UNIVERSITY OF TROMSØ UIT
FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Image and video processing using graphics hardware

Børge Lanes

Inf-3990
Master's Thesis in Computer Science
May, 2010

Abstract

Graphic Processing Units have during the recent years evolved into inexpensive high-performance many-core computing units. Earlier being accessible only by graphic APIs, new hardware architectures and programming tools have made it possible to program these devices using arbitrary data types and standard languages like C.

This thesis investigates the development process and performance of image and video processing algorithms on graphic processing units, regardless of vendors. The tool used for programming the graphic processing units is OpenCL, a relatively new specification for heterogenous computing. Two image algorithms are investigated, bilateral filter and histogram. In addition, an attempt have been tried to make a template-based solution for generation and auto-optimalization of device code, but this approach seemed to have some shortcomings to be usable enough at this time.

Acknowledgements

I would like to thank the following people:

- At Tandberg ASA:
  - My supervisor Jan Tore Korneliussen for incredible insight in all aspects regarding this thesis and being a good host during my visits to Lysaker.
  - Knut Inge Hvidsten for initial dialogues.
  - Odd Arild Skaflestad for approving this cooperation.
  - Members of the Movi team for gstreamer expertize.
  - Håvard Graff, my good friend that made first contact possible and having me as guest during my trips to Oslo.
- At the University of Tromsø:
  - My supervisor John Markus Bjørndalen for good feedbacks and discussions during the writing and finalization of this thesis.
  - The always service-oriented Jan Fuglesteg.
  - Joakim Simonsson and Tor-Magne Stien Hagen for help and input on GPU and OpenGL issues.
  - Christer André Hansen and his cat for good test photographs.
- My daughter Ragna, that has forced me to change focus on a daily basis.
- My extended family for good help, moral support and dinners during this year.

And finally I would like to thank my dear Hanne for her patience and confidence in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The computing world is moving towards parallel computing and hardware, not only for computational demanding areas like weather forecasts but also to applications that exist on consumer handheld devices. Conventional central processing units (CPU) are reaching the limits for how high the clock frequencies can go, and the current trend is to increase the number of cores on each CPU to achieve higher computational throughput. For developers of operating systems and their applications, utilizing all available cores in the best way possible is not a trivial task, and during the last years there has been an increasing number of tools and frameworks intended for easier development of parallel code. This goes for both shared-memory systems like multi-core processors, where there are tools like Intel Parallel Studio[1], but also for commodity hardware clusters, like Google's Mapreduce [15] and its derivatives.

As the CPU has been having trouble keeping its clock frequency growing and have added more cores to compensate for this, the graphical processing units (GPU), which have always been parallel hardware made for real-time 3D renderings, have evolved into devices that may also be programmed and used for general-purpose computing like scientific calculations.

Being driven by the home gaming market means that GPUs have become commodity hardware and has resulted in high availability and a very low price compared to other specialized pieces of hardware. Thus a GPU can be seen as a very cheap parallel supercomputer, and scientists have been interested in finding out how this processing power can be harnessed. The graphic rendering pipeline have gone from a fixed-function to a more programmable pipeline, at first allowing developers to customize rendering behaviour, but also being utilized by programmers for doing general computations. As GPU hardware has evolved, so has the software solutions and programming APIs.

The purpose of this thesis is to investigate image and video processing techniques and programmability on the GPU. Issues like video encoding is not in-

[1] `http://software.intel.com/en-us/intel-parallel-studio-home/`

vestigated, the focus is raw image and video frames. We will also look into the current advantages and disadvantages of utilizing the GPU for this domain, since the graphic processing techniques originally intended for the GPUs are not equivalent to the algorithms and techniques utilized for image and video processing.

1.2 Problem definition

This thesis goal is to explore some image processing algorithms and how they behave on the GPU versus a traditional CPU implementation, regarding both performance and the implementation process. The image processing algorithms used must be a selection that uses different memory access patterns like local neighbourhood iterations, reduction and scatter/gather.

We emphasize programmability of GPUs regardless of vendors. This means that there must be taken different code approaches to get optimal performance on different hardware. To achieve the independence of different vendors and for best and easiest programmability this thesis have focused on OpenCL as the programming framework for GPU computing.

1.3 Project context

This thesis is written in collaboration with Tandberg ASA.

1.4 Outline

The rest of the thesis is organized as follows. Chapter 2 describes the background and motivation for general-purpose GPU programming along with current programming techniques. Chapter 3 contains image processing concepts and definitions of the implemented algorithms. Chapter 4 gives a brief introduction to OpenCL concepts needed to understand the rest of this thesis' work. The test systems are listed in Chapter 5 along with the measurement techniques used, followed by Chapter 6 containing the algorithm implementation descriptions. The development process of GPUScribe, a template-based code generation system for GPU code is described in Chapter 7. Chapter 8 contains related work followed by discussion in Chapter 9. In Chapter 10 we conclude our work.

Chapter 2

General-purpose GPU programming

2.1 Overview

The original purpose of the graphic processing units has been to increase performance of real-time 3D rendering. During the recent years, GPUs have had a remarkable increase in processing power measured in raw computational throughput. This has been motivated by a market that have needed ever faster GPUs for many purposes like gaming and real-time visualizations of new buildings or city plans. The performance increase have been huge with CPUs too, but the raw computing capabilities have increased most on the GPU. Figure 2.1 and 2.2 shows the recent evolution of computing capabilities in gigaflops and memory bandwidths of the CPU and GPU.

Figure 2.3 shows the transistor layout of a modern CPU and GPU. The actual number of transistors on these two chips are not very different. The main difference is that most of the GPU transistors are used for strict computation while much of the CPU transistors are used for caching, branch prediction, out-of-order execution optimalization during runtime and other things that has become part of the optimizing pipeline for a general-purpose processor. The reason to this evolution is that the graphic processing units have evolved to do one thing well, and that is to run the needed graphic rendering pipeline at high throughput, containing many parallel operations that are totally independent of each other. Thus, one can say that the GPU excels at data-parallel problems.

Much material and resources for general-purpose gpu computing can be found on the website `gpgpu.org`[1]. As introduction material they recommend some survey articles [32, 33] and some other articles for information on programming models and the underlying GPU hardware [16, 25]. Another good resource for understanding GPU technologies is found in [24].

Currently there are two dominating hardware vendors in the field of general-purpose GPU computing – nVidia and AMD. Of these two, nVidia has had the

[1] Last visited May 2010

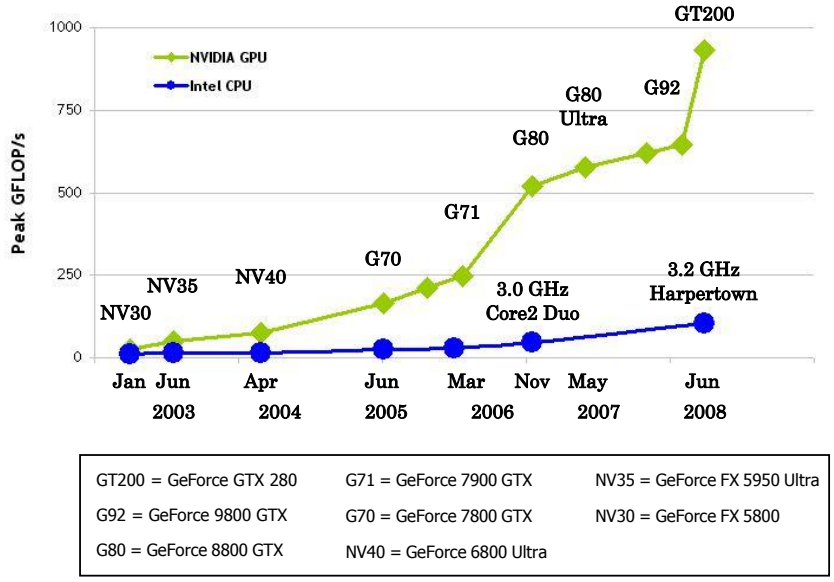


Figure 2.1: CPU and GPU compute capabilities. nVidia corporation [29]

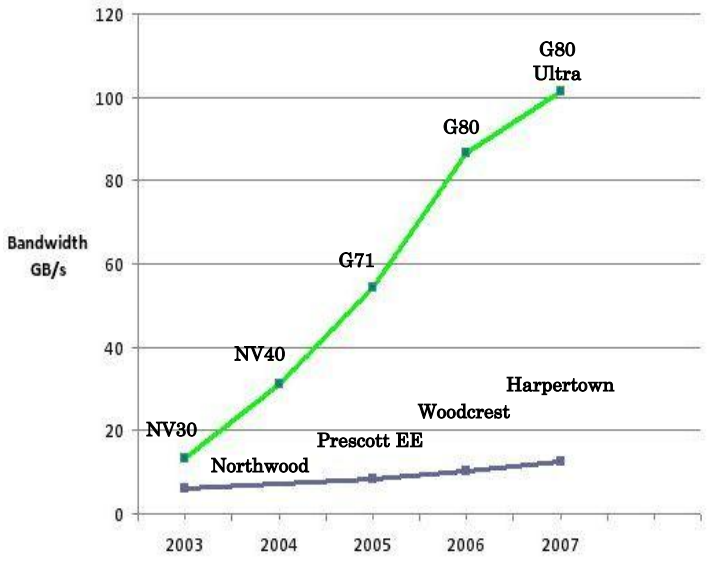


Figure 2.2: CPU and GPU memory bandwidth. nVidia corporation [29]

strongest focus on gpgpu issues when developing their hardware and software, that has lead to their success of CUDA (see Section 2.3.1) for gpgpu computing. nVidia have also had the best drivers available for systems other than Microsoft Windows. AMD, formerly known as ATI before they was acquired by AMD, has

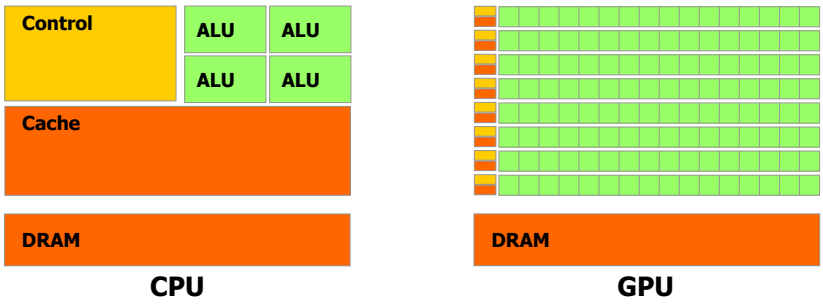


Figure 2.3: CPU and GPU transistor layouts. nVidia corporation [29]

had its hardware development focus primarily on gaming for the Windows platform. They are also into the gpgpu market with their ATI Stream Technology, targeting both CPU and GPU development for Windows and Linux platforms.

2.2 Limitations

The gpu is a specialized, highly parallel architecture. It is however not suitable for all parallel programming techniques, like setting up a pipelined workflow or using parallel primitives like a producer-consumer queue. In addition, debugging GPU code can be hard or require some special techniques.

One can say that the term *general-purpose GPU computing* in itself is somewhat misleading, since the GPU is not very well suited to do general-purpose computing. For an algorithm to run efficiently on a GPU versus on the CPU, it must be data-parallel as it is, or be rewritten to a data-parallel version. The original quicksort algorithm is for instance not suited for the GPU, but has been made into a version, *GPU-Quicksort* [13], suited for GPU computing.

2.3 Programming techniques

Until recently it has not been that easy to utilize the power of the GPU for general-purpose programming issues. It started with the programmable shaders that could be customized by graphic developers to alter steps in the default rendering pipeline, at first vertex and fragment shaders. This was also used for general-purpose programming, but then the input data had to be camouflaged as graphics and the final output data was also represented as graphics. nVidia released their CUDA SDK (see Section 2.3.1) in early 2007 that allowed a general-purpose C-style programming of their hardware.

Below are given descriptions of some of the tools currently available with short descriptions. Figure 2.4 shows the current techniques and where they are in the landscape considering level of abstraction and portability between different devices and/or vendors. Note that HLSL and DirectCompute (both part

of Microsoft DirectX) are only available for the Microsoft Windows operating system. The other systems are available to more platforms, see Table 2.1 for details. GLSL can also be used on other X window systems besides Linux. nVidia has drivers available for FreeBSD and Solaris.

