Assessment of OpenCL software development for GPGPU

Jonas Hansson & Niklas Eklund
dt05hj4@student.lth.se, e03ne@student.lth.se

Department of Electrical and Information Technology
Lund University

Advisor: Sebastian Sjöberg

February 17, 2011

Printed in Sweden
E-huset, Lund, 2011

Abstract

OpenCL is an open and relatively new standard for programming heterogeneous systems. It was created to answer the increasing demand for a way to program a variety of devices capable of parallel execution, such as GPUs or CPUs, through one common language. Until the writing of this thesis, research on OpenCL has mainly been focused on scientific computing and thus it's desired to investigate the standard's capabilities in consumer software development.

This thesis will investigate the process of implementing computational acceleration, through the use of graphics hardware and OpenCL, into existing applications. It will describe how performance is affected and what graphics hardware is suitable to use. The assessment is done by implementing two different algorithms in OpenCL, which will be integrated into the existing software. During the development process, experiences of importance will be noted and summarized in a discussions section.

Jonas developed the GStreamer plug-in wrapper and did research for sections 2.1 and 4.1. Niklas developed the OpenSSL engine wrapper and did research for sections 2.3 and 3.1. All other parts were developed and researched together.

The authors of this thesis would like to thank Felix Hall and Sara Brodin for proof-reading and valuable feedback on this report. We also thank our supervisor Sebastian Sjöberg at Axis for inspiration and support during the thesis work.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Purpose and Motivation

Today's development of microprocessors in stationary and mobile computers is increasingly moving towards the use of multi-core CPUs as the theoretically boundaries for transistor clock frequencies are closer to be reached. This will also lead to an increased focus on parallelization of software to be able to use the multi-core systems as effectively as possible.

Next to the development of ordinary CPUs, the graphical processing units (GPU) built into most stationary and mobile devices have become more advanced. The industry has now reached a point where general-purpose computation on graphics processing units (GPGPU) is highly desirable to meet the increasing demands from software vendors and end-users.

A relatively new and open standard called OpenCL (Open Computing Language) could be the answer to this demand by opening up the vast computing power provided by the GPU and other more or less application specific processing units, to be used for general purpose computational-intensive tasks.

Due to the fact that this standard is quite new, there isn't much documented work on implementing non-scientific applications that utilize hardware acceleration via OpenCL. Instead most documented work on both OpenCL and GPGPU focuses on applications within computational science. If software and computer system vendors are going to start using GPU-accelerated computations in their products on a wider scale, work must be done in proving and documenting the gains and challenges that comes with integrating GPU-acceleration into the product development process.

1.2 Thesis goals

This thesis will analyze the process of accelerating computational-intensive tasks in existing software by integrating OpenCL implementations of the tasks into the software. First, comprehensive studies of how an OpenCL implementation can be constructed are performed. This is followed by studies on how to port the task's algorithm into the OpenCL programming language in a parallelized manner. Later on, a standalone test-application will be created so that tests can

be done on the OpenCL implementation to see if it produces correct results and also analyze its stability and debugging capabilities. This will be followed by an integration of the OpenCL implementation into the existing software.

When the OpenCL-accelerated software is considered working and stable, several performance tests will be conducted mainly on the execution time, and compare it to an implementation without the GPU-acceleration. During the development, discovered gains and challenges connected to OpenCL will be documented.

1.3 Structure of this report

In chapter 2, background knowledge is presented. This is needed by the reader to understand concepts regarding the whole report.

To reassemble real-world software development two applications will be developed. The first to look at a first encounter with OpenCL and the second to look at development with some previous acquaintance with OpenCL. The two applications are described in chapter 3 and 4, respectively. Those chapters are both structured in the same way. First providing application specific background knowledge, then describing the implementation phase and finally presenting results. Results will contain performance measures made on the standalone implementations and on the final integrated applications. Results will also describe whether the implementations was considered to produce correct calculations.

Findings and experiences from both applications regarding OpenCL software development is documented in chapter 5. Starting with discussion about individual subjects and ending with the conclusions of this report. In chapter 6 a summarize of the report is given together with possible future work.

1.4 Stakeholder of this thesis

The work in this thesis is done in collaboration with *Axis Communications AB*, hereby called Axis. Development is carried out on Axis's headquarter office in Lund, Sweden, with provided resources and guidance.

Axis develops professional technological communication products and are currently focusing on networked surveillance cameras and other networking video solutions.

The department under which the thesis is administrated, develops video hosting services collecting video from deployed cameras.

Chapter 2
Background

2.1 GPGPU computing

For over thirty years, the evolution of CPUs has followed Moore's law from 1965, stating that the number of transistors placed on a single chip would double every second year. This has for a long time been very rewarding to the industry of modern computer software and scientific computing, enabling software developers to easily create more and more complex software which has been the key to the explosive growth of the entire IT-based society. But since a couple of years back, the almost exponential growth of transistors in CPUs has decreased dramatically as the infamous *frequency wall* was hit, forcing the industry's chip manufacturers to take on a new course when developing processors. The most popular solution today to maintain the evolution of powerful computing devices, is to create multi-core chips, consisting of several cores or processing units which together with software written for parallel execution will perform better and better.

Despite that virtually all of today's CPU-manufacturers are moving towards chips with more and more processor cores, the number of cores is very small compared to what's inside modern graphics cards. This is because cores in CPUs are much more complex and capable of many instruction sets. The evolution of graphics processing units (GPU) has for many years been focusing on an architecture consisting of up to hundreds of processing units which in turn are highly optimized towards numerical-intense computations. This to be able to perform the parallelized and computational-intensive graphics tasks they were designed for.

Thanks to the parallel nature of the GPUs, the industry has increasingly shown interest in using GPUs for *General-Purpose computation on Graphics Processing Units* (GPGPU) to assist the CPU in numerically computational-intensive tasks. This has for a long time been difficult since GPUs usually had to be programmed with graphics-specific *application program interfaces*, APIs, based on standard function calls (OpenGL, DirectX etc.).

