An efficient GPU framework for Image Processing

Per Karlsson

September 2014

Abstract

This thesis tries to answer how to design a framework for image processing on the GPU, supporting the common environments OpenGL GLSL, OpenCL and CUDA. An generalized view of GPU image processing is presented. The framework is called gpuip and is implemented in C++ but also wrapped with Python-bindings. The framework is cross-platform and works for Windows, Mac OSX and Unix operating systems.

The thesis also involves the work of creating two executable programs that uses the gpuip-framework. One of the programs has a graphical user interface and the other program is command-line only. Both programs are developed in Python.

Performance tests are created to compare the GPU environments against a single core CPU implementation. All the GPU implementations in the gpuip-framework are significantly faster than the CPU when executing the presented test-cases. On average, the framework is two magnitudes faster than the single core CPU.

Contents

1 Introduction

1.1 Background

1.1.1 Image processing

An image can be seen as a mathematical function $i(x, y)$ where x and y are the spatial coordinates and the output is a color at position (x, y). If the color has discrete quantities and the total image has a finite amount of samples, it is called a *Digital Image*[1]. The samples each has a unique spatial coordinate and are referred to as *pixels*. The field of *Digital Image Processing* refers to processing a digital image and its pixels. Every time Image Processing is mentioned in this thesis, it is assumed that it refers to Digital Image Processing.

The most typical application in image processing is when an algorithm is on an input image to create a modified output image. Operations such as blur, sharpen and noise removal are all examples of this and are commonly used in standard image editing software. Figure 1 shows an example of a blur operation.

Figure 1: A blur algorithm is applied on an input image and produces a blurred output image.

Another possible application is when a function is used to extract information and features in an image. Figure 2 shows an example of a feature detection algorithm performed on an image of a face. The algorithm manages to locate features such as the nose, the mouth and the eyes. Although the image is processed, not everyone agrees that this is the typical application of image processing. Some people think these kind of problems belong to the computer vision field.

Figure 2: The input is analyzed to find features such as the nose, the mouth and the eyes.

1.1.2 The evolution of computing hardware

In the early 1960s, computers finally had enough computing power to perform meaningful image processing. The *Jet Propulsion Laboratory* processed images of the moon where they corrected image distortion caysed by the on-board camera on the space probe *Ranger 7*[1]. Later on, fields such as medical imaging and astronomy started to explore the field of image processing.

As the years passed by, the capacity of the computing hardware found in computers would keep improving. The term *Moore's Law*[2] was introduced based on a statement from the co-founder of Intel Corporation saying that the transistor count in integrated circuits would be increasing by a factor of two every year. In the early 1980s, a *Central Processing Unit*, CPU, ran with internal clocks operating around 1 MHz. Today, around 30 years later, most CPUs have clock speeds around 2-4 GHz, which are three magnitudes faster. The faster a processor's clock is operated, the faster a floating point computation can be performed. Figure 3 gives an overview of the transistor count and clock speeds of Intel CPUS[3].

Due to power and heat restrictions and the physical size of the current transistors, it is hard to keep improving the clock speeds. The focus has recently shifted towards parallel computing and multicore processing units.

1.1.3 GPU computing

Computer games started became more popular in the 1990s. Computing was the big bottleneck in 3D graphics and it was hard to produce good real-time solutions. Therefore, companies started to experiment with a new computing device, the *Graphical Processor Unit*, GPU. Instead of doing everything on the CPU, lighting computations and transformation of 3D coordinates could

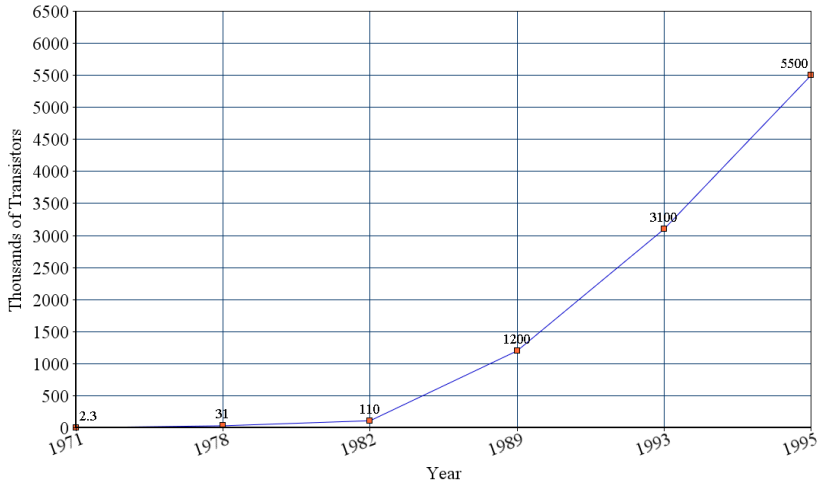

Figure 3: The CPU transistor count has been growing exponentionally.

now be done entirely on the GPU. Since the computations of each pixels could be calculated independently of the others, it motivated the use of parallelism.

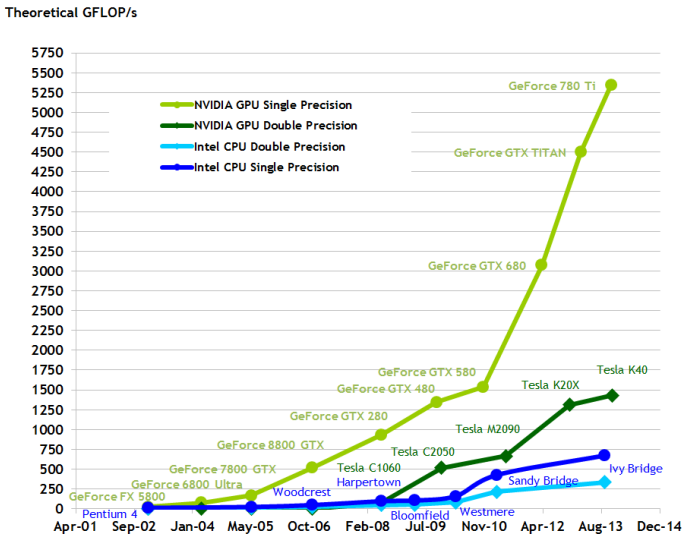

Figure 4: A comparison of theoretical floating point operations per second between CPUs and GPUs. This graph is from the CUDA Programming Guide[7].

When a GPU uses its full power and all computing units are used to maximum efficiency, a GPU is a lot faster than a CPU in terms of floating point operations per second. Figure 4 shows a comparison between the most recent CPUs and GPUs. The reason why the GPU has lot higher theoretical maximum is because it is specalized for compute-intensive and highly parallel computations. It is therefore designed in a way where more transistors are devoted for data processing rather than data caching and flow control (which a CPU handles a lot better)[4].

1.2 Purpose

Most image processing tasks are well-suited for parallel computing. The average image consists of millions of individual pixels. This is a good case for GPU computing where a lot of the compute units can be utilized at the same time. To get started with GPU computing, one has to choose a parallel GPU environment to write the code in. There are different environments available, each with its own pros and cons. Which one to choose can sometimes be difficult to decide since it requires testing. In some cases, it might not be clear which environment is the best overall since they all have unique subfeatures.