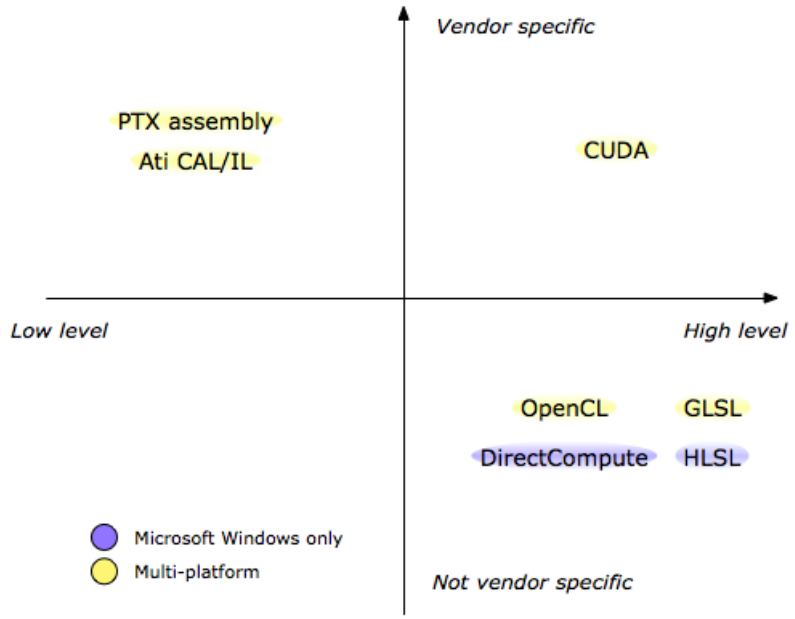


Figure 2.4: Current GPGPU programming languages

Table 2.1: Platforms supported by gpgpu tools

Tools	Microsoft Windows	Linux	Mac OS X
CUDA, PTX Assembly	X	X	X
OpenCL	X	X	X
Ati CAL, Ati IL	X	X	
GLSL	X	X	X
HLSL	X		

All these programming tools are rather similar, meaning that when one is familiar with one of them and have gotten the head around programming in a very data-parallel fashion, it is not hard to understand and use the others. An exception may be when a graphic programming framework, like OpenGL or Direct3D, is used for gpgpu computing, since they have quite different programming styles[2] and everything happens in a graphic context. In Figure 2.4, GLSL and HLSL are placed at high-level tools since the GPU code written using these languages is device agnostic, hence not needing detailed knowledge of the host GPU.

[2]OpenGL is a state machine and Direct3D is not.

The tools intended for gpgpu programming consist of host code running on CPU and separate code snippets that are compiled for the current system's GPU. The host code orchestrates allocation and deallocation of memory buffers on the GPU, executing programs and copying data to and from the device. None of these tools are really high-level, and none of them is as easy to program as making a standard C program on the CPU. There are however frameworks, utilizing tools like CUDA and OpenCL, giving higher abstractions at the cost of fine-grained optimalization. More details and comparisons on the basic programming model is found in Chapter 4. Descriptions of some of the higher level tools available are found in Chapter 8.

2.3.1 CUDA

CUDA is nVidia's proprietary system for doing gpgpu computation and it has been a great success and has been put to use for many different scientific diciplines. Reasons for this may be among others:

- It was the first high-level programming environment using a familiar, C-style syntax for code that should run on the device itself, supporting arbitrary data types.
- It provides good debugging tools (see Section 4 for details).
- Every CUDA device has a *CUDA compute capability* number, giving developers information about what exactly the device can do and how to best optimize the code for the device available.

For more details on CUDA, see [29].

2.3.2 Ati CAL

CAL is short for *Compute Abstraction Layer*. This is AMD's proprietary system for doing gpgpu computations on their GPU hardware. The host code is equivalent to the one used in CUDA, but the device code must be written in model-specific assembly or using the ATI Intermediate Language (see Section 2.3.5). See [19] for more details about Ati CAL.

2.3.3 OpenCL

This is an open and royalty-free standard for performing heterogenous computing. It was initiated by Apple and is now maintained by the Khronos group. This is the main tool used in this thesis for programming GPUs so it is given a more throughout description in Section 4. The programming models of OpenCL and CUDA are quite similar, except that OpenCL also has support for task parallel programming and is made for running on many different kinds of devices, not just GPUs. Its specification can be found in [27].

2.3.4 DirectCompute

This is part of Microsoft's DirectX 11 API, and is available only on the Windows platform. It allows gpgpu programming on all DirectX 10 and 11 capable devices.

2.3.5 Other

The shader languages GLSL and HLSL are also used for gpgpu computing. GLSL was introduced as a core part of OpenGL version 2.0, and is working on many devices and operating systems. It has been much used for gpgpu computing, but this may be less common with the release of OpenCL. HLSL is equivalent to GLSL, but being part of Microsoft Direct3D makes it only available for the Microsoft Windows platform.

Both nVidia and AMD have their own low-level, intermediate gpgpu languages, respectively PTX assembly [31] and Ati IL (intermediate language) [18]. None of these represent actual opcodes on GPU hardware. The benefit of having these languages is easier compiling from high-level languages for different generations of devices. The actual translation from PTX assembly to GPU machine code is probably done by the graphics driver, but this issue is not documented by nVidia.

Chapter 3

Image Processing

3.1 Introduction

Image processing issues discussed in this thesis is on uncompressed frames only. This means that we are not interested in optimizing things like jpeg image encoding and decoding, and in the case with video we do not investigate things like the H.264 video encoding standard. We limit the scope to the processing of raw pixel data, preferably at such speed that we can use the techniques for real-time video processing at high frame rates. Examples of such filter applications are

- Color transformations, like sepia and grayscale.
- Sharpen and blur effects.
- Dynamic range processing.
- Removing transform coding artifacts, also called deblocking.
- Image statistics, like histogram and median value.

We are also interested in interframe processing algorithms, meaning algorithms that uses several sequential video frames to produce a single output frame. This is for instance done in video encoding, which is outside the scope of this thesis, but can be used for other problems like noise removal, as shown in [11].

From [22] we can read:

> *FFT, convolution, and histogram are three important kernels that form the building blocks of many image processing applications. Optimizing these kernels will provide performance benefit to all applications that utilize them.*

FFT (Fast Fourier transform) is kept outside this thesis' scope, but we will investigate and implement histogram and bilateral filters, described in Section 3.3.

Bilateral filters are by definition not convolutions, but stencils which is a superset of convolutions.

An in-depth introduction to the area of image processing can be found in [6]. Note that these slides are intended for signal processing students and require some mathematical knowledge like fourier transformations and complex analysis. It is also a benefit being MATLAB proficient to fully understand all examples.

3.2 Image representations and color spaces

In this thesis, images pixels are represented by uncompressed RGBA or RGB pixel values. The symbol A denotes the alpha channel, used for describing a pixel's transparency. This can be used when having several layers of images on top of each other, making them blend correctly. It can also be used to exclude parts of an image from being visible, for making special effects. In this thesis' work the alpha channel is not being considered and images are always fully visible.

The RGB color space is a linear, additive, device-dependent color space. Each value is usually represented as unsigned integers in the range from 0 to 255, giving a total color depth of $3 \cdot 8 = 24$ bits. Different color examples are shown in Table 3.1. Being a device-dependent color space means that a specific color will not look identical on all monitors or printers, and the percepction of the color may vary depending on background lightning.

An example of a device-independent color space is sRGB, described in [39].

Table 3.1: RGB color examples

Color	**Red**	**Green**	**Blue**
Black	0	0	0
White	255	255	255
Yellow	255	255	0
Dark green	0	100	0

Color space transformations done related to work in this thesis is done by external libraries. Interpretation of jpg image files into RGBA arrays are for instance done by Mac OS X libraries. When reading from video formats, the needed transformations are done by gstreamer plugins. Compression standards for video and image data tend to use color spaces that are better suited for compression, based on human perception and efficient encoding and decoding, like $\mathrm{YC_BC_R}$.

One issue related to color perception done manually in this thesis is finding a pixel's intensity based on its RGB values. The intensity I is calculated by

$$I = 0.3R + 0.59G + 0.11B \tag{3.1}$$

This is the same method used for calculating the Y value when converting from RGB to YC_BC_R. Note that the weights have a total sum of 1.

3.3 Image processing algorithms

To test different techniques, some filters have been examined in detail. Below are descriptions of these filters and what makes them interesting for gpgpu experimentation.

3.3.1 Bilateral filter

Basic definition

The bilateral filter was originally introduced in [41]. Exploration of bilateral filters was chosen because they are based on local neighbourhood calculations and have many application areas. The bilateral filter is by definition not a convolution, as wanted in Section 3.1, but a stencil, that is a superset of convolutions. They are however both using local neighbourhood calculations.

A nice introduction to bilateral filters is found in [36]. To understand bilateral filters it is best to know the basic gaussian blur. A gaussian blur filtered image, GB, is defined below. Equations are from [36]. The notation $I_{\mathbf{p}}$ means the image value at position $\mathbf{p}$. The entire spatial domain of the image is denoted by S, meaning every possible pixel position. Spatial distance between pixel positions $\mathbf{p}$ and $\mathbf{q}$ is given by $\|\mathbf{p} - \mathbf{q}\|$.

$$GB[I]_{\mathbf{p}} = \sum_{\mathbf{q} \in S} G_\sigma(\|\mathbf{p} - \mathbf{q}\|) I_{\mathbf{q}} \tag{3.2}$$

where G is the gaussian kernel:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \qquad \text{1 dimension} \tag{3.3}$$

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \qquad \text{2 dimensions} \tag{3.4}$$

Since the area of interest is 2-dimensional images, equation 3.4 is used as the gauss kernel in this context. The gaussian blur gives a smooth blur effect without any disturbing artifacts.

The difference from bilateral filtering to gaussian blur is that the bilateral filter also take into account the intensity of each of the surrounding pixels when calculating the current pixel's weight. The result is that we have a filter that in practice is a gaussian blur with edge detection. Note that the image intensity threshold can be adjusted on the filter, say be set so large that the bilateral filter behaves just like the gaussian blur.

The definition of the bilateral filter, BF, is

$$BF[I]_{\mathbf{p}} = \frac{1}{W} \sum_{\mathbf{q} \in S} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|) I_{\mathbf{q}} \tag{3.5}$$

where W is a normalization factor, given by

$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in S} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|) \tag{3.6}$$

In this case, $|I_{\mathbf{p}} - I_{\mathbf{q}}|$ denotes the intensity difference between the pixels at positions $\mathbf{p}$ and $\mathbf{q}$. The function G_{σ_s} is given by Equation 3.3 and G_{σ_r} is given by Equation 3.4.

It is common practice to ignore neighbour pixels that are more than 2 spatial standard deviances away since their weight become negligible, thus the filter kernel size is directly dependent on the spatial standard deviance (σ_s). The algorithm definitions are originally meant for gray scale images, but can be used on color images too by either running the algorithms separately on each color channel or calculating each pixel's intensity using Equation 3.1. How the filter was actually implemented to run on the GPU is explained in Section 6.2.1.

An example of the effect of gaussian blur and bilateral filter is shown in Figure 3.1. Notice that the bilateral filter example (Figure 3.1c) is a mostly blurred image, but edges like the whiskers and eye of the cat are preserved. Changing the values of σ_r and σ_s will give different results.

Applications

The bilateral filter has many applications (from [36]):

- Denoising
- Texture and illumination separation, tone mapping, retinex and tone management
- Data fusion
- Three-dimensional fairing

Variants and optimizations

The bilateral filter defined above is known as the *brute-force* variant. A disadvantage of this implementation is that the algorithm order has complexity $O(n^2)$, dependent on the spatial standard deviation σ_s.

Because of its many applications there has been done much research on alternatives for implementing the bilateral filter and other edge detection algorithms [7, 8, 14, 35, 37, 42, 43]. These approaches are interesting, because they introduce data structures and concepts useful for image processing applications that are not so easily mapped to the current memory- and programming models of graphics hardware. Examples are the use of KD-trees [8], bilateral grid [14] and permutohedral lattice [7]. Implementing a tree structure on GPU is currently a challenge, since there is no such thing as C's `malloc` function for dynamic memory allocation during runtime and the GPU hardware is not very efficient at accessing small data-structures that are not aligned sequentially in its global memory. There has been done research on issues like real-time kd-tree generation [44] and parallel hashing [9] performed on graphics hardware.

3.3.2 Histogram

Definition

```
int h64[64];
int h256[256]; // all array values must initially be set to 0

for (pixel in image) {
    i = intensity(pixel);
    h64[i >> 2] += 1;
    h256[i] += 1;
}
```

Listing 3.1: Histogram examples

Listing 3.1 shows a pseudocode implementation of two image histograms with 64 and 256 bins. This implementation assumes that the `intensity` function in the code returns a value in the range 0,255. Notice that to get the correct bin counter increased in the 64-bin histogram, the intensity value is divided by 4 or bitshifted 2 positions to the right. This code example only uses the intensity of a pixel, but it is also possible to have separate histograms for each of the three RGB color channels.