Since 2007, the use of GPUs as general purpose processing units has increased dramatically because of Nvidia's pioneering technology for programming GPUs, called CUDA. Thanks to CUDA, it's been shown to be possible to harvest the enormous numerical computational performance of the GPUs and graphics cards are now being used widely in fields such as computational science, cryptographic

analysis and large-scale simulation systems.

2.2 The OpenCL standard

2.2.1 Programming heterogeneous systems

With the introduction of CUDA in February 2007, the software industry suddenly found several areas of usage for the GPGPU-technology [24]. Around 2008, ATI released their Stream SDK through which ATI GPUs can be programmed in ATI's own language called brook+ [1]. Since the release of the Playstation 3 console, the included Cell/B.E processor has also gained popularity thanks to the high performance delivered by its 8 *Synergistic Processing Elements* (SPE).

All those three processing platforms required the programmer to use an API written by their respective vendor. Resulting in the need to learn a new language for each platform that you wanted to implement your software for. This problematic diversity in ways to program multi-core platforms called for a industry-wide standard that could be adopted by all the major vendors.

The need to standardize a way to platform-independently program multi-core architectures was fulfilled by the Khronos group [21] which in December 2008 ratified the *Open Computing Language* standard, OpenCL. This standard was originally purposed to the Khronos board by Apple Computer inc [3]. Since the Khronos ratification board consists of almost all major processor vendors in the industry the standard was quickly adopted. In 2009 Nvidia and ATI released versions of their respective GPU SDK, both including support for OpenCL [2, 16]. Later on, IBM released an implementation for the Cell/B.E architecture and Apple integrated OpenCL support in their Mac OS X Snow Leopard release [7].

Thanks to the platform independence of OpenCL, it's said to be a language for programming *heterogeneous systems*. This means that a computer system consisting of e.g. one or many CPU and GPU device(s) can now be programmed and executed as one homogeneous system in a shared software environment.

2.2.2 Concepts of OpenCL

This section will list the most important aspects that an OpenCL-developer should be familiar with. For a comprehensive manual in the OpenCL API and language, consult the official specification maintained by the Khronos group [21].

The main structure of OpenCL is divided into 4 layers or models which according to the standard are called *Platform model*, *Memory model*, *Execution model* and *Programming model* [21, 7, 24].

The following sections will describe the platform model, execution model and memory model, since those are the most important for understanding the relationship between the host and the device, and for transferring memory objects between them in a correct fashion.

Platform model

The OpenCL platform is an abstraction of the controlling software, called *host*, and the computing hardware, called *device*, the interconnection between those when it comes to memory buffers and how the host controls execution on the device. The tasks of the host are to set up necessary memory buffers and to manage the compilation and execution of the *kernels*. Kernels are small programs written in the *OpenCL C*-language. The kernels describe the work done by the parallel tasks running on the compute device. The execution order of the kernels is managed on the host via a *command queue*. The host is typically a program running on the controlling CPU and operating system of the computer which the device is located in.

The device or *compute device* is the hardware on which the kernels will perform their computational operations when they're queried for execution by the host. The host can also query commands for writing to or reading from the memory that is physically connected to the device. OpenCL devices typically correspond to a GPU, a multi-core CPU and other processors such as DSPs and the Cell/B.E. processor [21].

Execution model

The focus in OpenCL's execution model is how kernels are executed, this is managed by the command queue inside the host. When the host is executing the computational part of the program, it queues several instances of a kernel, called *work-items* and then executes them in the order they where enqueued. Each work-item has an ID in the index space which is created when the kernel is enqueued. Work-items can also be organized into work-groups, in which each work-item has a local ID. All in all every work-item has a global ID which uniquely identifies it in the global index space, and a local ID which identifies it inside the work-group it belongs to. The index space can be defined in one to three dimensions, depending of the what kind of problem is to be processed.

Memory model

There are four distinct memory areas which a OpenCL kernel can access:

Global memory This is the device's physical memory which all work-items can access and read/write. The access operations can be cached and this is the slowest memory area on the device.

Constant memory This is a physical memory area which all work-items can access in a read-only fashion. This memory is initialized by the host and can utilize very fast constant memory caches on the device if those are supported.

Local memory This area is local to work-groups and can be accessed for read/write by all work-items inside the same work-group.

Private memory A memory area only accessible by the work-item that owns it, this is usually registers close to the *arithmetic logic units* (ALUs).

It's important to be aware of the separation between how the host and the work-items on the device can access those memory areas. The host can only access global and constant memory in a dynamic read/write way, the local and private memory is only used by the work-items. When developing OpenCL-applications, the choice of which memory area the work-items should use could be critical as it can make a significant impact on overall performance. This due to the fact that the access times differ so much between the memory areas. Computational results also depend heavily on a correctly programmed memory management, because of the parallel nature of OpenCL. The software developer must ensure that the work-items read and write the correct segment at the right time without interference when executing in parallel.

Figure 2.1 shows a conceptual overview of how an OpenCL device is defined. It shows that a device has a number of *Compute Units* where each compute unit consists of a number of *Processing Elements* (denoted PE) each with its designated private memory space. The number of compute units and processing elements is device specific and those two terms are described more in detail in section 2.3.