GPU computing can be split into two steps. The first step is the configuration phase where the GPU environment is instantiated, memory is allocated on the GPU, data is copied to the GPU and the image processing code is compiled to GPU machine code. The configurations vary from environment to enviroment and it can be tedious to do this setup in every program with GPU computing. This motivates the use of a framework for efficient image processing on the GPU. This thesis will focus on implementing this framework, supporting the common GPU environments. It will try explore and hopefully answer the following questions:

- What do the common GPU architectures have in common?
- How do you generalize GPU computing for image processing?
- Is it possible to write one functional framework although the environments and their architectures are different?
- What restrictions have to be made?
- Is it worth doing image processing on the GPU instead of the CPU?

The second part of GPU computing is implementing the actual algorithms that run on the GPU. This part is hard to generalize since every GPU environment has its own coding language and special features.

The work of this thesis will result in three different software components. They are the following:

1. **General backend framework**
    - C++ library
    - Cross-platform
    - Easy to disable components from compilation
    - Possible to dynamically rebuild GPU code at runtime
    - Support python bindings
    - Unit tests for all implementations and cases
2. **Graphical User Interface application**
    - Python
    - Graphical display of the image outputs
    - Quickly change parameter values
    - Support for live coding
    - Save an algorithm setup for later use, including code and parameters
3. **Command-line application**
    - Python
    - Run the algorithm setup saved from the GUI-version
    - Change parameter values through command-line arguments.

1.3 Limitations

As discussed in the previous chapter, there are different categories in image processing. This thesis will focus on the case where an input image is used to produce an output image, mainly because it is hard to generalize things like feature extraction across all GPU environments. The framework is instead going to support algorithms where an arbitrary number of input images are

going to produce an arbitrary number of output images. The number of input images and output images do not have to match. The framework needs to support multipass algorithms where multiple GPU programs are being called sequentially and where the output of one program can be the input of the following one. The framework is going support images with one to four channels of colors and where the data has either floating point precision or is of the unsigned byte type (often referred to as unsigned char).

The framework is going to support the following common GPU computing environments:

- **CUDA** - NVIDIA
- **OpenCL** - Khronos Group
- **GLSL** - OpenGL Shading Language

CUDA and OpenCL are the two most common choices today in the world of general-purpose GPU computing. Before OpenCL and CUDA, people used programmable shaders in 3D graphics libraries such as OpenGL and DirectX to perform image processing on the GPU. Since DirectX and their HLSL shading language and Direct Compute environment only are supported on Microsoft Windows systems, they are not included in the framework of this thesis. The framework is meant to be flexible and cross-platform, supporting Unix, Mac OSX and Windows operating systems.

1.4 Method

The first phase of the thesis work will involved individual work with CUDA, OpenCL and GLSL to gain more knowledge about the different GPU environments. The middle phase of the project was the implementation of all the software components. This was be an iterative process. First a prototype was be created and after feedback from a supervisor and other people involved in the project, a second improved version was be implemented. For every feature added in the backend library, a unit test was added to test the different cases. The last part of the thesis work involved testing and more specifically, testing with practical image processing problems. The testing part served the purpose of benchmarking performance, testing how practical the framework is and examined if any GPU environment is better than the others in terms of functionality.

2 GPU environments

2.1 GLSL

2.1.1 OpenGL

OpenGL is a cross-platform API that has been an industry-standard ever since it was introduced in 1992[5]. Major decisions are made by the OpenGL Architecture Review Board, ARB. ARB was created by Silicon Graphics in 1992 and had representatives from SGI, Intel, Microsoft, Compaq, Digital Equipment Corporation, Evans & Sutherland and IBM. Later on, companies such as 3Dlabs, Apple, ATI, Dell, IBM, Intel, nVIDIA and Sun Microsystems were added. OpenGL shares many characteristics of a previous API called Iris GL. It is designed in a way where it tries to be the lowest level interface for accessing graphics hardware but still provide hardware independence. OpenGL is supported in PC, Mac and Unix-systems.

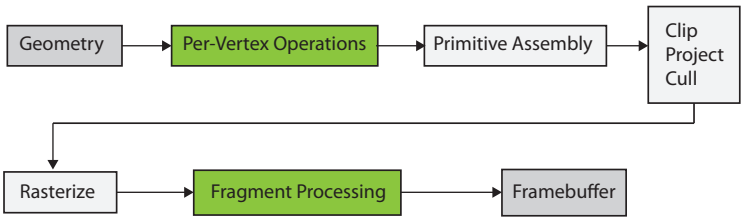

Figure 5: OpenGL orginially had a fixed pipeline where non of these steps could be modified.

Figure 5 shows the overview of the complete OpenGL pipeline in version 1.5 and earlier. It was said to have fixed functionality because every OpenGL implementation was required to have the same functionality. The set of operations and how they were applied were defined by the fixed OpenGL specification. The Fragment Processing step in Figure 5 is where each fragment gets its final value. A fragment can be thought of as the data needed to both shade the pixel decide if the fragment is to be displayed as a pixel (need information about depth, alpha). The fixed fragment stage could only handle tasks such as interpolating color values, texture mapping and fog.

2.1.2 The shading language

In version 2.0, OpenGL introduced GLSL, the OpenGL shading language. With the OpenGL shading language, the fixed functionality stages for vertex and fragment processing (green steps in Figure 5) could now be customized

and programmed[6]. It was still possible to do everything that the previous fixed pipeline supported but it also gave the software developer the opportunity to alone control the output. The programs written in GLSL are called OpenGL shaders, vertex shaders or fragment shaders. The OpenGL Shading Language is a high-level procedural language based on C and C++ syntax and flow control. Vector and matrix types/operations is natively supported together with a set of math functions commonly used in graphics shading.

2.1.3 Shaders

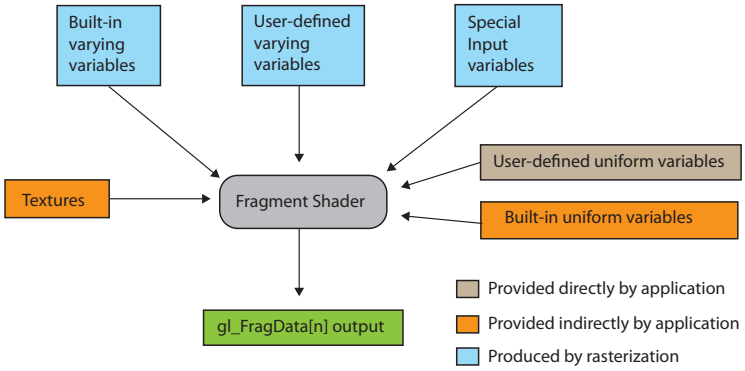

Figure 6: The inputs and outputs of a GLSL fragment shader.

Shaders are compiled from an input source of text at runtime. They are later linked to an OpenGL program and become executable. In image processing, only the fragment shader is of importance. In a fragment shader, any image processing algorithm can be applied on an input image. A fragment shader operates on one fragment at a time. Fragment shaders must be written in such a way that they can operate simulationeusly. When a fragment shader is being executed, it has no knowledge about other fragments and their data.

