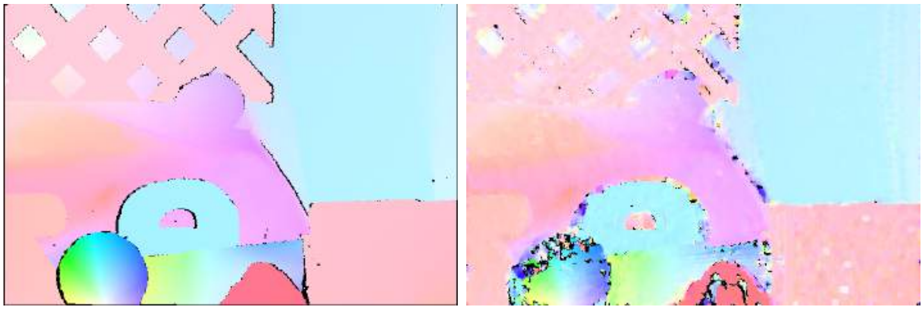


Figure 4.3: Source images from Rubber Whale sequence



(a) Middlebury ground-truth (b) Algorithm result

Figure 4.4: Result of the Rubber Whale sequence

The next test images are shown in figure 4.5 and the result in figure 4.6. The algorithm does not perform as good as the previous test sequence, because there are rather large areas with little texture, which is difficult to track. An improvement using scale pyramids is suggested in section 4.3 which would give better results for this problem.

The algorithm has been tested on a NVIDIA GeForce 8800 GTX graphics card. With a patch size of 8×8 pixels and an image resolution of 584×388 pixels, it took 0.041 seconds per iteration to calculate the dense optical field. An OpenCV version of the algorithm, which is a good CPU implementation, has also been tested with the same parameters. On a Core 2 Duo E6600 2.4 GHz CPU it took 0.67 seconds per iteration, resulting in a 16 times speed up for the GPU.

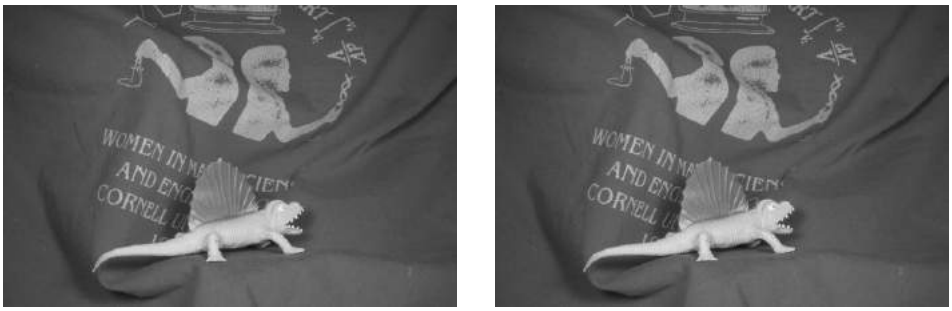
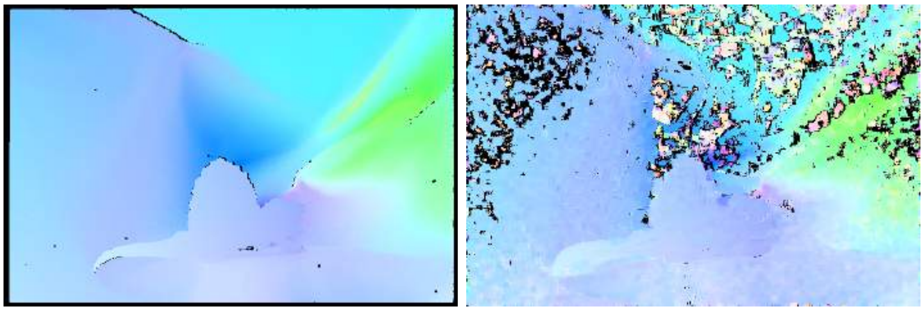


Figure 4.5: Source images in Dimetrodon sequence



(a) Middlebury ground-truth (b) Algorithm result

Figure 4.6: Result of the Dimetrodon sequence

4.2 Discussion

CUDA was made public in 2007 and the newest version is 2.0. In the beginning it was almost just NVIDIA's documentation that could help you get started, but now it is more spread and there are many universities offering courses where CUDA can be learned.

Because CUDA and traditional C-code is very alike it is easy to write programs without any knowledge of how the hardware mechanisms work, but it is much harder to get a fast and efficient program. The GPU is designed so the calculations can be done in parallel and this is the most important property to have in mind while implementing algorithms on a GPU. If the calculations can not be done on many data elements in parallel the program will probably run slower than on a CPU.

It is also important that the algorithm has enough amount of data to process. If it is small, the overhead time will be big compared to the computational time and the data will probably fit well into the CPU's cache if it is computed there.