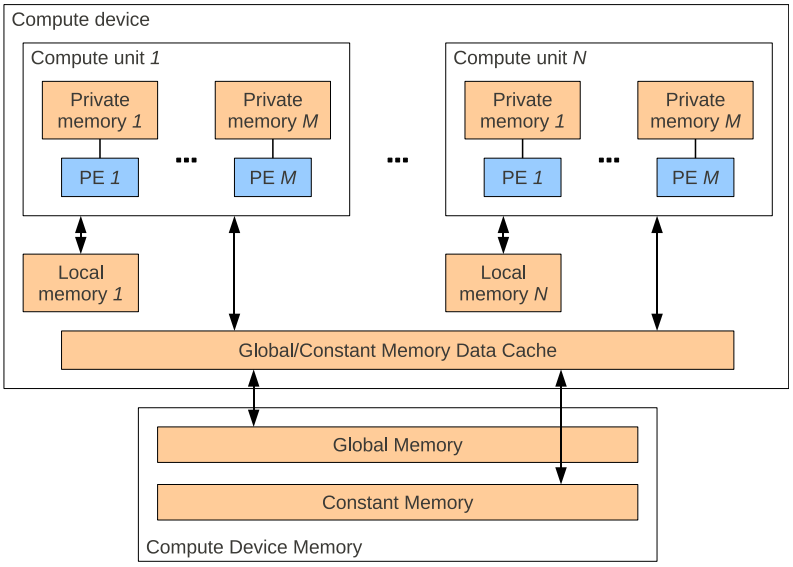

Figure 2.1: Conceptual OpenCL device architecture

Hardware and environment independence

One of the key features that was the base for developing OpenCL is that the standard has to be completely compatible with all devices that are capable of conforming to the standard. It also has to be independent of what operating system and what platform the OpenCL programs are executed on.

This means that if one writes a program that works on one OpenCL compatible platform, the same code should execute with a correct result without any modifications on any other platform supporting OpenCL. The software must be written according to the official OpenCL specification for this to apply.

2.2.3 Developing software in OpenCL

The OpenCL software framework

The framework is a collection of tools needed to compile and execute code written in OpenCL. It consists of two parts called the *OpenCL compiler* and the *OpenCL runtime library*.

The OpenCL compiler is the tool needed by the programmer to compile code written in the OpenCL C language and which describes the functionality of the kernels that are to be executed on the device. For every device hardware that supports OpenCL, the vendor must provide a compiler for each OS environment that OpenCL should support. The compiler will compile the kernel source code into binary code understood by the device hardware.

When the programmer is writing the host program, the program includes the collection of header files that defines the OpenCL API specification. It's then up to each device vendor to create a runtime library which implements this API so that the host will explicitly support the device hardware it's supposed to control.

Once the developer has acquired the correct compiler and runtime library for the device, the host can be compiled with a standard C/C++ compiler and then be executed as a standalone application or as a part of a larger application. At execution, the host program will invoke the OpenCL JIT-compiler, *Just In Time*, which compiles the kernel source code and finally sends instructions to the device according to the compiled kernel binary.

The OpenCL C language

OpenCL C is based on standard C99 and reminds much of C. Some of the more important differences are that there is added some new types but also removed some of the basic functionality found in standard C. Due to the fact that kernels written in OpenCL C are to be executed on devices only supporting basic instruction sets, the OpenCL C language must also be limited so that the programmer doesn't generate instructions not supported by the device. On the other hand, this enables the utilization of device built-in instructions such as vector computations and matrix-arranged memory spaces for very efficient handling of complex data collections.

One of the more important features introduced by the language is vector data types. Vector types are basically predefined vectors on which you can perform operations on all elements in one clock cycle, a so called SIMD (Single instruction multiple data) instruction. The vector types are named by the type of its ingoing elements followed by the number of elements in the vector, e.g. `float2`, `short4`, `uchar8`. Individual elements of the vector can easily be accessed by appending a special index annotation at the end of the vector's name. There are several ways

of annotating the index where one example is using one of the letters w, x, y, z which correspond to element 1, 2, 3, 4 respectively. A certain element can be accessed by writing e.g. `vector_name.x` to access the 2nd element.

A full description of the OpenCL C language can be found in chapter 6 of the OpenCL specification [21].

2.2.4 Benefits and limitations of OpenCL

Benefits and performance of OpenCL

The OpenCL C language introduced by the standard is completely based on standard C with some modifications. Due to this fact, programmers familiar with C can easily get started in developing kernels in OpenCL C and also easily port existing software to this language.

OpenCL is also considered to be "close to the metal" which means that its abstraction layer lies close to the native code that is executed on the hardware. Its structure when it comes to e.g. memory access and kernel execution also mimics the architecture that most of all GPUs follow which helps in harvesting as much performance from these devices as possible.

The programmer can also optimize an OpenCL-written program heavily for the certain hardware to execute against. This can be done by studying vendor documentation for the selected hardware and use this knowledge when writing the host and kernel code.

However it may seem that the optimization techniques go against the device independence of OpenCL, but the OpenCL standard is actually designed to prioritize performance before portability. The portability will always be present though, since one can write a base implementation of a certain program in OpenCL which then will execute on all compatible devices. As a next step one can optimize the code against each device which will require minimum work compared to the task of completely rewriting the software for each device [24].

Finally, Nvidia was one of the founding contributors to the OpenCL standard and its syntax reminds largely of their proprietary SDK CUDA, both when it comes to using the API calls and writing kernels. Programmers experienced with CUDA should therefore be able to move to OpenCL development only with a small learning threshold.

Limitations of OpenCL

The fact that the OpenCL standard itself is platform and operating system independent can however not be guaranteed in all individual cases. It's always up to the vendors of the hardware devices to choose which platforms and operating systems that their runtime library should support.

Thinking that all OpenCL API implementations can use both the external device and the host CPU as compute devices is wrong. Nvidia's OpenCL runtime for example does not support a CPU, only their GPUs. ATI's runtime in the meantime does support a x86 CPU as compute device, mainly because ATI is a trademark owned by AMD which also produces CPU chips.

2.3 ATI Stream SDK and Cypress GPU architecture

2.3.1 OpenCL API implementation

As the OpenCL standard only specifies an interface with no supplied code, the software layer from the interface down to the device must be implemented by the device vendor. This implementation includes all the API functions, a kernel code compiler for the specific device and often a device driver for the operating system. ATI has a software development kit, SDK, called *Stream SDK* for their GPU devices. This includes an OpenCL API implementation and kernel code compilers for both x86 CPUs and ATI GPUs. The device drivers are available separately in a software package called *Catalyst*.

With Stream SDK and Catalyst installed on the system, OpenCL applications can be developed and executed. When the host program is compiled it's including the header files supplied from Khronos and linked towards ATI's API function library. As the host program is run, the kernel code is loaded and JIT compiled before being executed on the device. The kernel code is usually in a readable file with .cl extension.