Figure 6 shows the inputs and outputs of a fragment processor. Some variables are built-in, specified by the OpenGL implementation. There is a notion of varying and uniform variables. Uniform variables are, as the name suggests, uniform across all shaders. Varying variables are defined per vertex in the vertex shader. Before processing each fragment, the hardware interpolates the geometry and gives the fragment shader the correct varying attributes. An example of a varying attribute is texture coordinates. The texture coordinates are defined at each vertex. Before the processing

of the fragment shader, the texture coordinates are interpolated across the geometry and can be used later in the fragment shader to do texture lookups.

```
uniform sampler2D texture0;
uniform sampler2D texture1;

uniform float scale;
varying vec2 texcoord;

void main()
{
    gl_FragData[0] = scale * (
        texture2D(texture0, texcoord) +
        texture2D(texture1, texcoord));

}
```

Code 1: Example of adding two images in GLSL

Code 1 shows an example of fragment shader in GLSL that adds two images together. The images have been converted to OpenGL textures, `texture0` and `texture1`. The function `texture2D` together with the varying variable `texcoord` is used to fetch the data from a texture. The value is multiplied with the uniform value `scale` (same for all fragments) and finally written to the framebuffer through the built-in `gl_FragData`. Syntax is similar to a C program and every fragment shader needs a `main` function.

2.2 CUDA

The CUDA architecture was released for the first time in 2006 to make it easier to perform general purpose GPU-computing[7]. Unlike previous GLSL methods that had to get around the shaders pipeline, CUDA allowed every aritmethic logic unit on the chip to be controlled by a CUDA-program. Another important feature was the possibility to read and write to arbitrary memory address on the GPU in comparison to previous methods that required textures as storage. The hardware was still in charge of memory caching but CUDA also exposed a software managed cache called shared memory. CUDA programs are written in the CUDA C language, a language very close to the C language with the exception of a small number of keywords added for special features in the CUDA architecture.

2.2.1 Blocks and threads

In the CUDA architecture, threads are single execution units that run kernels on the GPU. They are similar to CPU threads but the typically there are a lot more of them on the GPU. The threads are divided into thread blocks. Threads within a thread block can communicate with each others. The number of blocks and threads per block is decided by the developer when the kernel is called. The grid of thread blocks can be one, two or three dimensional. The maximum number of blocks and threads per block is decided by the GPU and its hardware. A CUDA kernel is executed simultaneously by warps. A warp consists of threads within a block. Typically each warp has the size of 32 threads, where actions like memory read and writes are performed in half-warps.

```
__global__ void VectorAddition(float * A,
                               float * B,
                               float * C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int idx = i + j * gridDim.x * blockDim.x;
    C[idx] = A[idx] + B[idx];
}
```

Code 2: Example of vector addition in CUDA

When a thread is executing a kernel, there are built-in variables to access information about which block the thread belongs to and the local thread

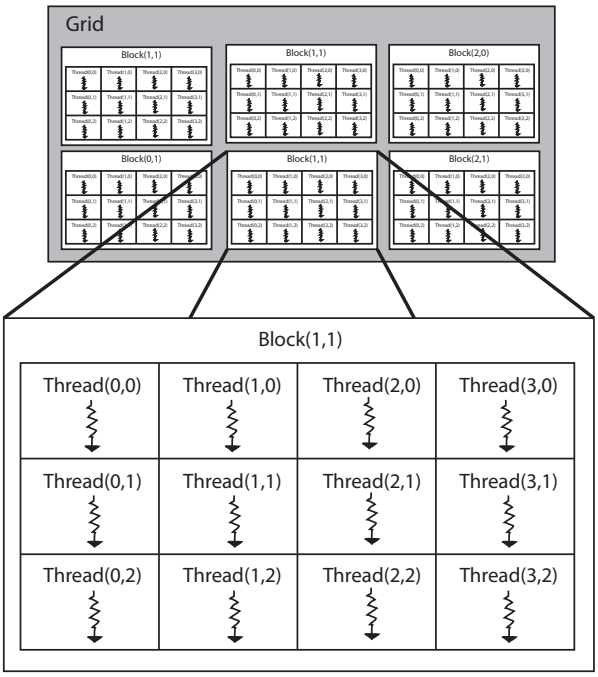

Figure 7: CUDA threads divided into blocks of threads.

index in the actual block. This information is often used to decide indices in global arrays for read and write operations. Code 2 shows an example of a simple CUDA kernel performing a vector addition. The keyword `__global__` is used to tell the compiler that the function is a CUDA kernel. The variables `blockIdx`, `threadIdx`, `gridDim` and `blockDim` are automatically built-in in a CUDA kernel and can be accessed at any time. `blockIdx` and `threadIdx` are of the type `uint3`, where each value represent an index in the corresponding dimension. `blockIdx` is the index of the block in the total grid of launched blocks and `threadIdx` is the thread index within the block. Both `gridDim` and `blockDim` are of the type `dim3` and are constant in all threads. It is not possible to launch a kernel with different block sizes.

2.2.2 Memory hierarchy

There are different memory spaces in the CUDA architecture as can be seen in Figure 8. Each thread has a local and private memory space. All the threads in a block share a memory space called shared memory. Shared memory is on-chip and is divided into different banks. Reading from shared memory in warp of threads is just as fast as reading from a register as long as there are no bank conflicts. Each bank conflict results in a serialized read

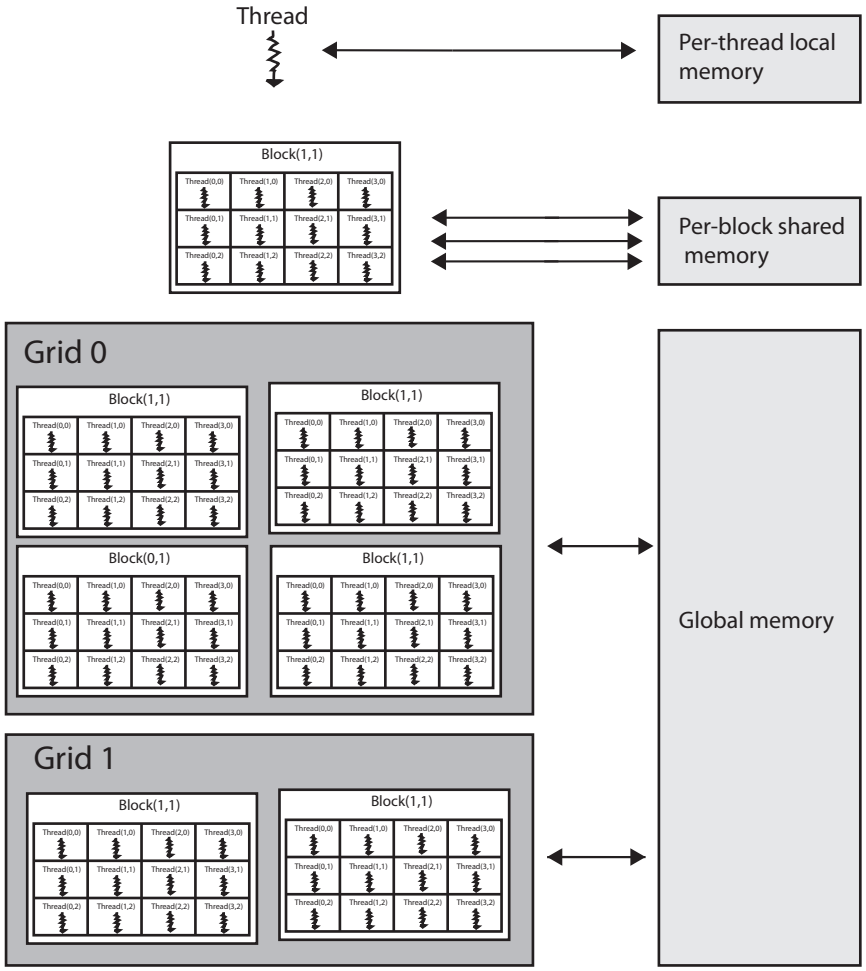

Figure 8: Different CUDA memory spaces

from the shared memory. A worst case scenario is when all threads in a warp only read from one bank in the shared memory. The shared memory read then becomes a lot slower then the regular global memory. Global, constant and texture memory can be accessed by all threads. Constant and texture memory are only used in certain cases while the global memory is the common choice for storing data. The texture memory space is cached in a way where a texture fetch only cost a GPU device memory read on a cache miss, otherwise it only costs a read from the texture cache. Texture cache is optimized for 2D spatial locality. If desired, a read from the the texture memory gets automatic linear interpolation. Constant memory is a read-only memory space. All the threads of a half-warp read from constant