How to use and handle the memory is a big issue when programming the GPU. The bandwidth between the host and the device is much lower than transactions within the device, so it is a good idea to let the data stay on the device if it is possible. The method of choosing the right memory space were discussed in section 2.6.3, and constant memory is good for constants used by many threads, and texture memory is good when there are some locally random accesses or when the hardware features is meant to be used. When using the global memory, the access pattern is very important to achieve high bandwidth with coalesced transactions. Except trying to access the memory in a good way, it is also important to hide the memory latency with lots of calculations or increasing the number of threads.

Depending on the algorithm's demands on accuracy, there are a couple optimizations that can be done and pit-falls that can be avoided. Since devices with compute capability 1.3 came out, double precision is supported, even though it has a lower performance. If the end result is not affected, or at least acceptable near the original result, single precision can be used instead of double precision. Previously written code designed for devices with lower compute capability may run slower than necessary on devices with higher compute capability, if single precision was not explicitly defined. If double precision functions and variables is used on a device with lower capability it will be demoted to single precision. Apart from using single precision there are also some intrinsic functions that can be used instead of regular functions. These are only supported on the device and execute faster then the regular function but with less accuracy.

One variable in the Lucas-Kanade algorithm is the size of the image patch. It should not be chosen too small because it is hard for the algorithm to find the correct match with a small amount of data and it will probably converge to another part of the image. Also if the patch is too small, noise will have greater influence. On the other hand, if the patch size is to big, the problem with occlusion is more probable to occur, which means that an object or a part of an object gets hidden behind other objects. If the patch is too big, other transformations than translation may be more visible. The sobel operator used in the algorithm gives a blurring effect and is good in a single scale implementation because this reduces some noise, but other filters can be chosen to calculate the gradient. The number of iterations is also a variable, which changes both how long the execution time will be and the result. For some matches the algorithm requires many iterations before it converges and it is difficult if the disparity is large.

One of the choices programmers encounter when using CUDA is how to set up the threads and blocks, the execution parameters. The G80 has 16 multiprocessors so there must be at least 16 blocks just to keep all of them busy. This will however force the multiprocessor to be idle during thread synchronization and during device memory readings if there are not enough threads per block to hide the load latency. The efficiency of the scheduling is reduced and more blocks is better, if resources allow. Each multiprocessor (G80) can take up to 8 blocks and a maximum of 768 threads. If the program is going to have good performance on future devices, the number of blocks need to be high, at least 1000 to scale across several generations. A good program will have about 5000-12000 threads at the same time on the G80 where the maximum number of threads is 12288.

Local variables are assigned to registers which are shared among all the threads in the multiprocessor. The usage of registers can affect the performance of the program, because if all registers are used, fewer threads will be able to run simultaneously. For example, if there are 256 threads per block on a G80, 3 blocks can fit into a multiprocessor with a total of 768 threads. If each thread uses 11 registers the total number of needed registers are 8448. The G80 has however only 8192 registers per multiprocessor and the hardware will reduce the number of threads by removing blocks until the number of needed registers is lower than 8192. This will result in only 2 active blocks and hundreds of threads will be removed. If the number of used registers can be reduced to 10 per thread all the blocks will fit in the multiprocessor. The same problem occurs when exceeding 16 kB shared memory. Newer devices have the double amount of registers but the problem still exists, especially when using double precision.

When trying to optimize a program it is important to identify the bottleneck. Programs are constructed in different ways and the main part of the time can for example be spent on memory transactions, computations in the kernel or instruction overhead, and the optimization effort should be on the parts of the algorithm that take longest to execute. Other things that may reduce performance is bank conflicts and divergent branching. Bank conflicts in shared memory can often be resolved by changing the access pattern. Divergent branching is a result of the SIMT architecture. If the threads diverge because of a conditional branch the execution time will be longer because every path is executed serially and all contribute to the total time. Other threads which are not in the same path are disabled but when all paths are complete the threads converge back to the same execution path. This problem only occurs with threads within a warp, so different warps can execute independently regardless of paths.

4.3 Future work

The development of new GPUs is very fast and it is common that several models are introduced every year. CUDA is designed in a way that current programs will work and perform faster on future devices. More blocks can be processed in parallel since the number of processors increases with newer generations. Except better performance, new properties have been added since the first CUDA-enabled cards such as double precision floating point numbers (since compute capability 1.3) and improved global memory access pattern (since compute capability 1.2).

In some applications there is an advantage in using the texture memory in steps following each other in the algorithm, but currently it is not possible to write in 2D textures. The data has to be written to global memory and then copied to a CUDA array so it can be accessed through texture fetches again. Hopefully these steps can be skipped in the future even though the copying is pretty fast. The warp size may also be changed in the future leading to fewer divergence problems.