In ATI's implementation the runtime is built on top of available software layers, as seen in figure 2.2. If using a CPU as compute device, the application's API function calls only traverse the OpenCL runtime. When using a GPU, the latencies of the API function calls are longer as there are more layers to traverse. The *Compute Abstraction Layer*, CAL, is ATI's proprietary programming language and platform used exclusively before OpenCL to program their GPUs.

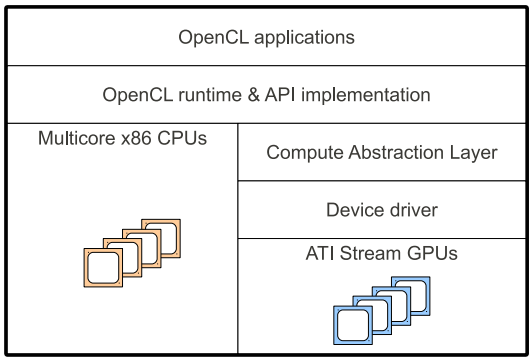

Figure 2.2: Layers of communication between OpenCL application and device.

The runtime includes a *low level virtual machine*, LLVM, where the kernel source code is first compiled into a LLVM intermediary language. For CPUs, the LLVM x86 compiler is used to produce assembler code for x86. The LLVM also has a

compiler that produces CAL code, which is in turn compiled by CAL to create assembler code for the GPUs.

The Stream SDK has received updates about once every three months. Since the OpenCL specification is quite new, the Stream SDK hasn't all the OpenCL functionality just yet. And there are optimizations to be done by ATI on the available SDK.

2.3.2 GPU architecture

The GPU device used in this thesis is the ATI Radeon HD5870, codenamed Cypress. It belongs to the Evergreen family of GPUs developed by ATI, using an architecture called Terascale. Understanding the basics of this hardware architecture is needed to comprehend how the OpenCL model concept maps to hardware entities and what software design decisions to make.

The GPU has graphic processing specific parts, e.g. texture units and video decoder, but it's the SIMD units that are used by OpenCL for execution. These SIMD units are called *compute unit* by both the OpenCL specification and the GPU architecture. The compute units are made up by several, by ATI called, *Stream cores*, mapping to the processing element of OpenCL. Each Stream core is what's executing one instance of the kernel.

The Stream cores each have five ALUs able of operating on different data. ATI calls these ALUs *processing elements*, they're however not mapping to the processing elements of OpenCL. With this design each Stream core can execute a *very long instruction word*, VLIW, every clock cycle doing work in all five ALUs. In the ATI GPU used in this thesis, the compute units have 16 Stream cores seen in figure 2.3 and there are 20 compute units in all.

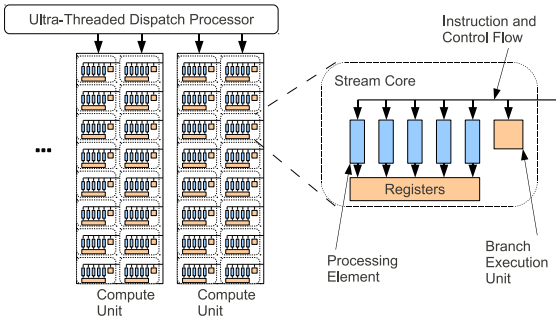

Figure 2.3: Simplified block diagram over ATI Stream architecture.

Having many Stream cores executing the same instruction is the reason for the use of *wavefronts*. A wavefront is a group of identical instructions to be dispatched for execution. ATI's design has a wavefront size of 64, meaning that the smallest

workload is 64 independent identical threads. Each wavefront is dispatched to a single compute unit, which in our GPU has 16 Stream cores. For every clock cycle a quarter wavefront is executed, followed by the next quarter for a total of four consecutive clock cycles.

The output from a Stream core is available at the input after eight clock cycles due to ALU pipeline latencies. The architecture always dispatches two wavefronts in pairs, giving eight quarter wavefronts to the compute units. This hides all ALU latencies, and the first quarter of the first wavefront can execute its next instruction with the previous output.

Memory latencies are hidden by having many active wavefront pairs, and doing a context switch to next pair when waiting for memory data. In contrast to a CPU, where a context switch is made in software and takes hundreds of clock cycles, the ATI architecture does context switches in hardware taking only one clock cycle.

2.4 Used hardware and software

The system used for development and testing is made up of the following component versions.

ATI equipped test computer

All tests were carried out on this computer if nothing else is mentioned.

Intel i7 950 A 64-bit CPU with four cores using hyper-threading for eight virtual cores at OS level. The clock frequency is 3.07 GHz.

ATI Radeon HD5870 Commercial grade high-end GPU with a clock frequency of 850 MHz and 1 GB of memory. Two of these are included in the system.

ATI Stream SDK 2.3 Software package including OpenCL support and tools for developing for ATI graphics cards.

Catalyst 10.12 Device drivers from ATI.

Nvidia equipped test computer

Intel Core 2 Duo E8500 A 64-bit CPU with two cores and a clock frequency of 3.16 GHz.

Nvidia Geforce 9400GT Commercial grade middle-end GPU with a clock frequency of 550 MHz and 512 Mb of memory.

Nvidia CUDA toolkit 3.2.9 Software package including OpenCL tools and libraries for developing on Nvidia GPUs.

Nvidia development drivers 260.19.12 Device drivers for Nvidia GPUs adopted for GPU development.

Used software

The following software were installed on both computers

Ubuntu 10.04.1 64-bit Linux operating system that was kept updated.

GNU C Compiler (GCC) 4.4.3 Common Unix-based compiler for standard C.

OpenCL 1.1 The latest version of the specification made available by Khronos.

OpenSSL 0.9.8k Toolkit implementing SSL/TLS protocols.

GStreamer 0.10.28 Pipeline-based multimedia framework made up from media-handling components.