In short, the histogram is a statistic collection keeping the occurency count of each colors intensity level, or specific ranges of intensity levels. How the histogram was implemented on the GPU is shown in Section 6.3.1.

Applications

When working with digital images, the histograms are very valuable image statistics that can give information like amount of contrast and whether a photograph is under- or overexposed. When applied on a video stream from a camera, the histogram can given information about needed compensations for brightness and contrast.

(a) Original image

(b) Gaussian blur

(c) Bilateral filter

Figure 3.1: Example of gaussian blur and bilateral filter

Chapter 4

OpenCL

4.1 Overview

OpenCL is a framework made for doing *heterogenous* computing across different devices like CPUs, GPUs and other hardware. It was initiated by Apple and the specification and header files was released Desember 2008 [5]. The specification has had some updates since then, see [27] for the currently latest one. The OpenCL specification is maintained by the Khronos Group[1], an industry consortium that also maintain other open standards like OpenGL and WebGL.

Although GPU computing is the most emphasized usage of OpenCL, it is unlike the other tools described in Section 2.3 not only intended for GPU computing. It can be used to orchestrate and run code for CPU, GPU and also other devices that supports the OpenCL specification. Examples are the IBM Cell processor and Intel Larrabee. The OpenCL specification supports both data-parallel and task-parallel programming paradigms. This thesis focuses on GPU computing, that is suited for the data-parallel paradigm, but the task-parallel features of OpenCL can be used nonetheless, as explained in Chapter 9.

The first implementation made public was as part of Apple's new operating system Snow Leopard, being released August 2009. Since then, other major vendors like nVidia and AMD has made OpenCL a part of their gpgpu programming frameworks for general-purpose GPU computing.

The following parts og this chapter describes the parts of OpenCL being necessary to understand the remaining chapters of this thesis. The specification document for OpenCL is 308 pages, so this is a brief description of the concepts and programming models it supports. Being only a specification, it is the hardware vendor's responsibility to provide correct OpenCL support for their devices according to the specification.

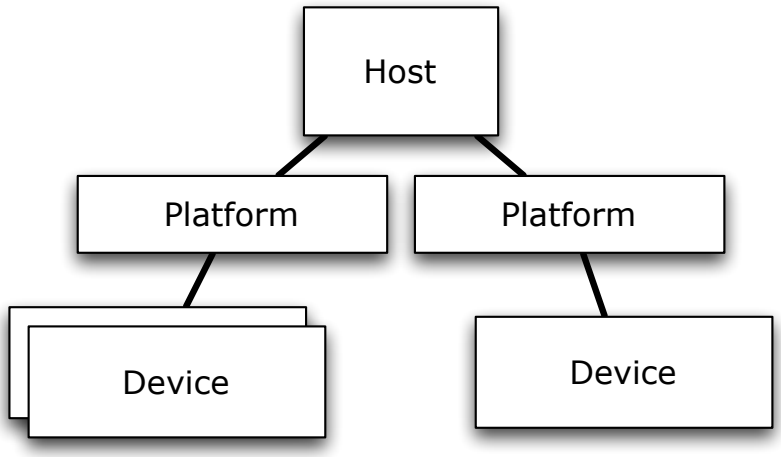


Figure 4.1: OpenCL architecture

4.2 Architecture

Figure 4.1 shows the architecture of OpenCL. The host is the entity that controls and orchestrates everything, using OpenCL API calls. If the current system has several OpenCL implementations available, like if using both an nVidia and an AMD card, these implementations are represented as different platforms. This was not a part of the initial specification, but became a part of it at a later time and is now supported by both nVidia and AMD's implementations. Before this standard was implemented, the host could ask for the current system's GPU devices directly, but now it has to query each platform respectively about its devices.

4.3 OpenCL devices

The architecture of an OpenCL device is shown in figure 4.2. It has a finite number of compute units with 1 or more processing elements and each compute unit has its own memory, only visible to the processing elements.

In Listing 4.1 is given the output of nVidia's device query example that is bundled with the CUDA developer SDK. This shows that this device has 30 compute units. It also shows information about nVidia-only attributes like the number of CUDA cores, telling us that this device has $\frac{240}{30} = 8$ processing elements for each compute unit.

[1] `http://www.khronos.org/`

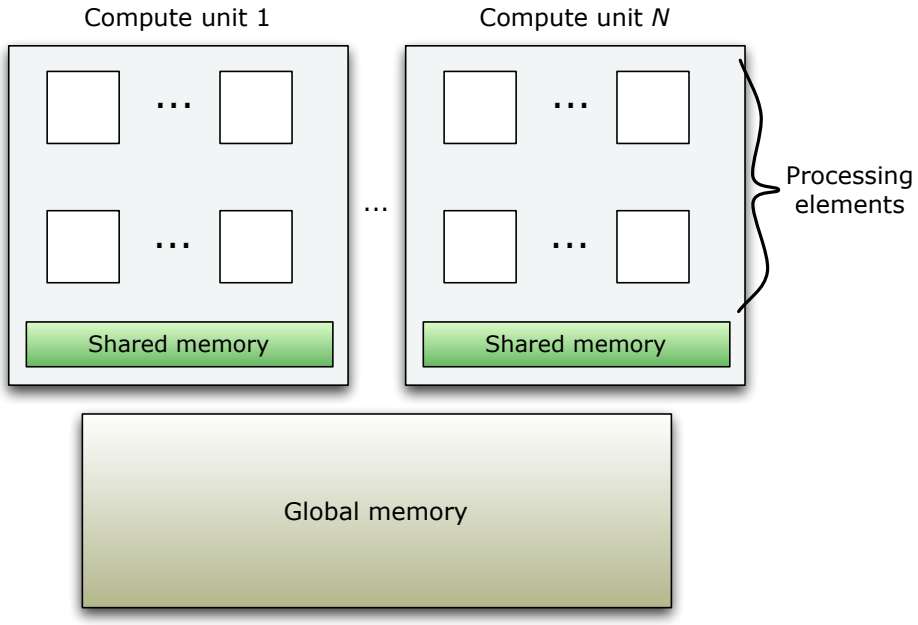


Figure 4.2: Conceptual view of an OpenCL device

```
-------------------------------------
Device GeForce GTX 295
-------------------------------------
CL_DEVICE_NAME:          GeForce GTX 295
CL_DEVICE_VENDOR:          NVIDIA Corporation
CL_DRIVER_VERSION:         195.36.15
CL_DEVICE_TYPE:        CL_DEVICE_TYPE_GPU
CL_DEVICE_MAX_COMPUTE_UNITS:      30
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: 3
CL_DEVICE_MAX_WORK_ITEM_SIZES:    512 / 512 / 64
CL_DEVICE_MAX_WORK_GROUP_SIZE:    512
CL_DEVICE_MAX_CLOCK_FREQUENCY:    1242 MHz
CL_DEVICE_ADDRESS_BITS:     32
CL_DEVICE_MAX_MEM_ALLOC_SIZE:     223 MByte
CL_DEVICE_GLOBAL_MEM_SIZE:      895 MByte
CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
CL_DEVICE_LOCAL_MEM_TYPE:     local
CL_DEVICE_LOCAL_MEM_SIZE:     16 KByte
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
CL_DEVICE_QUEUE_PROPERTIES:
     CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
CL_DEVICE_QUEUE_PROPERTIES:     CL_QUEUE_PROFILING_ENABLE
CL_DEVICE_IMAGE_SUPPORT:      1
CL_DEVICE_MAX_READ_IMAGE_ARGS:    128
CL_DEVICE_MAX_WRITE_IMAGE_ARGS: 8
CL_DEVICE_SINGLE_FP_CONFIG:     INF-quietNaNs round-to-nearest
     round-to-zero round-to-inf fma

CL_DEVICE_IMAGE <dim>       2D_MAX_WIDTH    8192
          2D_MAX_HEIGHT   8192
```

```
            3D_MAX_WIDTH    2048
            3D_MAX_HEIGHT   2048
            3D_MAX_DEPTH    2048

  CL_DEVICE_EXTENSIONS:      cl_khr_byte_addressable_store
            cl_khr_icd
            cl_khr_gl_sharing
            cl_nv_compiler_options
            cl_nv_device_attribute_query
            cl_nv_pragma_unroll
            cl_khr_global_int32_base_atomics
            cl_khr_global_int32_extended_atomics
            cl_khr_local_int32_base_atomics
            cl_khr_local_int32_extended_atomics
            cl_khr_fp64


  CL_DEVICE_COMPUTE_CAPABILITY_NV:   1.3
  NUMBER OF MULTIPROCESSORS:      30
  NUMBER OF CUDA CORES:       240
  CL_DEVICE_REGISTERS_PER_BLOCK_NV: 16384
  CL_DEVICE_WARP_SIZE_NV:     32
  CL_DEVICE_GPU_OVERLAP_NV:     CL_TRUE
  CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV: CL_TRUE
  CL_DEVICE_INTEGRATED_MEMORY_NV: CL_FALSE
  CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>  CHAR 1, SHORT 1, INT 1,
      LONG 1, FLOAT 1, DOUBLE 1
```

Listing 4.1: nVidia GTX 295 device information

4.4 Programming model

To execute programs on the GPU, a source file containing OpenCL device code must be compiled with a just-in-time compiler using the host APIs, similar to the technique used by GLSL. This program is called a kernel. To run it in the GPU, a n-dimensional kernel has to be invoked, and this can be 1, 2 or 3 dimensional. Visualizations of the kernel dimensions are shown in Figure 4.3, 4.4 and 4.5. A kernel consists of work items, equivalent to threads, divided in work groups.

The OpenCL programming model supports both task- and data parallelism. Example of a data-parallel OpenCL kernel is shown in Listing 4.2. This code is run by every thread. The `get_global_id(0)` function tells this work item its index in the first dimension, making it address the correct input and output values.

```
__kernel void (__global float *input_values, __global float *
    output_values)
{
    int i = get_global_id(0);
    output_values[i] = input_values[i] * 2;
}
```

Listing 4.2: OpenCL basic kernel

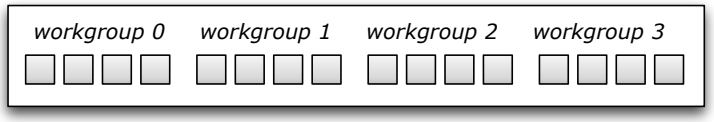


Figure 4.3: 1-dimensional NDKernel

Note that it is not possible to dynamically allocate memory in an executing kernel. All kernel memory allocations are statically allocated at compile time.

4.5 Data types and functions

The common datatypes from the C language are supported. In addition they are available in vector version of length 2, 4, 8 and 16.

Many of the functions in the OpenCL programming language work with different vector lengths, or just scalar variables.

Using the vector datatypes can make the code more readable, but it it also good to achieve optimization. If the code is compiled with x86 as target device, this can result in code using the SSE or new AVX instruction set in the machine code. The ATI optimization guide [21] says programmers should use `float4` instead of `float` on code for both the CPU and GPU.

4.6 Optimalization issues

Both AMD and nVidia have released guides on how to optimize OpenCL code for their GPUs [20, 21, 28].

Floating point computational throughput is the great strength of GPUs, and is preferred over integer arithmetics. There are however hardware accelerated integer functions like `mad24(a,b,c)` that calculates a*b+c, where a*b is 24-bit integer division.

The x86 architecture implicitly does a great deal of hardware optimalization through branch predictions, out-of-order executions and utilization of its cache hiearchy. Knowledge of the cacheline length can be a benefit for the developer for instance when manually optimizing multithreaded code, so many concurrent writes to the same cachelines are avoided.

When programming for graphic processing units, much utilization of the memory hiearchy must be done manually by developer. One thing is correct placement of data to the correct memory locations, but also important with both nVidia and AMD is using optimal methods for writing and reading to and from both local and global memory. How to do this is explained in the programming guides from Ati and nVidia [20, 30].