memory just as fast as from a register as long as all the threads read from the same address. The cost scales linearly with the number of different addressed read by the threads. For example, the worst case would be if an array would be stored in the constant memory space and each thread in an executing half-warp needs a unique value in this array. The global memory space is capable of reading 4, 8 and 16 bytes of memory into registers in one single instruction. The most efficient global memory reads are when all threads in a half-warp reads from continous memory in a coalesced read of 64, 128 or 256 bytes. Within in a kernel, it is not possible to read and write to the same array. CUDA supports atomic operations but they are usually slow and the more common choice is to allocate multiple buffers, one to read from and one to write to.

2.3 OpenCL

2.3.1 Standardized framework for heterogeneous systems

OpenCL is a parallel programming framework designed to fit heterogeneous systems where one can expect a range of different computing architectures[8]. An example of a program on a heterogeneous system would be a program where some parts of the computations and setups are done on the CPU and the rest on the GPU. OpenCL also supports parallel programming for homogeneous systems. In a case where one has a multi-core CPU, OpenCL can be used to have one thread control the state of the program while the rest of the threads performs computations and later sync with the main thread.

OpenCL is standardized by the Khronos Group, the same group in charge of the well known OpenGL standardization. The group consits of people from many different companies in the industry such as AMD, NVIDIA, intel, Apple, Samsung. They decide the direction the group is taking and make sure the framework is compatible for different system and platforms. One of the goals of OpenCL is to be as flexible as possible.

2.3.2 The OpenCL C language

OpenCL can be used in any parallel environment as long as the OpenCL compiler and runtime library is implemented. When writing the parallel code, a software developer do not have have care about operating system, processors and memory types. The OpenCL C language is very similar to the regular C language. It is focused around computations and some fea-

tures are added on top of the C language to simplfy things, like SIMD vector operations and multiple memory hierarchies. Other features, such as printing, have been removed as they are not as useful in computing and hard to implement on all platforms. The program calling the OpenCL code can be written in either C or C++ and will be using the OpenCL runtime library.

The word *host* is often used in standard and official OpenCL literature. The host refers to the environment where the OpenCL code is called from (not executed). This is the CPU in almost all of the cases. An OpenCL device is the environment where the OpenCL is executed. GPU, DSP, CELL/B.E, CPU are some examples of OpenCL devices that contain a lot of small compute units. The memory associated with these processors is also included in the definition of an OpenCL device. The OpenCL code executed on a device is called *kernel*.

2.3.3 Memory Hierarchy

Similar to the CUDA architecture, OpenCL also has a memory hierarchy. The OpenCL standard only specifies the access levels of the different memory spaces and there may be important performance details that are different on different hardware implementations. It possible to optimize the code for a certain hardware or vendor, but it is harder to generalize and write programs with high performance across different devices.

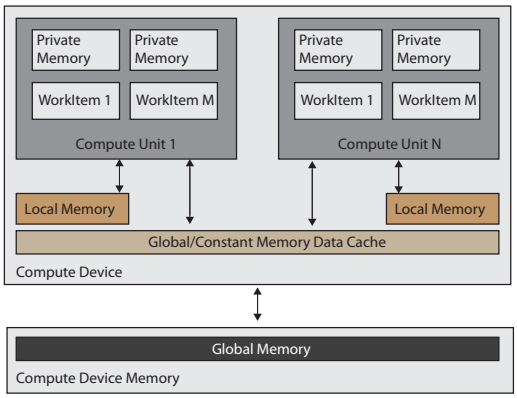

Figure 9: OpenCL memory hierarchy.

- **Global Memory** - The global memory has the largest capacity and can be used by all work items. It is considered being the slowest memory

subsystem. The best performance is achieved when streaming contiguous memory addresses or patterns that can explore the full bandwidth (similar to the coalesced memory reads in CUDA).

- **Private Memory** - The memory used in a single work item. Can not be shared between work items. Similar to registers in GPU multiprocessors or CPU cores. The private memory is allocated and partioned at compile time. There is no maximum private memory size defined in the OpenCL specification. Using too much private memory can lead to a slowdown since OpenCL will user slower memory spaces once the private memory is full.

- **Local Memory** - Local memory can be shared between work items in a work group, similar to the shared memory in the CUDA architecture. Local memory is used when data from global memory is needed and one wants to reduce the global memory reads within a work group.

- **Constant Memory** - The constant memory implemention differs across hardware. For example, on NVIDIA GPU cards, the constant memory is located at region good for broadcasting. On ATI GPU cards, the constant memory is part of the global memory but with optimized broadcasting.

2.3.4 Work Groups

Kernels are being executed over a 1D, 2D, 3D grid or NDRange. The kernel is then executed in parallel where each kernel instance is called work item. Work items are divided in work groups of the global grid. The developer can explicitly set the size of the work group or let it be decided at runtime.

```
1  __kernel void VectorAddition( __global float * A,
2                                __global float * B,
3                                __global float * C,
4                                int width)
5  {
6      const int x = get_global_id(0);
7      const int y = get_global_id(1);
8      const int idx = x + y * width;
9      C[idx] = A[idx] + B[idx];
10 }
```

Code 3: Example of vector addition in OpenCL

Code 3 shows an example of a simple vector addition. The keyword `__kernel` tells the compiler it is a OpenCL kernel and `__global` specifies a pointer to the global memory space. Inside the kernel, the function `get_global_id` is used to find the horizontal and vertical id of the work item (this only works if the kernel is executed with a two dimensional work size).

3 Implementation

3.1 Designing the core library

On an abstract level, the workflows in the differnet GPU environments are more or less the same. This motivates the use of a library where a developer can target a common abstract interface and not have to worry about specific GPU environment implementations. The library created in this thesis is named *gpuip* and stands for *GPU Image Processing*. The core superclass in *gpuip* is called `ImageProcessor` and it is through this class that all the GPU communication is going to happen. An implementation of an `ImageProcessor` needs to support the following common steps:

1. Compile kernel text code and build into GPU machine code.
2. Allocate memory on the GPU.
3. Copy data from CPU to GPU and vice versa.
4. Run kernels on the GPU.