Chapter 3

Application 1: The OpenSSL engine

The initial interest of this thesis was in accelerating decryption of data streams. In cryptography, an algorithm doing decryption and encryption is called *cipher*. The ciphers commonly used for data stream encryption consists of many steps of arithmetic operations. If the cipher is implemented solely in software it generates a relatively big workload for the CPU, the ciphers are calculation-intensive in respect to the processed amount of data. Depending on the data size being decrypted, the CPU core doing decryption can be occupied with this task for a considerable amount of time, halting other tasks. Since the same instructions are executed on all cipher blocks, it should be possible to decrypt the blocks in parallel. If the workload is put on an external device, e.g. a GPU, it could free the CPU for other tasks and decrypt the workload faster. Therefore the task of block cipher decryption may be suitable for OpenCL.

Axis had proposed the use of OpenCL in *Axis Video Hosting System*, AVHS. This is a cloud service developed by Axis for hosting many cameras and the video captured by them. The encrypted video streams from individual network cameras are decrypted by AVHS before being stored on the hard drive. The SSL/TLS protocols are used for decryption and communication between cameras and AVHS. The open source toolkit OpenSSL is the implementation of SSL/TLS used by AVHS. OpenSSL has an API for using hardware accelerators, where the hardware is loaded as an *engine*. If the OpenCL host is wrapped in an engine, it can be utilized by OpenSSL.

All performance tests of the resulting implementations will be made on the ATI-equipped test computer, but the applications will also be executed on the Nvidia equipped computer. This was to investigate whether the application runs on both platforms with a correct result without modifications. A debugging session was also carried out to investigate the debugging features of both ATI's and Nvidia's API implementations. No performance measures were made on the Nvidia GPU since the one used was a middle-end model whose performance cannot be compared to the used ATI GPU. Detailed info about the used Nvidia GPU and SDK can be found in section 2.4.

3.1 Background on OpenSSL

OpenSSL is an open source toolkit implementing the SSL/TLS protocols [17]. These protocols define how to set up a secure communication channel and how the data shall be encrypted [11, 13]. It's licensed under an Apache-styled license that makes it free for use in both commercial and non-commercial purposes. This toolkit is supplied with most Linux distributions and is therefore widely used.

OpenSSL has a structure that is presented in 3.1. The internal function calls of OpenSSL are hidden from external applications using the toolkit. The internal functions and most common cryptographic operations are implemented in the *EVP Crypto library*. The complex structure of OpenSSL makes it suitable for many systems but also abstruse, i.e. the inner workings are incomprehensible to the novice user.

There is an API defined for writing and loading *engines* [18], and the main use of this API is to make OpenSSL use accelerating hardware for some part of SSL/TLS. Implementations of the engine API communicates with various types of cryptographic hardware and communicates with the application via the EVP library. The engines are written so that OpenSSL is made known of the hardware capabilities, and so that the data operated on can be passed to and from the accelerating hardware. When a particular engine is to be used, it's either chosen with an option on the command line or as function calls in source code. OpenSSL then loads the engine, which includes probing for availability of it and initializing it. If OpenSSL is to do something that the loaded engine has capabilities for, the engine takes over that workload and does the task.

The toolkit includes some applications that make use of the OpenSSL implementation. One is the program *speed* that measures how fast a cipher or hash is calculated. It can test a wide variety of ciphers with different key sizes. An other program is *enc* that encrypts or decrypts data. It works on an input file and writes to an output file. The command line options define what cipher to use and if an engine is to be loaded.

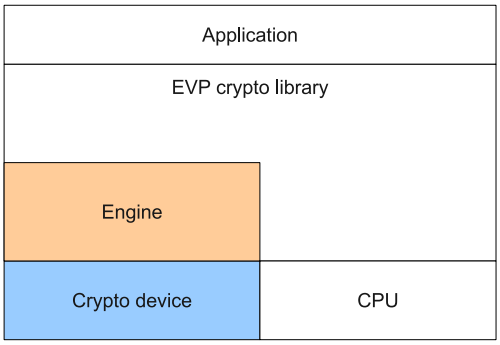

Figure 3.1: Layers of the OpenSSL architecture

3.2 The AES cipher algorithm

The algorithm used for encryption in this first application is the *Advanced Encryption Standard*. The standard is the result of a process [4] where different contributors competed with their designs, and the winner was announced as AES. The design deemed most suitable was the *Rijndael* [5] cipher, which is a symmetric-key encryption. For the AES, the Rijndael was reduced to three block ciphers. All three ciphers has a 128-bit block size with different key sizes of 128, 192 and 256 bits, respectively. They're called AES-128, AES-192 and AES-256.

Plain-text that is to be encrypted is divided into blocks of 128 bits, the message is padded with extra bits if needed to be evenly divisible with 128. Each block goes through the same algorithm and the data changes along the process. The 16 bytes making up the block are arranged in a 4x4 matrix called the *state*. There are four rows and four columns in the state, indexed from 0 to 3.

The key is expanded into *round keys* using the Rijndael key schedule [23]. The cryptologic strength and methodology of the cipher is outside the scope of this thesis, the interested reader can read more in the AES specification [15]. However, the individual steps and operations of the algorithm will be described so that the application implementation can be understood.

There are four operations in the cipher that are done on the input data. These operations are done sequentially in what is called a *round*. The algorithm does many rounds, the specific number depends on the key length. To be noted is that the first and last round differs in that they leave out some operations. Understanding what is done in the different steps is essential for implementing the algorithm. The four operations are *AddRoundKey*, *SubBytes*, *ShiftRows* and *MixColumns* depicted in figures 3.2, 3.3, 3.4 and 3.5 respectively.

AddRoundKey In this operation the round key is added to the state with bitwise XOR. This way the secret key is affecting the encryption process and different keys will generate different cipher-texts.