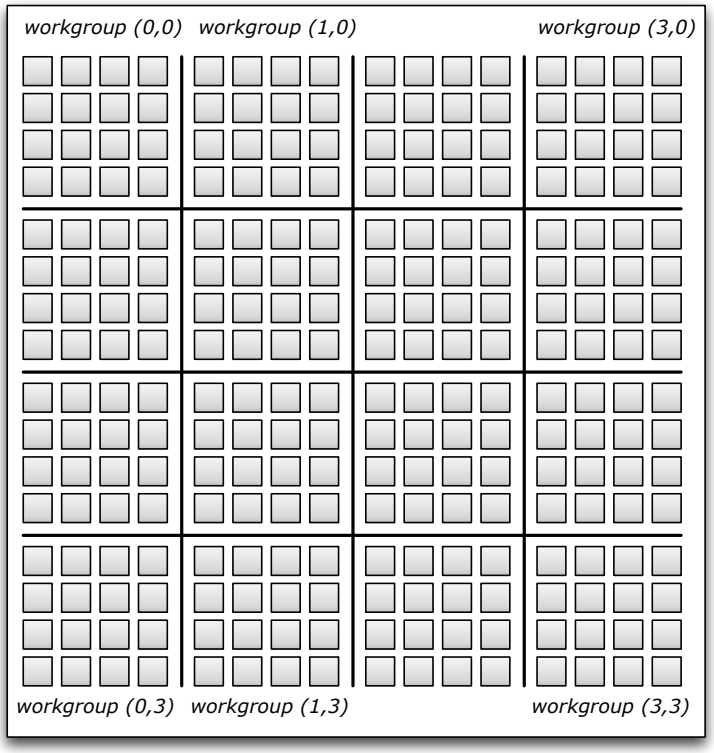


Figure 4.4: 2-dimensional NDKernel

4.7 Image processing techniques

When using OpenCL the developer has the choice of representing images plain C-style arrays or using OpenCL's `image2d_t` datatype. There are benefits of using the `image2d` datatype for the purpose of image processing:

- Image reads are optimized for 2-dimensional spatial locality.
- The image datatype handles what is returned when reads happen outside of the image bounds.
- Images are stored in the GPU's global texture memory, meaning that it also utilizes the texture cache.

As can be seen from this list, many of these benefits are implicitly using features of the GPU hardware that are not possible to use with C-style arrays, like the texture cache. The benefits of having images in C-style arrays are:

- No need to use special functions to access pixels.
- Can represent pixels using arbitrary color spaces.

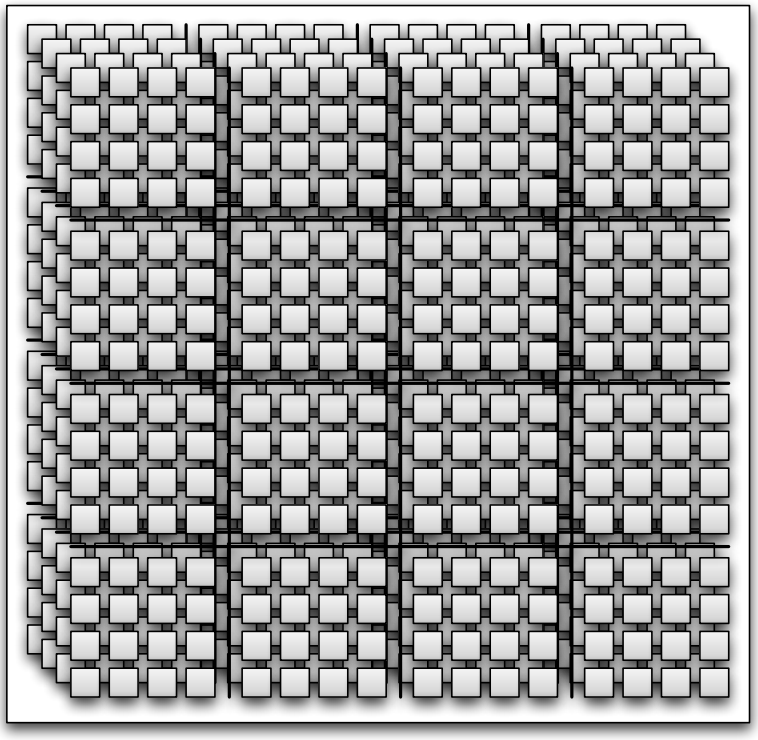


Figure 4.5: 3-dimensional NDKernel

- `Image` support is not always available for any given device

The color spaces supported by OpenCL are only linear RGB variants that are device dependent, but it is of course possible for vendors to make extensions that support more color spaces.

Chapter 5

Methodology

5.1 Test systems

During the work on this thesis, three different test systems have been used for development and testing, shown in Table 5.1, 5.2 and 5.3. Test system I is a MacBook , and test systems II and III are desktop workstations.

Table 5.1: Test system I

CPU	Intel Core 2 Duo @ 2 GHz
Operating System	OS X Snow Leopard
System Memory	4 GB
GPU	nVidia GeForce 9400m

Table 5.2: Test system II

CPU	Intel Penium 4 @ 3.2 GHz
Operating System	Ubuntu Linux 9.04 64-bit
System Memory	1 GB
GPU	nVidia GTX 295

Table 5.3: Test system III

CPU	Intel Core 2 Duo @ 2.66 GHz
Operating System	Ubuntu Linux 9.04 64-bit
System Memory	6 GB
GPU	nVidia GeForce 88000GT, ATI Radeon HD 5850

The theoretical memory bandwidths of the different GPUs are shown in Table 5.4. This bandwidth tells us how fast the cards can move data internally

and can be of help while optimizing kernels. The GTX 295 card has two identical chips, hence the 2x before the bandwidth number. The memory bandwidths when transferring data from host to the GPUs over the PCI bus is not a part of this number. When utilizing the GPUs, the PCI data transfer is an expensive process and should be kept at a minimum. It is however possibly, as long as the driver supports it, to move data asynchronously over the PCI bus as a kernel executes. The nVidia 9400m shares its memory with the host, and that may be the reason to why no memory bandwidths number was given. These numbers are collected from the specification websites of the various cards.

Table 5.4: GPU memory bandwidths

GPU	**Memory bandwidht (GB/sec)**
nVidia GTX 295	2x 111.9
nVidia 8800 GT	57.6
nVidia 9400m	?
ATI Radeon HD 5850	128

It was not physically possible to have the nVidia GeForce 8800 GT and AMD Radeon HD 5850 cards in test system III's cabinet at the same time. The ATI card was not available for use until february 2010. Initial tests on system III were done on the nVidia GeForce 8800 GT and later it was replaced with the ATI card. Therefore some test data is not available for the 8800 GT card.

5.2 Measurement technique

This thesis' focus is the development process for optimizing GPU kernels, so the main interest for measurements is the execution time of individual kernels. The measurements were done by using the profiling option in OpenCL, which can return 4 different timestamps for every event that is placed in a profiling-enabled queue. These are when the event was placed in a queue, when it was submitted to the device, when the execution started and when the execution finished. We only used the timestamps for when the execution started and finished. Initially, some test loops were run to see if these values could be trusted, and it seemed so. It was probably a benefit using this technique since the test systems were so different, regarding CPU architectures and mainboard characteristics.

When a kernel using approximately 5 milliseconds is run on an idle machine, the results of several simulatenous runs with identical input data is +- 100 microseconds. Since this variance is so small, only the average time of 5 runs is shown in the results.

5.3 Calculating theoretical and effective memory bandwidths

When optimizing code it may be hard to know if there is reason to tweak the code further than it already is. Analyzing code and counting how many flops the code has can be time-consuming and unprecise, unless there is some profiler programs that can do this during runtime. A good guide when optimizing is the effective memory throughput achieved versus the theoretical maximum bandwidth possible on the given hardware. The following formulas are found in [28].

If specifications for the hardware is available, the maximum theoretical bandwidht is

$$\text{bandwidth (GB/s)} = \frac{\text{memory clock rate (Hz)} \cdot \frac{\text{memory interface width}}{8}}{10^9} \tag{5.1}$$

When using DDR (double data rate) RAM, as current GPUs do, this number must be calculated by 2 (see [28, p. 13]), giving the new equation

$$\text{bandwidth (GB/s)} = \frac{2 \cdot \text{memory clock rate (Hz)} \cdot \frac{\text{memory interface width}}{8}}{10^9} \tag{5.2}$$

The effective bandwidth can be calculated by

$$\text{Effective bandwidth (GB/s)} = ((B_r + B_w)/10^9)/\text{elapsed time} \tag{5.3}$$

where B_r are number of bytes read and B_w are number of bytes written.

These examples define a gigabyte as 10^9 bytes. In [28], there is no clear definition of wheter a gigabyte is defined as 10^9 or 1024^3 bytes, as long as users are concise of which one is used.

Chapter 6

Algorithm implementations and observations

6.1 Overview

This chapter will describe how the brute-force bilateral filter and histogram, described in Section 3.3, were implemented and optimized. In addition, we will describe how the bilateral filter was applied on videos in real-time using gstreamer [3].

While developing the filter kernel code, we had a jpg test image with resolution 1280x720. This can also be representative for a single frame in a video stream. Two utilities were made for OS X. The first one read and parsed an image file and then wrote the raw data buffer to a file. This tool supports all image types supported by the `NSImage` class, which are all the common formats. The other tool read the data buffer from disk and created a tiff image file that were saved to disk. The raw image buffer were saved to disk as a plain bitmap as showed in Figure 6.1, where each value is an unsigned byte in the range [0,255]. The first pixel in the file is the pixel in the upper left corner of the image. Each image row is written sequentially, so the last portions of the file is the lowest pixel row in the image. The saved file holds no metadata like image size or color space, so that must be known outside the file. When saving a 1280x720 image this way we get a file that has a size of $1280 \cdot 720 \cdot 4 = 3686400$ bytes.

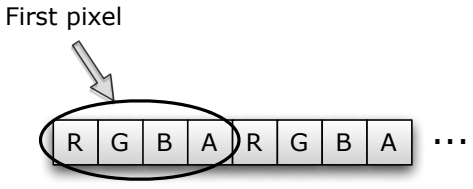


Figure 6.1: Bitmap representation saved to disk

For experimenting on the OpenCL kernels, a Python utility were written using the PyOpenCL[1] library. This library is a Python wrapper for OpenCL, and makes it easier to write the OpenCL code in a more Pythonic way. Some benefits are:

- There is no need to check return values. Exceptions are thrown in case of errors.
- Allocated GPU resources are automatically garbage collected.
- Setting kernel arguments and running the kernel is done in one call, OpenCL uses one call for setting each kernel argument.

In this way the host code could quickly be adjusted to new kernel versions during development. PyOpenCL read the plain bitmap buffer from disk and copied it to the GPU. After the execution on the GPU, the results were copied back to the host and written to file. The implementatino correctness, as long as there was no runtime errors or syntax errors in the device code, had to be checked manually by controlling the output image.

For measuring the kernel performance, we used the profiling option when making the OpenCL command queue. After the kernel run, we asked the kernel event about its profiling information, more specifically the timestamps for when the kernel started and stopped the execution.

Although the nVidia GTX 295 contains two GPUs, only one of them are utilized by the OpenCL kernels in this thesis. The OpenCL specification has the option to set execution offset numbers when invoking kernels, so in theory two kernels with different offsets could have been executed on each GPU. The specification states that this is a not supported issue yet, but it is not tested whether any of the implementations used support them. This approach would have needed another handling of memory transfers too, since the two GPUs on the GTX 295 do not share any global memory at all.

6.2 Bilateral filter

6.2.1 Implementation

The bilateral filter is described in Section 3.3.1.

To initially implement the bilateral filter, we just wrote Equation 3.5 as an OpenCL kernel. This implementation is shown in Listing 6.1. The difference from the equation is that we do not use neighbour pixels more than 2 standard deviances away. Since we used the `Image 2D` type as input argument, no extra logic were needed for handling reads outside the image bounds. This is implicitly handled by OpenCL according to the image sampler definition, named `s` in the code. The kernel was run with a 2-dimensional NDKernel with work group size equal to the image size, thus each thread's result was only one output pixel.