These steps are all virtual functions in the `ImageProcessor` class that needs to be implemented in the subclasses. The class `Buffer` provides memory allocation information to the `ImageProcessor` class. An algorithm can have an arbitrary amount of buffers. The input and outputs are stored in buffers. Each buffer has a unique name, information about data type and how many channels there are per pixel. The supported data formats are `unsigned byte`, `half` and `float`. `half` is a 16-bit floating point scalar (compared to the standard 32-bit `float`). Each buffer can have between one to four channels per pixel. Two or three channels per pixels is supported but not recommended as it leads to uncoalesced reads and writes[9].

```
1 struct Buffer{
2     typedef shared_ptr<Buffer> Ptr;
3     enum Type { UNSIGNED_BYTE, HALF, FLOAT };
4     const string name;
5     Type type;
6     unsigned int channels;
7 };
```

Code 4: Buffer struct

The `Kernel` struct provides the following:

1. GPU kernel code. Syntax and program structure depends on GPU environment.
2. Which buffers are going to be used as input? What are they called in the kernels?
3. Which buffers are going to have data written to them?
4. Parameters used in the kernel code.

```
struct Kernel {
    struct BufferLink {
        Buffer::Ptr buffer;
        string name;
    };
    typedef shared_ptr<Kernel> Ptr;
    const string name;
    string code;
    vector<BufferLink> inBuffers;
    vector<BufferLink> outBuffers;
    vector<Parameter> params;
};
```

Code 5: Kernel struct

As can been seen in Code 4 and Code 5, both structs have a `Ptr` typedef. A `Buffer::Ptr` is a shared pointer[10] to a `Buffer` object. To register a `Buffer` or a `Kernel` to an `ImageProcessor` object, the factory method pattern[11] is used. An example of this can be seen in Code 6.

```
class ImageProcessor {
    ... rest of ImageProccessor definitions ...

    Buffer::Ptr CreateBuffer(const string & name,
                             Buffer::Type type,
                             unsigned int channels);

    Kernel::Ptr CreateKernel(const string & name);
};
```

Code 6: Factory functions to create Buffer and Kernel objects

In this case, the factory method pattern guarantees that the `ImageProcessor` is the *owner* of the buffers and kernel objects since it is going to store a copy of the shared pointer internally. As long as the `ImageProcessor` exists, the `Buffer` and `Kernel` objects are also guaranteed to exist. This makes it impossible to reach cases where the `ImageProcessor` is using pointers that are pointing to deleted objects. Anyone using the library can still create and delete `Buffer` and `Kernel` objects on their own, but they will never be able to register these objects themselves.
To make things simple, all GPU operations are synchronous which means that once a function that communicates with the GPU has been called, it is not going to return until the GPU is done proceeding the task. There are asynchronous ways of calling the GPU in all environment but they require syncing stages. As discussed in [9], it might be worth supporting asynchronous calls in the future since they could potentially give better performance.

```
class ImageProcessor {
    ... rest of ImageProccessor definitions ...

    virtual double Allocate(string * error);
    virtual double Build(string * error);
    virtual double Run(string * error);
    virtual double Copy(Buffer::Ptr buffer,
                        CopyOperationEnum operation,
                        void * data,
                        string * error);
};
```

Code 7: `ImageProcessor` API for GPU operations

As can be seen in Code 7, all functions that perform GPU operations returns the execution time as a `double` and takes a pointer to an error `string`. If an error occurs inside one of these functions, a *negative* value is returned and the error message can be found in the error string.

3.2 Implementing the subclasses

3.2.1 GLSL

The user-provided kernel code is a fragment shader. To build a GLSL program, one also needs to provide a vertex shader. The built-in vertex shader in gpuip is simple and draws a 2D quad across the viewport, covering all pixels, and defines texture coordinates in each corner. These texture coordi-

nates, as mentioned in Chapter 2.1, will be interpolated across all fragments. Since all fragment shaders use the same vertex shader code, it is compiled once and shared between all fragment shaders.

For each buffer, an OpenGL texture is generated and memory is allocated on the GPU. For each kernel, a framebuffer object is created. Depending on the kernel setup, every output buffer of a kernel is mapped the corresponding framebuffer object. This means that later on when the simple quad is drawn, the data is rendered to the textures directly. To copy data from and to the GPU, the synchronous functions `glGetTexImage` and `glTexImage2D` are used.

The input textures (buffers) and parameters has to be set before the GPU kernel code can be executed. Each uniform attribute has a location in the GLSL program. To get the location, `glGetUniformLocation` is used. It is important that the input textures and parameters in the kernel code are named the same as the `Kernel::BufferLink::name` and `Kernel::params`, otherwise the GPU code will not run. Before calling the draw functions, the viewport has to be resized to the same resolution as the output buffers. The GPU calls are synchronized with `glFinish` and timings are queried with `glGetInteger64v` and `GL_TIMESTAMP`.

3.2.2 OpenCL

OpenCL needs a context for saving states and an event queue to register GPU operations in. Allocating memory is done through OpenCL buffers and kernel code is compiled at runtime with the standard OpenCL API calls `clCreateProgramWithSource`, `clBuildProgram` and `clCreateKernel`.

When executing the kernel code, the kernel arguments have to be passed in the same order as they are presented in the kernel function. Kernel arguments include pointers to input and output buffers, user-defined parameters and gpuip parameters. Since the C++ code is bound to compile time and the kernel code is written at runetime, there have to be rules that decides which order arguments appeaer in. The current ordering rules are:

1. `const` pointers to input buffers
2. Pointers to output buffers
3. User-defined arguments

4. Gpuip-arguments (like image width and image height)

Launching the kernels can only be done asynchronously. To guarantee that all GPU computation is done when the function returns, the OpenCL state is synced with `clFinish`. Timings are captured with the `clGetEventProfilingInfo`.

3.2.3 CUDA

Unfortunately, CUDA does not support compilation of GPU code at runtime which was one of the goals of gpuip. However, it does support loading of ptx, parallel thread execution, at runtime. Ptx is a pseudo-assembly language used by NVIDIA on the GPU. To get ptx files, we must use the NVIDIA-provided CUDA compiler *nvcc*. Nvcc is called by `popen` (`_popen` on windows) which creates a pipe and invokes a shell command. When closing the pipe with `pclose`, the exit status of nvcc can be queried. If the exit status is anything other than zero, the compilation failed and the error string can be read from the pipe.

CUDA comes with a driver API and a runtime API. The runtime API is user-friendly and the driver API gives more control. They can both be used at the same time. Loading compiled ptx files and execution of kernel calls are done through the driver API and the runtime API is used for context creation, memory allocation and data transfer. Allocating memory on the GPU is straightforward with the `cudaMalloc` function, which is very similar to the C version `malloc`. Copying data is also simple with `cudaMemcpy`.

CUDA kernels are executed in blocks with a fixed amount of threads per block. In gpuip, every block consists of 256 threads distributed in a 16x16 thread-block. To make sure every thread corresponds to one pixel, we use the following equations to determine the number of blocks N_x and N_y:

$$N_x = floor(W/16) + 1,\ N_y = floor(H/16) + 1 \tag{1}$$

where W is the image width in pixels and H is the image height. Some threads will have xy-coordinates outside of the image domain. To avoid writing to non-allocated memory, all threads need to check if they are inside the image domain.

3.3 Python bindings

Boost Python[14] is used to make the C++ code accessible in a python environment. When copying data from and to the GPU, one has to pass a

void pointer in the gpuip C++ API. The concept of pointers does not exist in python. Instead, in the python bindings, the CPU data is attached to the buffer itself using a numpy [array]. All the data transfers between GPU buffers have to go through the numpy array.

A common task in image processing is to read image data from disk and later on write to disk once the processing is done. To simpify this step, read and write functions are included in the python bindings. Depending on the per element data in the numpy array, different file formats are available. For `half` and `float` precision, the target format is OpenEXR[16]. When the data consists of unsigned bytes, the more common image formats png, jpeg, tiff and tga are available through the header-only library CImg[15].