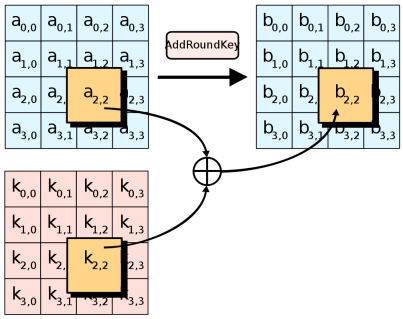

Figure 3.2: AddRoundKey, XOR between each byte of the state and the round key.

SubBytes Each byte in the state is substituted for a new value with the help of a lookup table. The lookup table is known as an *S-BOX*, holding all 256 possible byte values in a predetermined order. This operation adds non-linearity to the cipher.

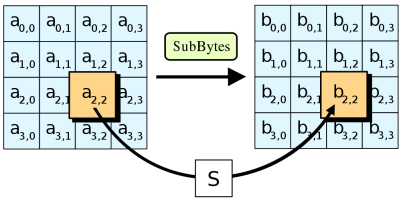

Figure 3.3: SubBytes, byte replacement from 8-bit lookup table.

ShiftRows The rows of the state is cyclically shifted to the left, the number of steps is decided by the index number of the row. The top row has index 0 and is therefore not shifted at all. The bottom row has index 3 and is shifted three steps to the left. This operation together with MixColumns gives diffusion to the cipher.

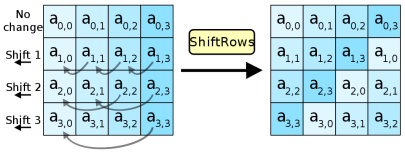

Figure 3.4: ShiftRows, cyclically rotating the rows.

MixColumns The four bytes of each column are combined in an invertible linear transform. The column is multiplied with a known 4x4 matrix so that all input bytes affects all output bytes of the resulting column. The multiplication is a special type described in the Rijndael mix columns operation [5].

The initial round only uses the AddRoundKey operation. Here the first round key is used, which also is the original key. The following rounds do SubBytes, ShiftRows, MixColumns and AddRoundKey. The round key used for AddRoundKey is the round key that corresponds to the round number. The last round has no MixColumns operation. When using a key size of 128 bits, AES-128, there are 11 rounds counting the initial round.

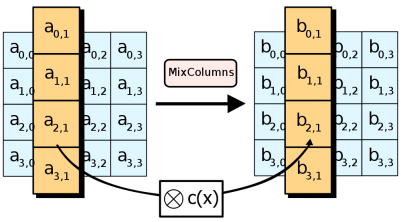

Figure 3.5: MixColumns, each column is multiplied with a fixed polynomial.

1. Initial round
   (a) AddRoundKey
2. Main round (x9)
   (a) SubBytes
   (b) ShiftRows
   (c) MixColumns
   (d) AddRoundKey
3. Final round
   (a) SubBytes
   (b) ShiftRows
   (c) AddRoundKey

The decoding of the message uses the same four operations but in reverse order. The S-BOX used for lookup is also reversed. In ShiftRows the shift is to the right and the matrix used for multiplication in MixColumns is inverted.

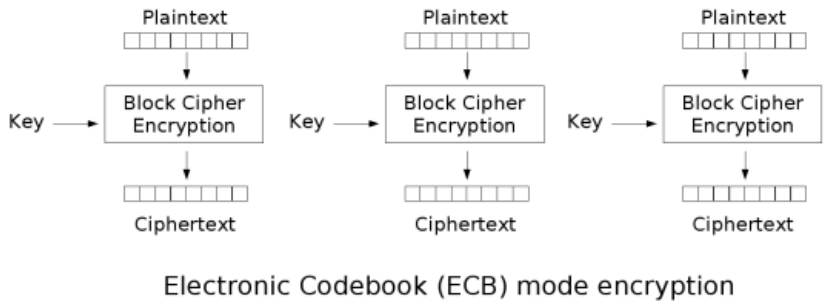

Figure 3.6: Block chart over ECB encryption.

A block cipher can be used in different *modes* [14]. Different modes differs in what is done outside the main algorithm, usually some XOR on the input state or output state. The simplest mode is electronic codebook, ECB, where nothing is done outside the main algorithm as seen in figure 3.6. This has a disadvantage in that all identical plain-text blocks will generate identical cipher-text blocks. An attack on the cipher can find patterns by just changing one bit in the plain-text and observe the change in cipher-text. If the whole plain-text differs little between blocks, the cipher-text will show a coarse pattern of that difference. Hence ECB is considered weak and it's not widely used.

The more common mode is cipher-block chaining, CBC, where a propagation between the blocks is used, figure 3.7. The cipher-text of the previous block is added to the plain-text of current block using XOR. Even though all plain-text blocks are identical, this will generate differing cipher-text blocks. An attacker will not see any patterns. To the first block an initialization vector, IV, is added to the plain-text with XOR. This IV is known to the receiver of the cipher-text and usually appended in plain-text to the message.

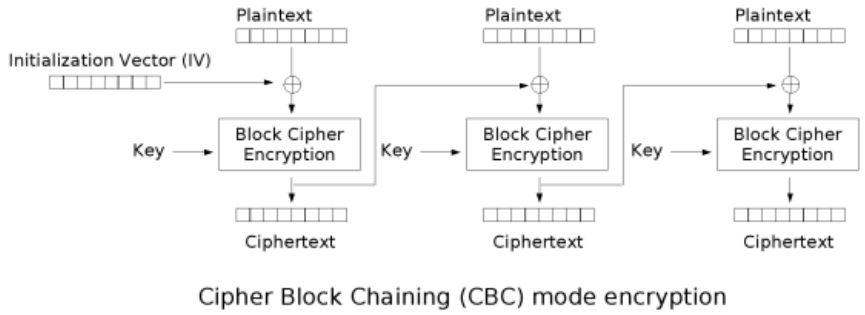

Figure 3.7: Block chart over CBC encryption.

Since the output of one block is needed for the input of the next block in CBC encryption, all blocks have to be calculated sequentially. CBC decryption and ECB encryption/decryption can calculate all blocks in parallel.

There are more modes of operation but these two modes are the ones implemented in our application.