[1] `http://mathema.tician.de/software/pyopencl`

```
__kernel void bilateral_filter(
    __read_only image2d_t input,
    __write_only image2d_t output,
    float spatial_sigma,
    float range_sigma)
{
    const sampler_t s = CLK_NORMALIZED_COORDS_FALSE |
        CLK_FILTER_NEAREST | CLK_ADDRESS_CLAMP_TO_EDGE;

    int x = get_global_id(0);
    int y = get_global_id(1);

    int spatial_end_point = (int) (spatial_sigma * 2);

    float4 divisor = (float4)(255.0f);
    uint4 origo_color_i = read_imageui(input, s, (int2)(x,y));
    float4 origo_color = native_divide(convert_float4(
        origo_color_i), divisor);

    int i,j;
    float4 weight, tmp_color;
    uint4 tmp_color_i;
    float4 new_color = (float4)(0.0f);
    float4 normalization_factor = (float4)(0.0f);
    normalization_factor.w = 1.0f; // To avoid divide-by-zero
        error.

    for (j = - spatial_end_point; j <= spatial_end_point; ++j) {
        for (i = - spatial_end_point; i <= spatial_end_point; ++i)
             {
            tmp_color_i = read_imageui(input, s, (int2)(x-i,y-j));
            tmp_color = native_divide(convert_float4(tmp_color_i),
                 divisor);

            weight.x = exp(- (pown(fabs(tmp_color.x-origo_color.x)
                ,2) / (2*pown(range_sigma,2))) - ((pown(fabs((float
                )i),2) + pown(fabs((float)j),2)) / (2*pown(
                spatial_sigma,2))));
            weight.y = exp(- (pown(fabs(tmp_color.y-origo_color.y)
                ,2) / (2*pown(range_sigma,2))) - ((pown(fabs((float
                )i),2) + pown(fabs((float)j),2)) / (2*pown(
                spatial_sigma,2))));
            weight.z = exp(- (pown(fabs(tmp_color.z-origo_color.z)
                ,2) / (2*pown(range_sigma,2))) - ((pown(fabs((float
                )i),2) + pown(fabs((float)j),2)) / (2*pown(
                spatial_sigma,2))));

            normalization_factor += weight;
            new_color += weight * tmp_color;
        }
    }
    new_color /= normalization_factor;
    new_color.w = 1.0f;

    tmp_color_i = convert_uint4(new_color * 255.0f);
    write_imageui(output, (int2)(x,y), tmp_color_i);

}
```

Listing 6.1: Bilateral filter, initial device code

Then the following steps were done to optimize the implementation:

- Division was done using the `native_divide` function.
- The initial implementation calculated each color plane separately, resulting in 3 weights and 3 normalization factors. This was changed to have just 1 weight and 1 normalization factor for the entire pixel. To get this correct, each pixel's intensity was calculated using Equation 3.1.
- The gauss constants for spatial and intensity ranges were calculated once at startup and sent as arguments to the kernel.
- The intensity constant weights were quantized into a lookup table with 11 elements.
- All pixel values and intensity values for each work group and the needed surrounding neighbourhood were stored in the local memory.
- The inner loop was manually unrolled.
- Different work group sizes were tried. Using the two-dimensional work group with size $16 \cdot 16 = 256$ gave often best results.

Using only one color plane and quantizing the intensity difference weights will affect the filter's final result, but can be adequate for many purposes. Figure 6.2 shows two output images of the bilateral filter run with a kernel size of 21x21 and intensity sigma 0.25, one version using one intensity per pixel and the other version using all color channels. Note that the pixel values must be converted from unsigned chars to floats and back upon reading and writing from the image. This is due to that the image is defined with `UNSIGNED_INT8` as the color channel type. With nVidia, using the `read_imagef` function on this image type returned 0.0, so `read_imageui` had to be used instead. This is correct behaviour according to the OpenCL specification.

The fast local memory is a limited resource, so when it is utilized the work group size and the amount of local memory available must be taken into consideration. For experimentation, the bilateral filter implementation using the local memory for optimization was hand-tuned for the nVidia GTX 295 that has a local memory size of 16 kB[2]. It was also hardcoded for a filter kernel size of 9x9. Before calculations were performed, the threads should do the readings from global to local memory. In addition, it should read 4 pixels outside the work group's own pixel area. The work group size was set to $16 \cdot 24 = 384$, then the whole area read has size $(16 + (4 \cdot 2)) \cdot (24 + (4 \cdot 2)) = 24 \cdot 32 = 768$. Since 768 is the double of 384, all the local memory values is filled in two operations by each thread. A float uses 4 bytes of storage and the values stored in local memory are red, green, blue and intensity, thus this will need $768 \cdot 4 \cdot 4 = 12288$ bytes total, that is below the 16 kB boundary.

[2]This is the local memory size on all nVidia cards prior to the Fermi architecture

(a) Bilateral filter, separate color calculations

(b) Bilateral filter, not separate color calculations

Figure 6.2: Comparison of intensity treatment using the bilateral filter

```
__kernel void
bilateral_filter_all_optimizations(
    __read_only image2d_t input,
    __write_only image2d_t output,
    float spatial_sigma,
    float range_sigma,
    __constant float *spatial_constants,
    __constant float *range_constants,
    int spatial_box_width,
    int spatial_end_point,
    __local float intensities[32][24],
    __local float red[32][24],
    __local float green[32][24],
    __local float blue[32][24]
)
```

Listing 6.2: Bilateral filter optimized kernel declaration

The fastest running kernel declaration is shown in Listing 6.2. Notice that the local memory arrays are defined as 2-dimensional arrays in the kernel code, but the host code just sets these kernel arguments as local memory buffers with size of $32 \cdot 24 = 3072$ bytes. The nVidia compiler handles the necessary two-dimensional arithmetic when table lookups are done. This array declaration worked faster than using a 1-dimensional array and manually doing the lookup like this:

```
value = array[y * width + x];
```

The `mad24` function was also tried:

```
value = array[mad24(y,width,x)];
```

but did not work faster than the two-dimensional declaration. Pointers to pointers is not allowed in kernel arguments by the specification, but in this case nVidia has some smart table handling involved under the hood. It is worth noticing that the two-dimensional array declarations worked fastest on nVidia, but is *not* allowed by the Ati compiler that was tested later when the ATI card became available. It seems that the Ati compiler only tolerates pointer declarations as kernel arguments, not arrays.

During this work it became clear that small code changes could have a relatively large impact on performance compared to when programming for the x86 architecture. This may be due to more mature compilers and behaviours of the x86 processor.

Not all optimalizations could be run on every device. The GeForce 8800 GT could not handle the version where the code were manually unrolled. The runtime just gave an out-of-resources error, so it probably lacked enough registers or code capacity on the compute units. It was the same case with the nVidia 9400m card. The Ati card did not accept the array arguments, so it could not run the local memory optimized version. In addition, Ati did not support using image datatypes until April 2010, and then there was not found time to rewrite the arithmetics of the code.

6.2.2 Results

Table 6.1: Bilateral filter runtimes

Vendor	**Model**	**Time**
nVidia	GTX 295	4.8 ms
nVidia	Geforce 8800 GT	14 ms
nVidia	9400m	200 ms
Ati	Radeon HD 5850	8.2 ms
Intel	Core 2 Duo, 2 GHz	2.4 s

Table 6.1 shows the achieved execution times after optimization. These runtimes are on a picture with dimensions 1280x720. The OpenCL code was also run a

Intel Core 2 Duo using OS X Snow Leopard. Both cores on the CPU are utilized, but it is not checked if any SSE vector instructions are used. For comparison, the initial implementation in Listing 6.1 had a runtime of 440 ms on the nVidia GTX 295, so the optimization steps achieved a speed up factor of approximately 100.

6.2.3 Discussion

The usage of GPUs seems beneficial for image processing algorithms performing local neighbourhood calculations (stencils). By comparing the CPU runtime to the fastest GPU runtime achieved we see a speed up factor of $\frac{2400}{4.8} = 500$. Comparing by the slowest GPU runtime give a speed up factor of $\frac{2400}{200} = 12$, which is also a good result.

An image of size 1280x720 contains $1280 \cdot 720 \cdot 4 = 3686400$ bytes of data, also counting the alpha channel. Thus, total number of bytes read from and written to global memory is $2 \cdot 3686400 = 7372800$ bytes. The effective bandwidths achieved are listed in Table 6.2. These values are far from the theoretical maximum bandwidhts from Table 5.4. The reasons for this must be the 9x9 kernel, forcing 81 iterations with multiplications, addition and several reads from local memory. We know that the float addition and multiplication operations are cheap on GPU hardware, but the impact of just unrolling the inner loop (not documented) was quite large, so the iterations are expensive. The latest nVidia OpenCL release[3] has a `#pragma unroll` preprocessor directive, probably for a good reason.

Table 6.2: Bilateral filter effective bandwidths

Vendor	**Model**	**Effective bandwidth (GB/s)**
nVidia	GTX 295	1.54
nVidia	Geforce 8800 GT	0.53
nVidia	9400m	0.04
Ati	Radeon HD 5850	0.90

Processing more than one pixel per thread were not tested. Using any of the optimization approaches from Section 3.3.1 could have given better results.

6.3 Histogram

6.3.1 Implementation

The definition of a histogram is in Section 3.3.2. We implemented code that collected three 256-bin histograms, one for each color channel.

A histogram is a sort of reduction operation, except that the memory areas, or histogram counters, that must be accessed is dependent of the input values. This

[3] As of May 2010

makes it harder to implement a histogram than a reduce function on a GPU, because we either need support for atomic writes, or each thread must start working on a thread-private histogram. The behaviour when several threads add values to the same local memory address simultaneously is not specificed.[4]

The implementation is dependent on two extensions:

- `cl_khr_global_int32_base_atomics`
- `cl_khr_local_int32_base_atomics`

These extensions make it possible to do atomic addition, subtraction, increase, decrease and memory exchange on global and local memory respectively. The extension for global-memory atomic functions is found on all GPUs used in this thesis, but the atomic local-memory functions are only available on the nVidia GTX 295 and ATI Radeon HD 5850.

The implementation uses work groups with size 256, same as the number of bins in the histogram. There were made versions using both the conventional buffer and the image type. The conventional buffer version was a 1-dimension NDKernel with work-group size of 256. The image version used a 2-dimensional NDKernel with workgroup size 16x16. Every workgroup has its own local histograms. The code for the image version is shown in Listing 6.3.

[4]nVidia later released an OpenCL histogram version, using knowledge of hardware behaviour to implement a version with write conflicts. They knew that only one of the conflicting threads would make a successful memory transaction.

```
__kernel __attribute__((reqd_work_group_size(16,16,1)))
void histogram256_image_version(__read_only image2d_t input,
      __global uint histogram_r[256],
      __global uint histogram_g[256],
      __global uint histogram_b[256],
      __local uint partial_histogram_r[256],
      __local uint partial_histogram_g[256],
      __local uint partial_histogram_b[256]
     )
{

  const sampler_t s = CLK_NORMALIZED_COORDS_FALSE |
      CLK_FILTER_NEAREST | CLK_ADDRESS_CLAMP_TO_EDGE;

  int x = get_global_id(0);
  int y = get_global_id(1);
  int lx = get_local_id(0);
  int ly = get_local_id(1);
  int lwidth = get_local_size(0);
  int tid = mad24(ly,lwidth,lx); // calculating local id

  /* Resetting local memory */
  if (tid < 256) {
    partial_histogram_r[tid] = 0;
    partial_histogram_g[tid] = 0;
    partial_histogram_b[tid] = 0;
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  uint4 origo_color = read_imageui(input,s,(int2)(x,y));

  atom_inc(&partial_histogram_r[(uchar)origo_color.x]);
  atom_inc(&partial_histogram_g[(uchar)origo_color.y]);
  atom_inc(&partial_histogram_b[(uchar)origo_color.z]);
  barrier(CLK_LOCAL_MEM_FENCE);

  /* Copying from local memory to global memory */
  atom_add(&histogram_r[tid], partial_histogram_r[tid]);
  atom_add(&histogram_g[tid], partial_histogram_g[tid]);
  atom_add(&histogram_b[tid], partial_histogram_b[tid]);
}
```

Listing 6.3: Histogram code, image version

We also made two other code versions where each thread processed more than one input pixel. The conventional buffer version processed 4 pixels per thread, and the image version processed 2 pixel per thread. This was achieved by modifying the code and reducing the global work size accordingly. The image version could not have a version using 4 pixels per thread. Then the 2-dimensional global work size would be

$$\left(\frac{\text{image width}}{2}, \frac{\text{image height}}{2}\right) = \left(\frac{1280}{2}, \frac{720}{2}\right) = (640, 360) \tag{6.1}$$

and this was not possible with a work group size of 16x16, since 16 does not divide 360. Altering the work group size may not be so smart since image reads often are hardware optimized for 2d spatial locality.

The worst-case scenario of this implementation is if the input image has all identical pixels, leading to queues on the corresponding memory elements banks.

A small C program was made to compare the GPU runtimes with a CPU version. An `unsigned char` array, representing an RGB image, is created and filled in with random values in the range [0,255]. This was run on an Intel Core 2 Duo 2GHz. It ran only a single thread.

6.3.2 Results

The achieved runtimes for the different histogram versions is shown in Table 6.3. It is only run on the GPUs that support the required `cl_khr_local_int32_base_atomics` extension. The execution time of the CPU program was 3328 µs.