3.4 Graphical user interface

The graphical user interface application is written in python since it often means faster development iterations. To fit the cross-platform requirements, the UI framework Qt[12] is used. The core of Qt is written in C++ but there are python bindings available. Gpuip uses PySide[13] since they are well documented and supported on the official Qt homepage.

The application will be a `QMainWindow`. Main windows in Qt support menus and toolbars. All the different components will be dock widgets. A dock widget is resizeable and can be detached to a solo window. The following components will be added as dock widgets:

1. Toolbar. Add menu items and toolbar items as `QAction`. It is possible to bind an action to a hotkey.

2. Preview. Display the content of a buffer using `QGLWidget`. Supports zoom and pan. If the image has floating point precision, a slider controlling the exposure is added. This is based on the same display algorithm as exrdisplay[17].

3. Code. This is a `QTextEdit` containing the kernel code. When building a kernel, gpuip reads the text from this widget.

4. Params. Per kernel setup with `QComboBox` dropdown menus for buffer selection and editing parameter value with `QSlider` and `QLineEdit`.

5. Log. Output log for all commands using `QTextBrowser`.

3.5 Cross-platform build

3.5.1 Generate build setup

Every platform has their own way of compiling source code into binaries. On Unix systems, gcc and makefiles is the common option while on Windows systems most code is compiled with Microsoft Visual Studio. CMake is a cross-platform build system by Kitware[18]. CMake controls the software compilation step using platform independent configuration files. Option variables and cached string values can be defined in the configuration files and later on be modified at either the command line version of CMake or the gui version. This makes CMake a very powerful tool to setup customizable builds. For example, in gpuip, one can easily disable the build of the python bindings if it is not needed. Another case could be if the compiling system does not have a NVIDIA GPU and want to build gpuip without CUDA support. Once all options are set, CMake generates either Unix Makefiles, a Microsoft Visual Studio solution or other build setups that are already configured. This means all the include paths for header files have been set and linking to other libraries is taken care of.

3.5.2 Library dependencies

Gpuip and especially its python bindings part depends on other open source libraries. When the compiler is invoked, information about where these libraries are located has to be passed. The locations can be vary a lot in different setups and it is hard to come up with a solution that is going to work nicely across all platforms. Luckily, CMake has a nice feature called `FindPackage` where it is possible to register scripts to find external libraries. The most common libraries and their `FindPackage` script are shipped with CMake. Some of the libraries used by gpuip were not recognized by `FindPackage` in CMake and scripts for finding them were added.

It can be annoying to prepare all the prerequisites when building a library that depends on a lot of other libraries. To simplify this step, gpuip tries to make the build process as out of the box as possible. If a third party library is missing and it is an open source library, CMake will try to download the missing library at compile time and build it. This can be done through the `ExternalProject_Add` feature where one specify the path to a git or svn repository where the open source code exists. It is also possible to specify specific configure, build and install command if the open source library does not use CMake as build system.

3.5.3 Regression testing

CMake comes with ctest, which is a tool that can be used for testing the code after building it. Gpuip has three differnet tests: One for testing the standard API calls in C++, one for testing the standard API calls in the python bindings and one that compares performance of gpuip vs cpu implementations (both single and multi-threaded).

3.5.4 Documentation

An API documentation is generated at build time (if the option is enabled) with the help of Doxygen[19]. Doxygen reads the comments of the header files and generates reader-friendly html version to publish online.

4 Results

4.1 Performance

The main reason to do image processing on the GPU and not on the traditional CPU was to gain better performance. Therefore, four tests are added to compare the different GPU environments against a non multithreaded CPU implementation. The four tests are *Linear Interpolation*, *Box Blur*, *Gaussian Blur* and *Separable Gaussian Blur*. All blur algorithms use the same size to decide neighboring pixels. The algorithms can be found in Appendix A. The CPU code was executed on a machine with a Intel Xeon Processor(12M Cache, 2.80 GHz, 1600 MHz FSB) and the GPU kernels were executed on a NVIDIA GTX 760 card. The timings can be seen in Table 1. In the GPU timings, the time it takes to copy input data to the GPU and output data to the CPU is included in the timings.

Table 1: Performance timings

Algorithm	**OpenCL**	**CUDA**	**GLSL**	**CPU**
Linear Interpolation	9.6 ms	11.7 ms	46.2 ms	38.0 ms
Box Blur	21.0 ms	24.3 ms	34.4 ms	3024.0 ms
Gaussian Blur	21.1 ms	25.1 ms	34.6 ms	7650.0 ms
Separable Gaussian Blur	7.9 ms	11.3 ms	31.15 ms	1802.0 ms

Table 1 shows that in all cases except the linear interpolation, the GPU is a lot faster. The linear interpolation is fast on the CPU since the memory fetches are linear and can be predicted ahead of time (something most CPU caches do). It is interesting to see that the Box blur and Gaussian blur are about as fast on the GPU, while the Gaussian is twice as slow on the CPU. This is because the hardware of the GPU is designed to be faster at computations and have special components to perform operations such as exponentials, logarithms, sin and cosine.

Table 2: Average speedup factor

	OpenCL	**CUDA**	**GLSL**
Speedup factor	184.7	148.0	91.9

Table 2 shows the average speedup factor for each GPU environment. The speedup factor is computed by $time_{CPU}/time_{GPU}$. The speedup factor is not a reliable metric as it is very case dependent but at least it shows that OpenCL seems to be faster than CUDA and GLSL in these test cases and that image processing on the GPU can be about two magnitudes faster than the CPU.

4.2 Graphical User Interface application

The GUI application can create new, open old and save `.ip` files. A `.ip` file is a XML-based textfile. Figure 10 shows a screenshot of the gpuip GUI application where the file `lerp_opencl.ip` has been opened. A toolbar with icons can be seen at the top for quick interactions. To the left is the preview of one of the buffers, *buffer2*. The exposure slider is available in the Display tab since the data type of *buffer2* is `half`. The syntax highlighted kernel code can be found in the middle of the GUI with corresponding kernel settings to the right. At the bottom is the log output. Figure 12-15 show different GUI components.

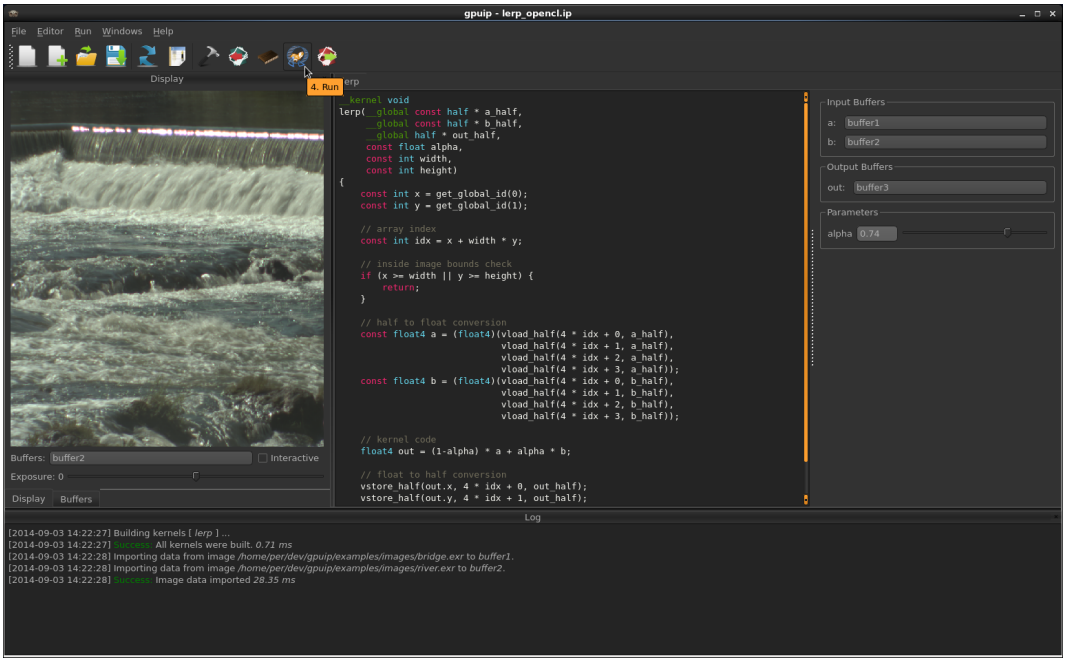

Figure 10: The gpuip GUI application.