3.3 Implementation

To evaluate OpenCL software development the finished application should be compared to a plain C application. It's therefore natural to write a version of the application in plain C as well. Since the development is to resemble real-world software development, the application should be used together with an existing application. If only used as a standalone application, many limitations can be

disregarded and the results will not compare to real-world results. The main advantage of GPU execution, compared to CPU execution, is the high amount of parallel computational power. This advantage is what is expected to deliver better performance from the OpenCL application.

3.3.1 Application specification

The application should handle AES-128 cipher support, in ECB and CBC mode for both encryption and decryption. AES-192 and AES-256, with key sizes of 192 and 256 bits will not be implemented. The input plain-text, secret key and initialization vector are provided to the application. A plain C code version of the application shall be implemented, hereby called the *reference implementation*. The OpenCL version shall try to take advantage of all possible types of parallelization. The application will first be written as a standalone executable, hereafter called *standalone*, to verify correct execution. Later, an engine for OpenSSL shall be developed, integrating the standalone application. Result data should be collected for both OpenCL and C version as standalone and as engine.

The implementation will be split up in three parts:

1. Writing standalone applications which implement the algorithm in plain C and OpenCL.
2. Writing an engine with the capabilities of AES-128 encryption and decryption using ECB and CBC, as a skeleton, loadable by OpenSSL.
3. Integrating the standalone applications into the engine skeleton to perform the work given by OpenSSL.

3.3.2 Writing the standalone

For the plain C standalone, available code from Chris Hulbert [10] was used as a starting point. Only small modifications were made to make it more compatible as an OpenCL standalone. This code takes a cipher block and operates sequentially on each byte of the block. The code has to be run as many times as there are blocks in the plain-text. This makes for no parallelization at all since the code is single-threaded and is sequentially executed on one CPU core.

The source code for the plain C implementation can be found in appendix A.1. When the plain C standalone was tested and verified for correct execution, work started on the OpenCL standalone.

The main function has to make calls to a *host program* that handles the OpenCL API calls and kernel execution. The code for the host program can be found in appendix A.2. The host program was constructed in three parts: a host set-up function, a host run function and a host tear-down function. If the standalone was to calculate a new message with the same secret key, the kernel only had to be rerun with the new input data.

To write the host program is a big part of the OpenCL software development. Many things are controlled by the host, among other things what device to use, device memory usage and kernel execution. The set-up of the host follows the pattern described here:

1. An available platform is asked for with the `clGetPlatformIDs` API call. The host program can be more or less complex. A specific platform might be selected or just the first available. Then devices are selected that are available on the chosen platform, the `clGetDeviceIDs` API call is used for this.

2. For the device a context is created with `clCreateContext`. The context holds all memory objects, command queues and kernels for a specific device. A command queue is created and associated with the context and device, `clCreateCommandQueue`. This is the queue that the host passes commands to the device through.

3. Necessary memory objects are created and filled with data using `clCreateBuffer`. These memory objects are located in the device's memory. In this application buffers for the input data, output data, S-BOX and round keys were created.

4. The API call `clCreateProgramWithSource` is used to read the kernel source code into a program object. The code is compiled for the device with `clBuildProgram`. From the compiled code a kernel object is created, `clCreateKernel`, specifying which kernel to create as the source code can contain many different kernels. The kernel function has arguments that are assigned with `clSetKernelArg`.

5. The host can also perform things that are not specific for OpenCL. E.g. the host set-up function expands the secret key to the round keys using plain C.

The main function stores the created host in memory. When a message is to be calculated, the main function calls the host run function of the host program. This function copies the input data to the memory object on the device. The work is divided into work-items that are to be executed concurrently, the API call `clEnqueueNDRangeKernel` is used to place the work-items in the command queue for execution. The host program must make sure that the right number of work-items are used so that the input data is evenly mapped onto them. When the device has calculated the output data, the result is copied from device memory to host memory. The host program then returns to the main function. If no more messages are to be calculated, the main function calls the host tear-down function. The device is released and the host related memory is freed.

With the host program constructed, work started on translating the algorithm from plain C code into OpenCL kernel code. The final kernel code can be viewed in appendix A.3 for reference in the following description of the code development. The kernel source code was written in a separate file with the .cl extension. Such a file can contain kernels and other functions used by the kernel functions. Kernel functions have the `__kernel` identifier. The code can be very similar to plain C code but OpenCL specific code can be used to take advantage of the device.

On the device mainly used in this thesis, the ATI HD5870, each processing element has five ALUs that can operate on different data in one clock cycle, as

described in 2.3. This offers a low level of parallelization if one processing element calculates more than one data element. The first approach was to have one work-item for each state element since it's an intuitive mapping. But then only one of the five ALUs would be used resulting in low utilization. Since the AES state is a 4x4 matrix it was decided that a better approach was to calculate four state elements in each processing element.

In OpenCL the hardware processing element maps to a software work-item, and one kernel instance is issued for every work-item. Therefore the kernel code needed to handle four data elements in parallel, each data element being a byte in the state block. The OpenCL specification includes different vector data types of which `uchar4` was used to store four unsigned bytes. The code was written so that each operation calculated four bytes.

In the MixColumns operation, values calculated by other work-items are read. This resulted in the use of the *local memory* area that is shared within a work-group. A barrier was needed between some operations that access other work-items results, to wait for the results to be done. OpenCL provides a built-in barrier functionality to synchronize all work-items in the same work-group. No work-item can go past the barrier before all work-items in the work-group have reached the barrier. Barriers was used after the operations where work-items need to synchronize, by adding `barrier(CLK_LOCAL_MEM_FENCE)` in the code. The local ID, or work-group ID, of the work-item was used to decide which state elements to operate on.

The four steps of an AES round will be implemented in the kernel in the following ways:

- AddRoundKey: each work-item reads a row from the state and the round key and does bitwise XOR on them.
- SubBytes: the work-item finds all four substitution values in the S-BOX and assigns them in parallel to one row of the state.
- ShiftRow: the OpenCL vector data type operation *swizzle* is used to cyclically rotate one row using the row index.
- MixColumns: one work-item calculates four bytes of the matrix multiplication. This is the most calculation intensive operation.