Table 6.3: Histogram runtimes

GPU	Image datatype	Pixels per thread	Time
nVidia GTX 295	Image 2D	1	3305 µs
Ati Radeon HD 5850	Image 2D	1	1388 µs
nVidia GTX 295	Buffer	1	3439 µs
Ati Radeon HD 5850	Buffer	1	1862 µs
nVidia GTX 295	Image 2D	2	2101 µs
Ati Radeon HD 5850	Image 2D	2	599 µs
nVidia GTX 295	Buffer	4	3431 µs
Ati Radeon HD 5850	Buffer	4	1551 µs

6.3.3 Discussion

The runtimes show no significant differences in execution times. The single-threaded CPU version is as fast as the buffer versions, but all the image versions run faster. It is clearly a benefit utilizing the image datatype for this thesis' domain. The highest speed up factor is approximately 5, so it may not be worth the cost of PCI data traffic unless the histogram kernel is one of many kernels of a GPU image processing pipeline.

Altering the image could have given other results. An all white image would lead to many sequential instead of parallel writes. This was not tested.

The reason for the high speed on the CPU may be that CPUs are optimized for low latency, while GPUs are optimized for processing sequential memory elements with high computational throughput.

There is a `glHistogram` function available in OpenGL, but according to [38] it is not hardware-optimized by nVidia.

6.4 Gstreamer visualization

By using gstreamer [3] it was possible to visualize the bilateral filter developed with OpenCL in real-time. Gstreamer has a pipelined architecture, making it

possible to make multimedia applications in an easy way. See [12, 40] for more details.

To utilize the GPU with gstreamer, we used the `gst_plugins_gl` package that consists of several OpenGL plugings that perform image processing on the GPU. The filters are implemented using GLSL. We rewrote a plugin and its Makefiles to make it utilize OpenCL in addition to OpenGL.

For development, we used the existing `glfiltersobel` plugin and redefined many of its callback functions. All gstreamer plugins consist of initialization and finalization code. When intializing the plugin we created an OpenCL context. The video input to the effect plugins in the `gst_plugins_gl` package is an OpenGL framebuffer object that exists in the GPU memory. To make this work, we used the OpenGL/OpenCL sharing APIs that was implemented by nVidia at this time.[5] On every frame, the plugin used the `clEnqueueAcquireGLObjects` call to aquire the framebuffer object from OpenGL. On the first frame, an OpenCL 2-dimensional image object was made, the same size as the framebuffer object. This was necessary since reading from and writing to the same image is not allowed by OpenCL kernels. This image object was used on every consecutive frame. To apply the filter, two kernel executions were necessary. The first one, called *identity* , copied the acquired framebuffer object to the OpenCL image object, and the second one applied the filter kernel, reading from the temporary image object and writing the result back into the acquired framebuffer object. When the OpenCL kernels were finished, `clEnqueueRelaseGLObjects` were run, handling the object back to OpenGL.

Instead of using the *identity* kernel for copying, the function `clEnqueueCopyImage` was tried but did not work. This have probably not lead to any noticeable performance loss.

Our custom OpenCL filter was placed in the gstreamer pipeline, as can be seen in Figure 6.3. The pipeline does the following steps (according numbers are in the figure):

1. Read the video from file.
2. Separate video and audio from the Quicktime container (this example uses quicktime movies).
3. Select correct codec and decode the video stream.
4. Do necessary color space conversion.
5. Transfer the stream to the GPU.
6. Apply custom filter, all processing done on the GPU.

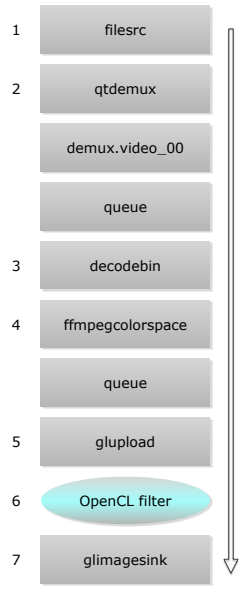


Figure 6.3: Gstreamer pipeline

[5] December 2010

7. Show the result on screen.

The audio stream, if existent, is ignored. Note that the pipeline is only using already existing elements, except for the filter that was custom made for utilizing the interoperability between OpenCL and OpenGL. The purpose of the *queue* elements is to force the use of multiple threads. They behave as producer-consumer queues, and forces the entire pipeline to spawn several threads. The pipeline in Figure 6.3 has 2 *queue* elements, making it use a total of at least 3 threads, more if any of the plugins spawns more threads. This gave a much better result, probably since the video visualization was run on a Linux machine with a quad-core processor. The GPU was an nVidia GTX 260.

By using gstreamer it was quite easy to visualize the result of applying filters to video in real-time. In addition, it was not necessary to write a dedicated application to set up the pipeline and plugin attributes. Gstreamer has a utility called `gst-launch` that can be used to run arbitrary pipelines from the command line using the current system's available gstreamer plugins. This is done by the command showed in Listing 6.4. Using `gst-launch` was also very beneficial for debugging purposes while developing the OpenCL plugin.

```
gst-launch  filesrc location=<filename> ! qtdemux name=demux
    demux.video_00 ! queue ! decodebin ! ffmpegcolorspace ! queue !
     glupload ! glopenclfilter spatial_std_dev=2 intensity_std_dev
    =0.25 ! glimagesink
```

Listing 6.4: Using `gst-launch` to run a gstreamer pipeline

GStreamer is implemented using glib[6], hence utilizes the GLib Object System. This is a library that allows object-oriented programming in C, but more similar to Objective-C than C++. These objects support introspection, making it possible to ask an object about its properties. Every gstreamer plugin is a GObject, and by using the `gst-inspect` utility from the command line, it is possible to see all properties of a plugin, like the media types it is expecting as input and output streams and which color spaces it supports. When a gstreamer pipeline is initialized, for instance by `gst-launch`, the plugins negotiate with the neighbours about stream media types and adjusts their input and output types accordingly. If two consecutive plugins can not agree about the data stream format, this is reported as an error and the pipeline can not be initialized. It is also possible to set plugin attributes, as showed with the `glopenclfilter` plugin in Listing 6.4. This means that if the OpenCL code for the filter kernel is written correctly, no filter attributes or video resolutions have to be hardcoded. If no attributes are set, the plugin's default values, defined by the plugin developer, are used.

The result was that we had the bilateral filter applied to a video with resolution 1280x720 and frame rate of 24 fps running smoothly. When applied to a video

[6] `http://library.gnome.org/devel/glib/`

with the same frame rate and resolution 1680x1050 the result was not quite as good and some frames were lost. We tried to set this up with videos at frame rates of 30 and 60 fps, but they were not encoded and hence very large, so the hard drive could not read fast enough from disk. These videos were made for testing the noise reduction capabilities of the bilateral filter. We heard that lossy encodings of the video would implicitly remove the noise, so we tried to use gstreamer to encode the video using a lossless coding algorithm but this was not successful. Using a live camera feed was not tested.

6.5 Observations

During the initial implementations, some issues became apparent. The OpenCL implementation with OS X Snow Leopard was just released when this thesis' work started, and it became obvious that these were new and immature implementations. These issues are worth mentioning because it has to do with code portability. The initial implementations were initially done on Snow Leopard, and later moved to nVidia and AMD compilers.

6.5.1 Compiler behaviour

For instance

```
float4 weights = (float4)(0.3f, 0.6f, 0.1f, 1.0f);
result = weights / 3.0f;
```

worked with Snow Leopard compiler, but not on nVidia. Then had to use

```
float4 weights = (float4)(0.3f, 0.6f, 0.1f, 1.0f);
result = weights / (float4)(3.0f);
```

which work on all tested implementations. Another example is using

```
__constant float gauss_weights[HEIGHT][WIDTH]
```

instead of

```
__constant float *gauss_weights
```

as kernel argument was fastest on nVidia cards, instead of having to manually calculater the index of an 1-dimensional by using `mad24()` function. This does not work with the Ati compiler.

The initial compiler that was shipped with Snow Leopard also gave very bad error output, sometimes nothing at all even if the faults were plain syntax errors. This complicated the development process.

6.5.2 Driver updates

Most of the image processing was done using OpenCL's `image_2d` data type. How good this was implemented varied. At one point, the nVidia drivers were updated leading to that the bilateral filter used one third of the earlier runtime.

The implementation provided by Apple on Snow Leopard changed the number of compute units on the nVidia 9400m from 16 to 2, without changing the driver version number. These numbers may be due to the fact that nVidia hardware have 8 stream processors in each compute unit.

6.5.3 Debugging

Code debugging can be hard. When a kernel runs on the GPU, it is not possible to set breakpoints in the code. This is possible on CUDA, and CUDA also has a function `cuPrintf` that can be used in the code running on the GPU that prints status messages to the console. To have some sort of debugging, we had a global array in the device's memory where we had one thread responsible for storing debugging status variables. When the kernel was finished executing, this array was read back from the GPU and printed out by the host application.

Chapter 7

GPUScribe

7.1 Motivation

The results from chapter 6 shows that utilization of graphic processing units can achieve great speedups for some image processing applications. The work needed to achieve this also showed that optimalization tuning had to be manually done by the developer according to his or her hardware. After talking to people who are developing image processing logic for a living at Tandberg ASA it became clear that there should be a better way of testing and developing filters without having to do so much manual tweaking. In addition, it should be easy to find out exactly what are the strengths and weaknesses of the hardware that the developer is currently working on and what the device support.

It is important to have in mind that usage of the GPU is not intended when developing the algorithms for image or video filtering, but it is supposed to be a implementation tool for achieving best possible runtimes of already defined algorithms. Thus the filter development can be done in for instance MATLAB, using its Image Processing Toolbox. The purpose of our tool is to help port these filters for execution on the GPU, and we are always interested in having the fastest possible runtime. This means we want to have a result that can be easily put into a C or C++ program, or a C/C++ utility library that can be part of the system. The reason for not considering languages like Java is that the specification and OpenCL host calls is made for and implemented in C, and can be used also in C++ code directly. When not using any language bindings we get all features available for each implementation. Wrappers like PyOpenCL can lead to that we miss some features of the available implementations, like some of nVidia's proprietary extensions that are not defined in the official OpenCL specification. Mail correspondences with the author of PyOpenCL made it obvious that this was a deliberate choice.

At the time when this devlopment process started, there were no tools for auto-optimizing code for devices from different vendors. There were projects for nVidia CUDA, like Qilin [26], hiCuda [17] and PyCUDA [23], but no projects that auto-optimalized code for GPUs from different vendors.

7.2 Wanted features

The algorithms mentioned in Section 3.3 give us some hints about what we need for making a system for helping develop image filters in general, especially if we want to take benefit of the optimizations and data structures used by the examples in Section 3.3.1. The scope of this solution was limited to only help optimize the kernel code, since those are the building blocks that have most impact on execution time.

This was the wanted functionality:

1. The system should optimize OpenCL kernels for the current system's hardware, GPUs only.
2. Kernels should be annotated or have templates that would help generate the optimal code.
3. The tool had to be usable from a C or C++ context, so it could be used for running code at highest possible performance.
4. There may be need for special data-structures, like a hash table implementation for using the permutohedral lattice [7]. We can assume that these data structures will be frequently updated in-place or rebuilt from scratch, maybe for every frame in a video stream.
5. The kernels may need several input frames for interframe processing.
6. Should be able to optimize chaining of image filters.
7. It should be possible to represent images both as OpenCL 2D images or normal arrays.
8. Must be able to generate filter constants, like a convolution filter matrix, or define such constants in another way. These constants must either be used as `__constant` arguments to the kernel or be hardcoded in the generated code.
9. Test variants of generated kernels with representative test data and use the one with the best runtime.

7.3 Implementation

7.3.1 Initial framework

The intention was to make GPUScribe a tool with expandable template functionality, focusing only on the OpenCL device code. For a clean syntax, it was considered making GPUScribe as a code generator converting from Python code to OpenCL device code, using the Python abstract syntax trees, but this was considered being a too complicated task for the remaining time.[1]

[1] An implementation working this way, Clyther [1], was later made public late April 2010.

A bottom-up approach was taken on the development. At first were made some microbenchmarks that tested various characteristics of the current hardware. The following issues could be measured/tested:

- The effect of using vector load and store functions.
- Effective bandwidht, using identity kernels.
- The effect of using asynchronous copies between local and global memory. This is supported by the OpenCL specification, but not necessarily by the current device.
- The effect of using the `prefetch` function, if any.
- Give each thread more work, like outputting four pixels instead of one.
- Speed benefits from using hardware native functions, will then lose the IEEE-754 compliance.
- Effects of using different build options.
- Effects of loop unrolling.
- Effects of using pinned memory for data transfers between host and device.
- Effect of reading from `image2d` datatype versus using a normal array.
- Effect of different access patterns for global memory reads and writes.
- Concurrent kernel support.
- Whether using normalized coordinates affects performance.
- Effect of having consecutive reads from the same pixel, using the `read_imagef` function.