The standard Lucas-Kanade algorithm is not suitable for large disparities because of the approximation by omitting higher order terms. This problem can however be solved by using scale pyramids which reduces the image resolution. In the lower resolution the disparity is small and the approximation is good. This approach is also better in the sense of speed because there are less data in the lower resolution images and the disparities gained here are then used as estimates in the higher resolution images, resulting in fewer iterations. If a too high detailed image is used at once, the algorithm may get stuck in a local minimum. Implementing the scale pyramid would be a natural addition and improvement to the algorithm.

The algorithm is based on the assumption that the changes between two image patches occur because of a translation. This assumption is often enough in nearby frames in a sequence, but if there are two distant frames and the patches are scaled, rotated or in another way transformed, the result may be poor. As described in [7] the model can be expanded to handle these kind of transformations by introducing more parameters. There will be more calculations and a bigger equation system but the result might be better depending on the sequence. Shi and Tomasi [28] examined an implementation with affine transformations and saw that the pure translation model gave a more reliable result than the affine model when the translations are small between nearby frames. The affine model is however necessary when comparing distant frames to determine the dissimilarity.

Apart from changing and improving the Lucas-Kanade algorithm, it would also be of interest to investigate other algorithms, both to see differences in the results and in the speed. All algorithms are however not easy to make parallel and are therefore not suitable to implement on a GPU.

Bibliography

[1] "CUDA zone." [Online] Available: http://www.nvidia.com/object/cuda_home.html (2008-09-29).

[2] B. Jähne, *Digital Image Processing (6th revised and extended edition)*. Springer, 2005.

[3] P. E. Danielsson, O. Seger, M. Magnusson, and I. Ragnemalm, *Bildanalys*, 2006.

[4] B. Horn, *Robot Vision.* MIT Press, 1986.

[5] B. Jähne and H. ecker H., *Computer vision and applications: A guide for students and practitioners.* Academic Press, Inc., 2000.

[6] S. Xu, *Motion and Optical Flow in Computer Vision*, 1996.

[7] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, Apr. 1981, pp. 674–679.

[8] B. K. P. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, pp. 185–203, 1981.

[9] M. J. Black and P. Anandan, "The robust estimation of multiple motions: Parametric and piecewise-smooth flow fields," *Computer Vision and Image Understanding*, vol. 63, no. 1, pp. 75–104, 1996.

[10] B. Glocker, N. Komodakis, N. Paragios, G. Tziritas, and N. Navab, "Inter and intra-modal deformable registration: Continuous deformations meet efficient optimal linear programming," in *IPMI*, 2007, pp. 408–420.

[11] R. Fernando and M. Kilgard, *The Cg Tutorial.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[12] "CUDA-enabled GPU products," [Online] Available: http://www.nvidia.com/object/cuda_learn_products.html (2008-09-29).

[13] W. Hwu and D. Kirk, "Lecture 1 - introduction," 2007, [Online] Available: http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html (2008-09-29).

[14] “CUBLAS library,” 2008, available in the CUDA toolkit.

[15] “CUFFT library,” 2008, available in the CUDA toolkit.

[16] “NVIDIA CUDA Compute Unified Device Architecture - Programming Guide (Version 1.1),” 2007.

[17] W. Hwu and D. Kirk, “Lecture 3 - GPU computing and CUDA intro,” 2007, [Online] Available: http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html (2008-09-29).

[18] W. Hwu and D. Kirk, “Lecture 2 - GPU computing and CUDA intro,” 2007, [Online] Available: http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html (2008-09-29).

[19] “NVIDIA CUDA Compute Unified Device Architecture - Programming Guide (Version 2.0),” 2008.

[20] “GeForce 8800,” [Online] Available: http://www.nvidia.com/page/geforce_8800.html (2008-09-29).

[21] “GeForce GTX 280,” [Online] Available: http://www.nvidia.com/object/geforce_gtx_280.html (2008-09-29).

[22] W. Hwu and D. Kirk, “Lecture 7 - GPU compute core,” 2007, [Online] Available: http://courses.ece.uiuc.edu/ece498/al1/Syllabus.html (2008-09-29).

[23] C. Harris and M. Stephens, “A combined corner and edge detector,” in *4th ALVEY Vision Conference*, 1988, pp. 147–151.

[24] V. Podlozhnyuk, *Image Convolution with CUDA*, 2007, available in the CUDA SDK.

[25] M. Harris, *Optimizing Parallel Reduction in CUDA*, 2007, available in the CUDA SDK.

[26] “Middlebury vision,” [Online] Available: http://vision.middlebury.edu/flow/ (2008-09-29).

[27] S. Baker, S. Roth, D. Scharstein, M. Black, J. Lewis, and R. Szeliski, “A database and evaluation methodology for optical flow,” 2007.

[28] J. Shi and C. Tomasi, “Good features to track,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR’94)*, 1994, pp. 593 – 600.