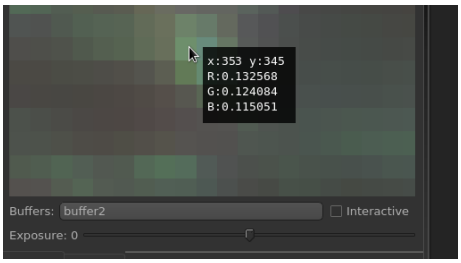

Figure 11: Debug info pops up by right-clicking on the image in the Display view.

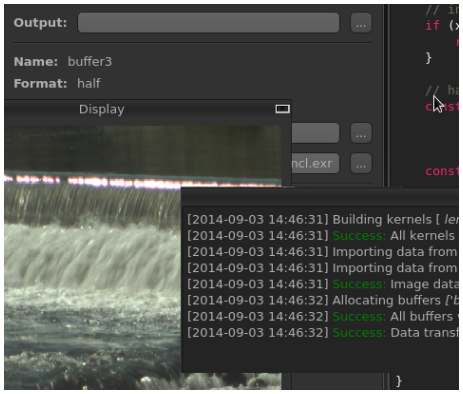

Figure 12: All widgets can be detached.

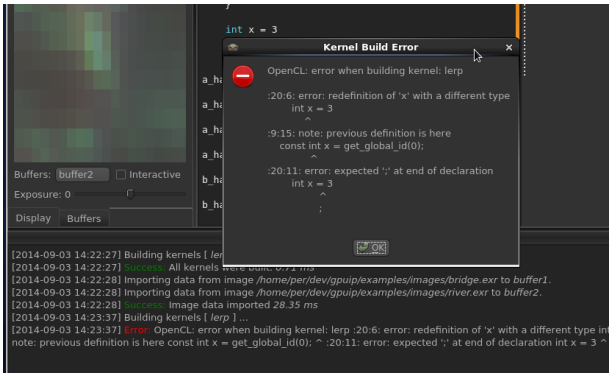

Figure 13: Example of error feedback when compiling.

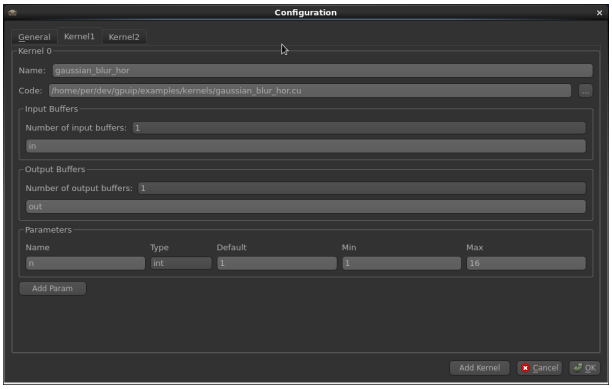

Figure 14: Example of the configuration step when creating a new `.ip` file.

4.3 Command-line application

Gpuip can be called from the command line by adding `--nogui` somewhere in the `gpuip` command. An `.ip` file generated by the GUI version is required as input (`-f FILE` option). It is possible to change values of buffers and parameters at the command line. Code 8 shows the output of the help command. Appendix B shows an example of running gpuip from command line with the verbose option enabled.

```
\$> gpuip --help
usage: gpuip [-h] [-f FILE] [-p kernel param value]
             [-i buffer path] [-o buffer path] [-v]
             [--timestamp] [--nogui]

Framework for Image Processing on the GPU

optional arguments:
  -h, --help            show this help message
  -f FILE, --file FILE  Image Processing file *.ip
  -p kernel param value, --param kernel param value
                        Change value of a parameter.
  -i buffer path, --inbuffer buffer path
                        Set input image to a buffer
  -o buffer path, --outbuffer buffer path
                        Set output image to a buffer
  -v, --verbose         Outputs information
  --timestamp           Add timestamp in log output
```

```
19  --nogui                    Command line version
```

Code 8: gpuip command-line application

5 Discussion

5.1 Comparison between the GPU environments

All GPU environments have both their pros and cons. After working on the implementation of each GPU environment, these are the my summarized conclusions:

- OpenCL
  - + Supports runtime compilation.
  - + Works on any GPU and any CPU.
  - − Poor 16-bit floating point support.
  - − Does not support templating.
- CUDA
  - + Easy to use with runtime API.
  - + Debugging tools.
  - + Supports templating and operator overloading.
  - − Does not support runtime compilation.
  - − Only works on NVIDIA GPUs.
  - − Poor 16-bit floating point support.
- GLSL
  - + Works on any GPU. OpenGL is include in a lot of systems by default.
  - + Supports runtime compilation.
  - + Abstract way of dealing with data.
  - + Display functionality for free.
  - − Setup is not clean. Need to fake render a quad.
  - − Can only write to one pixel at a time.
  - − No local shared memory between work items.

If I had to pick only one GPU environment to use in a new application, I would choose OpenCL because it is very flexible and runs both on CPU and the GPU. GLSL would be an okay choice as long as the application is about image processing. GLSL should work out of the box on most systems and that

makes it very easy to deploy the application to other parties. However, for more general purpose computing, GLSL quickly becomes bulky and requires a lot of tricks. Writing OpenCL and CUDA kernels is about the same and the setup in CUDA is actually easier than OpenCL. The limitation of NVIDIA-only GPUs is too big of a factor to me and if NVIDIA would make CUDA run on all GPUs, I think CUDA would increase a lot in popularity. All the tests in this thesis were very basic and there might be a possibility that CUDA is the best environment to use once you want to optimize further and really get the most out of the GPU.

5.2 OpenGL interoperability

Currently, if an image has been created/modified with gpuip and needs to be displayed, it first has to be copied back to the CPU and then uploaded back to the GPU for viewing. Both OpenCL and CUDA support OpenGL interoperability by mapping a GPU buffer to an OpenGL buffer. This means that data produced by OpenCL and CUDA can be used as an OpenGL texture without unnecessary transferring between the CPU and the GPU. Although more internal work, the public API for `Buffer` would not change much:

```
struct Buffer{
  // ... rest of Buffer declarations
  bool glInteroperability;
  GLint glTexture;
};
```

Code 9: gpuip OpenGL interoperability

I think this option would make the library more lucrative to use in applications that are using OpenGL for viewing graphics. One particular case I could see it being useful is in deferred rendering for realtime 3D graphics. Once geometry has been rendered to different textures, it might be faster to apply operations in OpenCL or CUDA than it is in GLSL (there is no support for sharing memory between execution threads in GLSL).