Because of the remaining timeframe, the code was written using PyOpenCL instead of using the native OpenCL calls directly from C or C++. This was due to the neeed amount of text processing and template processing where Python has an advantage of the language itself and many template frameworks compared to C. After having seen at several templating systems for Python, Jinja2[2] was chosen.

The procedure followed by GPUScribe is as follows:

1. Input is a template file, should be support for other intermediate data buffers.
2. Find GPU devices on the system.
3. For each device:
   (a) Run microbenchmarks and save the results.

[2] `http://jinja.pocoo.org/2/`

4. Process template file according to results from step 3a, if possible in several versions.
5. Do test runs of the different versions.
6. Pick the fastest version from step 4.

In this way, GPUScribe was an open-ended tool where the available templates and necessary microbenchmarks could be added as they were developed. In addition, being able to use the Jinja2 template system in a OpenCL kernel file could be an advantage in itself.

7.3.2 Template functionalities

Filter chain

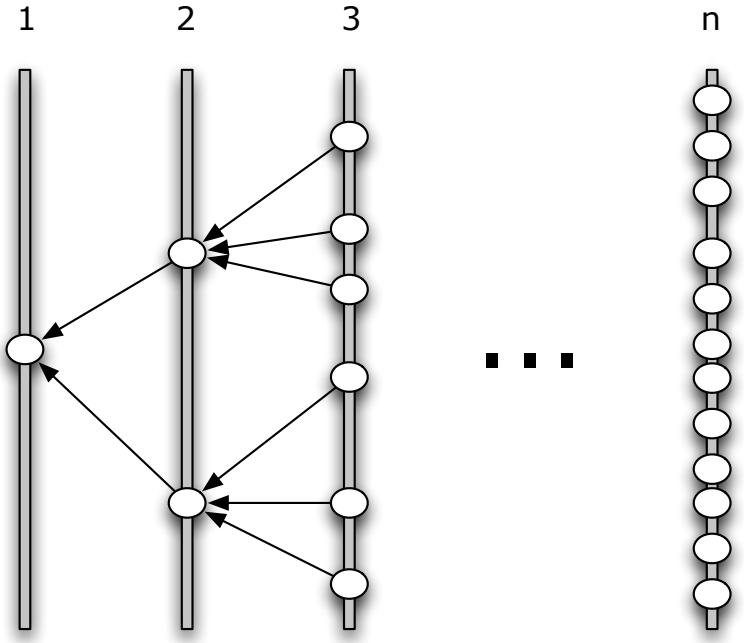


Figure 7.1: Example of pixel aggregation with a filter chain

In many real-life applications, image processing is done by running several filters sequentially. Running several iterations with one filter can also give wanted effects.

With the success of using the OpenCL image datatype for image processing, a filter chain helper was made. Its behaviour is illustrated in Figure 7.1. The purpose is to run several image filters as parts of the same kernel, thus saving the overhead needed for invoking several kernels. An example template file is given in Listing 7.1. This approach resulted in functions automatically knowing from

where they should read their input pixels and filter constants are calculated by Python and hardcoded in the generated output code. We tried to make it possible to run several iterations of the same filter, but this was not made possible. This approach give a depth-first traversion of the functions called, begin a benefit because of the texture cache utilization.

```
{% macro gauss_const(x,y,spatial_sigma) -%}
  {{ (1.0/(2.0 * numpy.pi * spatial_sigma**2)) * math.exp(-(((x|
      abs)**2+ (y|abs)**2)/(2*spatial_sigma**2))) }}
{%- endmacro %}

{% set spatial_sigma = 2 %}
{% set blurFilter = Filter('gaussian_blur', filter_size=(9,9)) %}
{% set grayFilter = Filter('grayscale') %}
{% set filterChain = FilterChain(blurFilter,grayFilter) %}

inline float4 gaussian_blur(__read_only image2d_t source_image,
    const sampler_t s, int2 coord)
{
    float4 sum = (float4)(0.0f);
    int2 tmp_coord;

    {% for x,y in blurFilter.point_list %}
      tmp_coord.x = coord.x + ({{ x }});
      tmp_coord.y = coord.y + ({{ y }});
      sum += {{ blurFilter.get_pixel('source_image', 's', '
          tmp_coord') }} * {{ gauss_const(x,y,spatial_sigma) }};
    {% endfor %}

    return sum;
}

inline float4 grayscale(__read_only image2d_t source_image, const
    sampler_t s, int2 coord)
{
  float4 pixel_value = {{ grayFilter.get_pixel('source_image', 's'
      , 'coord') }};
  float intensity = native_divide(pixel_value.x + pixel_value.y +
      pixel_value.z, 3.0f);
  return (float4)(intensity,intensity,intensity,1.0f);
}

/* Main kernel that run the filter chain */
__kernel
void imageread_gen_vec_test(__read_only image2d_t input_image,
    __write_only image2d_t output_image)
{
    const sampler_t s = CLK_NORMALIZED_COORDS_FALSE |
        CLK_FILTER_NEAREST | CLK_ADDRESS_CLAMP_TO_EDGE;

    int idX = get_global_id(0);
    int idY = get_global_id(1);

    float4 res = gaussian_blur(input_image, s, (int2)(idX,idY));
        // TODO: Autogenerate this function call!!
    res.w = 1.0f; // just in case...
    write_imagef(output_image, (int2)(idX,idY), res);
}
```

Listing 7.1: Filter chain template example

7.4 Discussion

Only one template functionality is not much, and the syntax shown in Listing 7.1 feels quite chaotic and is not very helpful. The filter chain worked, but it did not feel like a helpful tool. In addition, it does not use any of the collected hardware profiling information.

The problem with GPUScribe was the bottom-up approach to the design and implementation. When the GPUScribe development process started, there was not chosen a template engine and there were not defined any syntax definitions initially that could have been guiding this process. In this way we could have discovered at an earlier point that this was not a good idea needing further investigations. In addition, the difference found in the OpenCL implementations from Ati and nVidia were not known at the point when this process started, as the Ati card was not yet available. The template-based approach we tried had too may things outside of its control and too many possibility permutations for optimizing code, in addition to a lacking system to verify correctness of the generated output, compared to any of the other output alternatives. Often there are tradeoffs between optimalization and correctness of the result. The bilateral filter implementation, for instance, used only one value representing the intensity of a pixel instead of using calculating on the RGB color channels separately. This approach gave a result we could accept while reducing the total number of needed calculations.

Too much time was also used trying to solve running several iterations of the same filter.

Functionality that could have been implemented is using arbitrary data types, as discussed in [23]. Another functionality could have been to input a convolution matrix and auto-generate the code performing the convolution, maybe checking first if the convolution is separable and if so, optimize accordingly.

Chapter 8

Related work

Much related work was published during the work on this thesis, due to general-purpose GPU computing being a popular research area at the current time. These are tools for utilizing GPUs in various ways:

- Runtime frameworks using CUDA: Qilin [26], PyCUDA [23] and hiCUDA [17].
- Sample programs bundled with the SDKs from Ati and nVidia.
- GAtlas [2], auto-optimization of linear algebra routines for OpenCL
- CLyther [1], generating OpenCL code from Python using abstract syntax trees.
- GpuCV [10], GPU implementation of OpenCV.
- FCUDA [34], generates FPGA code from CUDA.
- Core Image, part of Mac OS X. Implicitly using GPUs.
- Intel Ct [4], data-parallel runtime development framework. There are plans for future GPU support using OpenCL or GLSL.[1]

[1] `http://www.youtube.com/watch?v=hX-ectOgAso`

Chapter 9

Discussion

This chapter is an overall discussion of this thesis' work. The discussions regarding the specific implementations are found in Section 6.2.3, 6.3.3 and 7.4.

During this tesis' work, it has become apparent that OpenCL suffers from being a relatively new specification. The implementations we have tried have had several issues:

- Not following the specification at all details. An example is that nVidia's implementation
- Difference in device code acceptance.
- Profiling and debugging tools, if any, are primarily available for the Windows platform.
- Very sparse output when syntax errors are found in the device code, if any at all.

The development process could have been simplified by using only CUDA. Available debugging tools are better, making it possible to set breakpoints in the GPU device code. In addition, the hardware variations would have been limited. Nonetheless, this thesis' focus was to explore GPU programmability regardless of vendors.

We have only focused on the device code itself, but a new approach for helping developers could have been to make a C or C++ runtime that takes care of all the host API calls. This approach would assume that *development and optimalization of OpenCL kernels is the developer's work, and optimizations is done manually using testing and failure.* The API calls in OpenCL are quite low level. C++ bindings have made them easier to program without losing performance. This tool could have been used to place events on a command queue in an intuitive way, making it possible to quickly test different attributes of a OpenCL pipeline, like whether a host memory buffer is pinned in the operating system and testing different work group sizes and build parameters for compiling the device code. This runtime could get a task-graph description in xml or json, being exported by a GUI utility used for setting up the various events and

buffers. Making this runtime vendor-independent will probably be easier than modifying the OpenCL kernel code, as we have tried.

Chapter 10

Conclusion

We have investigated several properties of utilizing graphic processing units from different vendor for image and video processing. Algorithms using local neighbourhood calculcations like the bilateral filter did achieve calculation speedup factors of 500, not considering data traffic latency on the PCI bus. Histogram calculations had a maximum speed up improvement of 5. There are, however, many image processing techniques requiring data types like kd-trees and hash tables that are not very well suited for the current memory model of the graphics hardware, especially not for real-time performance. The usage of OpenCL's image2d datatype has both made the development process easier and it also achieved the best results. For prototyping OpenCL kernel code, PyOpenCL has been a very valuable tool making it faster to change parameters like kernel arguments and memory buffers.

The filters have been successfully applied and visualized on videos at real-time using gstreamer.

The development process suffered from OpenCL being an immature framework where not all implementations did follow the specification good enough for code to be portable without making significant changes. In addition, the profiling tools are mainly available on the Windows platform, that was not utilized when working on this thesis.

Attempts on making a template-based solution, GPUScribe, for creating OpenCL GPU kernels have been tried, but this did not seem like a good approach at this time due to differences in OpenCL compiler behaviours and the existence of other, maybe better alternatives, like Clyther, PyCUDA or the C++ template support provided by CUDA. Much of these techniques came along while working on this thesis. In addition it was hard to settle for template functionalities that actually helped the developers instead of cluttering their code and adding unnecessary levels of abstraction. All in all the wanted result was a high enough speedup factor in C/C++ applications that it is worth the data traffic latency on the PCI Express bus. The effenciency of OpenCL kernels is also dependent of work group sizes and build parameters, being outside of GPUScribe's control. In addition, there are coming vendor-specific extensions that further complicates this approach.

Bibliography

[1] Clyther website. http://clyther.sourceforge.net/ Last visited 14.05.2010.

[2] Gatlas website. http://golem5.org/bucket/gatlas/ Last visited 20.05.2010.

[3] Gstreamer. http://www.gstreamer.net/ Last visited 14.05.2010.

[4] Intel ct website. http://software.intel.com/en-us/data-parallel/ Last visited 24.05.2010.

[5] The khronos group releases opencl 1.0 specification. http://www.khronos.org/news/press/releases/the_khronos_group_releases_opencl_1.0_specification/ Last visited 12.05.2010.

[6] Lectures on image processing. http://www.archive.org/details/Lectures_on_Image_Processing Last visited 24.04.2010.

[7] A Adams, J Baek, and A Davis. Fast high-dimensional filtering using the permutohedral lattice.

[8] A Adams, N Gelfand, J Dolson, and M Levoy. Gaussian kd-trees for fast high-dimensional filtering. *graphics.stanford.edu*.

[9] DA Alcantara, A Sharf, F Abbasinejad, S Sengupta, M Mitzenmacher, JD Owens, and N Amenta. Real-time parallel hashing on the gpu. *ACM Transactions on Graphics (TOG)*, 28(5):1–9, 2009.

[10] Y Allusse, P Horain, A Agarwal, and C Saipriyadarshan. Gpucv: An opensource gpu-accelerated framework for image processing and computer vision. *MM'08: Proceeding of the 16th ACM international conference on Multimedia*, pages 1089–1092.

[11] Alireza Nasiri Avanaki. A spatiotemporal edge-preserving de-noising method for high-quality video. *Signal Processing and Information Technology, 2006 IEEE International Symposium on DOI - 10.1109/ISSPIT.2006.270789*, pages 157–161, 2006.

[12] Richard John Boulton, Erik Walthinsen, Steve Baker, Leif Johnson, Ronald S Bultje, Stefan Kost, and Tim-Philipp Müller. Gstreamer plugin writer's guide (0.10.29). Apr.

[13] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA*, 14, Aug 2009.

[14] J Chen, S Paris, and F Durand. Real-time edge-aware image processing with the bilateral grid. *ACM Transactions on Graphics (TOG)*, 26(3):103, 2007.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), Jan 2008.

[16] K Fatahalian and M Houston. A closer look at gpus. *Communications of the ACM*, 51(10):50–57, 2008.

[17] TD Han and TS Abdelrahman. hi cuda: a high-level directive-based language for gpu programming. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, 2009.

[18] Advanced Micro Devices Inc. Ati intermediate language (il) specification, version 2.0a. pages 1–430, Feb 2010.

[19] Advanced Micro Devices Inc. Ati stream sdk cal programming guide v2.0. Jan 2010.

[20] Advanced Micro Devices Inc. Ati stream sdk opencl programming guide. Apr 2010.

[21] Advanced Micro Devices Inc. Ati stream sdk v2.01 performance and optimization. Jan 2010.

[22] Daehyun Kim, V Lee, and Yen-Kuang Chen. Image processing on multicore x86 architectures. *Signal Processing Magazine, IEEE DOI - 10.1109/MSP.2009.935384*, 27(2):97–107, 2010.

[23] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: Gpu run-time code generation for high-performance computing. *arXiv*, cs.DC, Jan 2009.

[24] Aaron Lefohn, Mike Houston, Johan Andersson, Ulf Assarsson, Cass Everitt, Kayvon Fatahalian, Tim Foley, Justin Hensley, Paul Lalonde, and David Luebke. Beyond programmable shading (parts i and ii). *SIGGRAPH '09: SIGGRAPH 2009 Courses*, Aug 2009. Very nice descriptions of GPU hardware workings!!

[25] E Lindholm, J Nickolls, S Oberman, and J Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE DOI - 10.1109/MM.2008.31*, 28(2):39–55, 2008.

[26] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2009.

[27] A Munshi. The opencl specification version: 1.0 document revision: 48. Oct 2009.

[28] nVidia Corporation. Nvidia opencl best practices guide. Aug 2009.

[29] nVidia Corporation. Nvidia cuda programming guide 3.0. Feb 2010.

[30] nVidia Corporation. Opencl programming guide for the cuda architecture. Feb 2010.

[31] nVidia Corporation. Ptx: Parallel thread execution, isa version 2.0. Jan 2010.

[32] J Owens, M Houston, D Luebke, and S Green. Gpu computing. *Proceedings of the IEEE*, Jan 2008.

[33] JD Owens, D Luebke, N Govindaraju, M Harris, J Kruger, AE Lefohn, and TJ Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[34] A Papakonstantinou, K Gururaj, J.A Stratton, D Chen, J Cong, and W.-M.W Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on DOI - 10.1109/SASP.2009.5226333*, pages 35–42, 2009.

[35] S Paris and F Durand. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision*, 81(1):24–52, 2009.

[36] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, and Frédo Durand. A gentle introduction to bilateral filtering and its applications. *SIGGRAPH '08: SIGGRAPH 2008 classes*, Aug 2008.

[37] TQ Pham and LJ Van Vliet. Separable bilateral filtering for fast video preprocessing. *IEEE International Conference on Multimedia and Expo, 2005. ICME 2005*, page 4, 2005.

[38] A Ruiz, M Ujaldon, J.A Andrades, J Becerra, Kun Huang, T Pan, and J Saltz. The gpu on biomedical image processing for color and phenotype analysis. *Bioinformatics and Bioengineering, 2007. BIBE 2007. Proceedings of the 7th IEEE International Conference on DOI - 10.1109/BIBE.2007.4375701*, pages 1124–1128, 2007.

[39] M Stokes, M Anderson, S Chandrasekar, and R Motta. A standard default color space for the internet–srgb. *Microsoft and Hewlett-Packard Joint Report*.

[40] Wim Taymans, Steve Baker, Andy Wingo, Ronald S Bultje, and Stefan Kost. Gstreamer application development manual (0.10.29). Apr.

[41] C Tomasi and R Manduchi. Bilateral filtering for gray and color images. *Computer Vision, 1998. Sixth International Conference on*, pages 839–846, 1998.

[42] Ben Weiss. Fast median and bilateral filtering. *SIGGRAPH '06: SIGGRAPH 2006 Papers*, Jul 2006.

[43] Qingxiong Yang, Kar-Han Tan, and N Ahuja. Real-time o(1) bilateral filtering. *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 557–564, 2009.

[44] K Zhou, Q Hou, R Wang, and B Guo. Real-time kd-tree construction on graphics hardware. *ACM SIGGRAPH Asia 2008 papers*, page 126, 2008.

Appendix A

Presentation held at Tandberg

On the following pages are the slides from a presentation that was held for a group at Tandberg ASA, December 2009.

Using GPU for real-time video processing

Presented by Børge Lanes

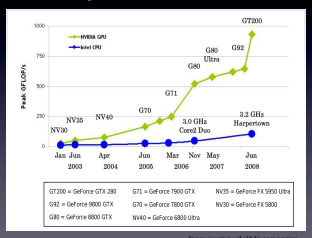
About me

- 30 years old
- Working on master thesis in computer science at University of Tromsø
- Interested in parallel programming and concurrency issues
- Thesis delivery due 15th of May 2010

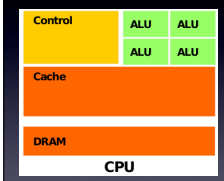
Outline

- GPU Architecture
- gpGPU programming APIs
- OpenCL in depth
- My work
- Demo

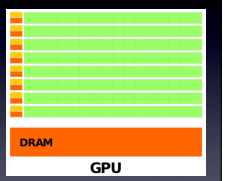
Floating point operations per second



How is this possible?

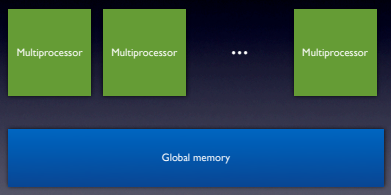


Many transistors devoted to logic and branching

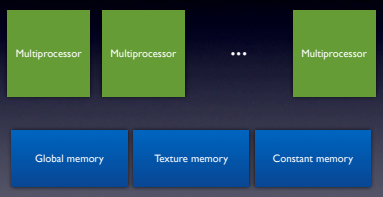


Almost all transistors devoted to floating point processing

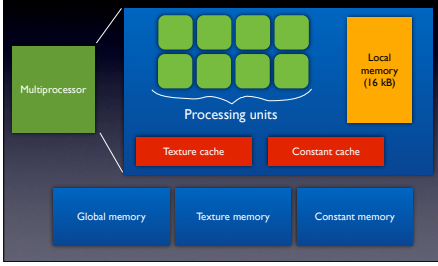
GPU Architecture



GPU Architecture



GPU Architecture



Current GPGPU Programming Issues

- All threads and multiprocessors run identical code
- Branched execution is expensive within a multiprocessor
- Architecture is optimized for single-precision float operations
- To-be-released Fermi architecture will allow developers to run different code on different multiprocessors simultaneously

Current gpgpu programming APIs

- CUDA - nVidia's proprietary environment
- DirectCompute - part of DirectX 11
- OpenCL
- OpenGL shading language, data must be camouflaged as graphics

OpenCL

- Open Computing Language
- Maintained by Khronos group
- Initiated by Apple, many industry leading participants
- Has many similarities with CUDA
- Correctly written code can run on any OpenCL device regardless of vendor

OpenCL participants

OpenCL programming model

- Data-parallel and task parallel programming models
- GPUs are well suited for data-parallel model
- Supports proprietary extensions
- Devices can be queried about their capabilities

OpenCL programming model

- Host orchestrates creation of context and device global memory allocations
- OpenCL code is compiled during application runtime with customizable build parameters
- Host CPU can also run OpenCL-code
- OpenCL/OpenGL interoperability

OpenCL programming language

- C99 based with some restrictions - recursions not allowed
- IEEE-754 compliant floating point instructions
- Can use native hardware functions
- Vector data types and operations

OpenCL kernel example

```
__kernel void rgb_permutation (
  __read_only image2d_t input,
  __write_only image2d_t output)
{
  float4 color; float intensity;
  sampler_t s = CLK_NORMALIZED_COORDS_FALSE |
  CLK_FILTER_NEAREST | CLK_CLAMP_TO_EDGE;

  int x = get_global_id(0);
  int y = get_global_id(1);
  float4 color = read_imagef(input, s, (int2)(x,y));
  color.yzx = color.xyz;
  write_imagef(output, (int2)(x,y), color);
}
```

Examples of special functions

- `T native_divide(T a,T b);`
- `mad24(a,b,c) = (a * b) + c`
- `x.lo, x.hi, x.odd, x.even`
- `float dot(float4 a, float4 b);`

Memory usage

- Local memory (shared by work group) is much faster than global memory
- Threads must use correct memory access patterns for optimal performance

My work so far

- Have used Snow Leopard and Linux as development platforms
- Devices tested:
  - nVidia 9400m
  - nVidia 8800 GT
  - nVidia GTX 295
  - Intel Core2 duo
  - not yet ATI/AMD
- PyOpenCL used during development
- gstreamer framework used for video-filter visualization
- Implementations
  - Brute-force bilateral filter
  - Histogram

Bilteral filter times

- Single frame, 1280x720
- 9x9 kernel

nVidia GTX 295	4.8 ms
nVidia GTX 260	7.9 ms
nVidia 8800 GT	14 ms
nVidia 9400m	200 ms
Intel Core 2 Duo, 2 GHz	2.4 s

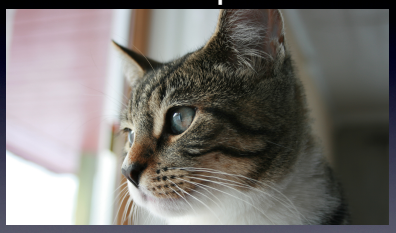
Experiences

- Using native functions greatly increases speed - but we loose IEEE-754 compliance
- Hardware optimized for texture memory usage
- Work-group size is very important
- Hard to develop code that runs optimal on different GPUs
- PyOpenCL is a good tool for kernel development and testing
- Compilers behave differently
- Debugging is not easy

Challenges

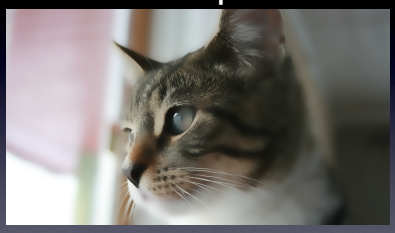
- Hardware-optimized features may not be known by developer, dependent of hardware and its OpenCL driver implementation
- Hiding OpenCL device source code without using hardware-dependent precompiled binaries
- Data transfers over PCI-bus

Example

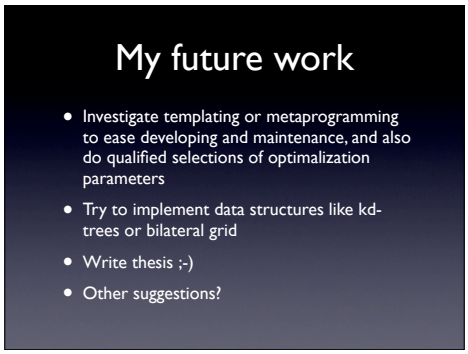


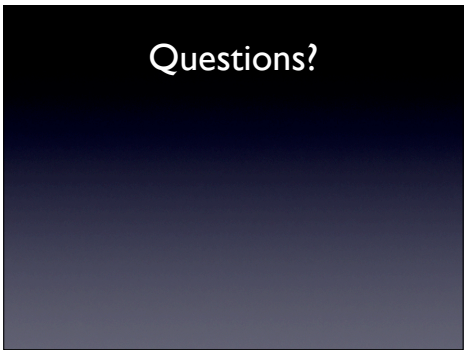
Original image

Example



Bilateral filter 40x40 kernel


My future work
• Investigate templating or metaprogramming to ease developing and maintenance, and also do qualified selections of optimalization parameters
• Try to implement data structures like kd-trees or bilateral grid
• Write thesis ;-)
• Other suggestions?


Questions?