5.3 Template kernels

Consider a case where you only want to write one kernel file but support multiple fileformats. For GLSL, this is already true and quite practical. However, it is not possible in OpenCL and CUDA. CUDA supports templated functions. CYDA only supports `half` storage and not computation and it is therefore not possible to template a function and have it work the same with

`half`, `unsigned byte` and `float`. In OpenCL, half computation is supported if the graphics drivers come with the `cl_khr_fp16` extension. OpenCL does not support templates but since we parse the kernel code ourselves in gpuip, we could implement our own templating rules and generate one OpenCL kernel for every data type that gpuip supports.

5.4 Computations as input

It is only possible to write data per pixel in gpuip. If an algorithm would require a global property of a buffer, like the maximum or average value, it would not be possible. For example, if someone wants to write a tone mapping algorithm, they might compute a tone mapping value per pixel and then use the average value of all pixels as input to a second step in the tone mapping. A kernel can have user-defined parameters but not parameters that depend on the output of a kernel. It might be nice to add an option to make it possible to use the computation of a buffer as input. To make things still fairly simple, it could be restricted to only allow computations of one-dimensional buffers. Then a computation could support operators such as min, max, median, avg and sum. Internally, gpuip would perform the GPU algorithms. For example, to get the minimum value of a one-dimensional buffer, it would have to run a reduce algorithm. It might be worth exploring the common libraries for the GPU computing. Using libraries like Boost Compute[20] and Thrust[21] would save time and probably have better performance.

5.5 Work item distribution

Some algorithms might be optimized further by allowing a work item/thread to write to more than one pixel. For example, in separable blur algorithms, it might be worth splitting the algorithm in two steps and have each work item operate on a row/column alone. Memory lookups tend to be the expensive part of an algorithm and if a work item can work alone on a row, data could be stored in the local registers as the work item iterates over the pixels. A different work item distribution is not possible in GLSL where every kernel has to be executed on a per pixel level.

6 Conclusion

The GPU environments available today are implemented differently but the main idea is the same in all of them:

1. Write GPU code and compile
2. Allocate GPU memory and transfer from/to CPU
3. Run GPU program

To create a GPU framework for image processing, the public interface has to support these steps. Memory allocation and transferring is simplified if inputs and outputs to the algorithms are restricted to images only. Supporting other kinds of inputs and outputs is possible but makes it harder to generalize. Some functionality only exists in some GPU environments and it is up to the framework to decide if features that do not exists in all environments should be added or not. All three environments tested in this thesis had their pros and cons and which one to choose is case-dependent.

Using a GPU framework is worth it if an image processing algorithm requires per pixel computations and performance is important. The more computation required, the faster the GPU is compared to the CPU. The CPU only performs well in cases where there is little computation and the memory fetch pattern is linear.

References

[1] R. Woods, R. Gonzalez
Digital Image Processing.
3rd Edition, 2007

[2] R Schaller
Moore's law: past, present, and future.
1997

[3] *Moore's Law for Intel CPUs.*
http://www.physics.udel.edu/ watson/scen103/intel.html [2014-09-04]

[4] J. Sanders, E. Kandrot
CUDA By Example.
1st Edition, 2010.

[5] OpenGL ARB, D. Shreiner, M. Woo, J. Neider, T. Davis
OpenGL Programming Guide: The Official Guide to Learning OpenGL.
5th Edition, 2005.

[6] R. J. Rost
OpenGL Shading Language.
2nd Edition, 2006.

[7] NVIDIA
CUDA C Progamming Guide.
http://docs.nvidia.com/cuda/cuda-c-programming-guide/ [2014-09-04]

[8] Khronos OpenCL Working Group
The OpenCL Specification.
Version 2.0, 2014.

[9] NVIDIA
CUDA C Best Practices Guide.
http://docs.nvidia.com/cuda/cuda-c-best-practices-guide [2014-09-04]

[10] Gregory Colvin
Exception Safe Smart Pointers.
C++ committee document 94-168/N0555, 1994.

[11] E. Gamma, R. Helm, R. Johnson, J Vlissides
Subject Design patterns, software engineering, object-oriented programming.
1st Edition, 1994.

[12] Digia plc
Qt Project.
http://qt-project.org/ [2014-09-04]

[13] *PySide, Python For Qt.*
http://qt-project.org/wiki/PySide [2014-09-04]

[14] *Boost Python.*
www.boost.org/libs/python/doc/ [2014-09-04]

[15] *The CImg Library.*
http://cimg.sourceforge.net/ [2014-09-04]

[16] Industrial Light & Magic
OpenEXR.
http://www.openexr.com [2014-09-04]

[17] Industrial Light & Magic
Using exrdisplay.
http://www.openexr.com/using.html [2014-09-04]

[18] KitWare
CMake.
http://www.cmake.org/ [2014-09-04]

[19] *Doxygen.*
http://www.doxygen.org/ [2014-09-04]

[20] Kyle Lutz
Boost Compute.
http://kylelutz.github.io/compute/ [2014-09-04]

[21] NVIDIA
Thrust.
http://docs.nvidia.com/cuda/thrust/ [2014-09-04]

Appendix A: Test algorithms

Algorithm 1 Linear interpolation

for all pixels p **do**
  out[p.idx] = $(1 - \alpha)$ inA[p.idx] + α inB[p.idx]
end for

Algorithm 2 Box blur

for all pixels p **do**
  value = 0
  sum = 0
  for neighboring pixels p_i **do**
    value += p_i
    sum += 1
  end for
  out[p.idx] = value / sum;
end for

Algorithm 3 Gaussian blur

for all pixels p **do**
  value = 0
  totalWeight = 0
  for neighboring pixels p_i **do**
    weight = $exp(-\frac{(p.x-p_i.x)^2+(p.y-p_i.y)^2}{\Delta^2})$
    value += weight * p_i
    totalWeight += weight
  end for
  out[p.idx] = value / totalWeight;
end for

Algorithm 4 Separable blur

for all pixels p **do**
  value = 0
  totalWeight = 0
  for neighboring horizontal pixels p_i **do**
    weight = $exp(-\frac{(p.x-p_i.x)^2}{\Delta^2})$
    value += weight * p_i
    totalWeight += weight
  end for
  tmp[p.idx] = value / totalWeight;
end for
for all pixels p in tmp **do**
  value = 0
  totalWeight = 0
  for neighboring vertical pixels p_i **do**
    weight = $exp(-\frac{(p.y-p_i.y)^2}{\Delta^2})$
    value += weight * p_i
    totalWeight += weight
  end for
  out[p.idx] = value / totalWeight;
end for

Appendix B: Output of verbose command-line

```
\$> gpuip -f gaussblur_opencl.ip --nogui --verbose
Created elements from settings. 68.32 ms
Building kernels [['gaussian_blur']]. 0.80 ms
Importing data from /home/per/dev/gpuip/examples/
   images/bridge.exr to buffer1
Importing data done. 18.22 ms
Allocating done. 2.65 ms
Transfering data to GPU done. 2.37 ms
Processing done. 1.44 ms
Exporting data from buffer2 to /home/per/dev/gpuip/
   examples/output_images/gaussblur_opencl.exr
Exporting data done. 20.18 ms

All steps done. Total runtime: 114.25 ms
```

Code 10: Example command-line application with verbose option enabled