After the internal parallelization of the processing element was utilized, focus was moved to parallel execution of work-items. From section 2.3 it's known that the device uses wavefronts of size 64. Hence a work-group size of 64 was decided, letting each work-group calculate 16 cipher blocks in total. These 16 blocks execute in parallel and as the message size increases, more work-groups are dispatched. In this way many blocks are calculated concurrently, limited only by the number of processing elements on the device.

The OpenCL standalone was tested and verified in the same manner as the plain C standalone. Experiences from this are described in the subsection below. Performance measurements were taken from both implementations.

3.3.3 Debugging and verifying the standalone implementation

Since the goal of this thesis is to investigate the software development process with OpenCL, debugging and verification technologies on programs written with OpenCL should also be investigated.

According to ATI's OpenCL programming guide chapter 3 , experimental support for debugging through GDB is available in their OpenCL library. Note that it's considered experimental and that it's unknown how this will affect the implementation's reliability over a long time of usage. In the programming guide there is described an example session of GDB that executes an OpenCL host and also steps into the code of the kernel. For this it's required that the host program is running with the CPU as compute device and that breakpoints in the kernel are accessed by appending a special syntax onto the names of the kernel functions. All those steps was follow to carry out a simple debug session. When the kernel code was stepped into, GDB stepped through every line of the code. Some values of variables and elements of the vector types were printed out to check the functionality. Worth noticing is that no severe errors were experienced when implementing the OpenCL host and kernel, so no conclusions could be made on how well the debugging would work if there actually were complex problems with the implementation.

To verify a program's stability the popular analysis tool called *valgrind* [22] can be used, which is intended for profiling, debugging and memory leak detection. It works by setting up a virtual machine which simulates a simple computer, with its own CPU and memory areas, and then executes the program on this virtual machine. It will then report detected possible errors related to e.g. writing and/or reading memory, using arithmetic instructions and if the program has caused any memory leaks.

At first, the reference implementation of the standalone was executed through GDB and valgrind. Valgrind reported no errors and no memory leaks.

When the OpenCL-implemented standalone was executed through valgrind on the ATI device, some interesting results were observed. Valgrind reported over 1,000,000 errors related to memory access and logical operations. It also reported that the standalone caused a memory leak classified as definitely lost with a size of 7,184 bytes and a leak classified as possibly lost with a size of 916,949 bytes. The sources of those memory leaks was searched for in the source code but this was without any success.

The error count of over one million lead to the suspicion that it had something to do with compatibility issues between valgrind and OpenCL implemented programs. To find out more about this, questions was asked on the developer forums for the Stream SDK and OpenCL found on ATI's website [2]. Developers working for ATI confirmed that there were known compatibility issues between software written with the Stream SDK and verification software such as valgrind. They also claimed that those issues would hopefully be solved in upcoming versions of their SDK.

The OpenCL-implemented standalone was also executed through GDB and valgrind on the computer with the Nvidia GPU. The GDB debugging session went fine there as well, valgrind reported zero errors and a memory leak classi-

fied as definitely lost of 13,729 bytes.

3.3.4 Writing the OpenSSL engine

This was a time-consuming work as the engine API is lacking good documentation. As the main functionality of the engine is in the OpenCL host and kernel, only a simple engine skeleton porting OpenSSL to the host was needed to be constructed. Still this was no straightforward task without documentation.

In the OpenSSL toolkit there are some engines supplied. By studying their source code and parts of the OpenSSL source code, the engine structure was apprehended. The engine code should contain functions to let OpenSSL know the engine's name and capabilities. An initialization routine loads the engine and allocates the needed resources. When some work is to be done, OpenSSL calls the corresponding function and supplies the work data.

The implemented engine did no operation on the data at first. It only loaded and interacted with the OpenSSL toolkit installed on the test system. The engine was compiled as a shared library accessible by OpenSSL.

A data-set with known plain-text and cipher-text was used to verify that the engine was working correctly. The OpenSSL *enc* tool was given the plain-text, and the encrypted result was compared to the known cipher-text.

3.3.5 Integrating standalone implementation into the engine

To make the engine perform real work, the AES standalone had to be integrated into it. Since OpenSSL includes plain C versions of the AES algorithm, they were used as reference implementations. The engine skeleton had to map to the host program in some way. Deciding how to do this was fairly easy since the functions in the engine skeleton maps well to the three functions making up the host program.

When the engine is loaded and initialized the host set-up function is called. The secret key is also expanded in the engine initialization routine, and the OpenCL device to be used is selected. If the engine is to do some AES calculations it calls the host run function of the host. The data supplied by OpenSSL is forwarded to the host and the device calculates the output data. The engine clean up function maps to the host tear-down function. After that OpenSSL unloads the engine and resources are freed.

Through command line options the engine could be selected when launching an OpenSSL tool. The output was compared and verified to data known to be correct.

3.4 Results

As this was the first application written in OpenCL, an extensive study of the OpenCL specification had to be done. There are many terms and concepts that needed to be understood and used in the right way. The mapping of hardware resources to OpenCL resources took some time to figure out. This required the

learning of the OpenCL programming language, and studying of the SDK supplied by the device vendor ATI. In this regard, the learning curve of OpenCL is steep when writing a first application.

A considerable amount of time was also spent understanding the OpenSSL engine API, and this task was made hard because of poor documentation. This is however not OpenCL specific and would also apply even for writing a plain C engine. What can be said is that the OpenCL host program is flexible enough to be mapped into the engine functions.

The AES algorithm proved to be suitable for OpenCL parallelization as the cipher blocks could be handled concurrently. Each byte in the state block was calculated in parallel and many cipher blocks were dispatched at the same time, except for the encryption in CBC mode. In CBC encryption the cipher blocks were dispatched sequentially.

3.4.1 Performance results

The execution time for calculating specific messages was used to compare performance. The absolute variation in time when running the same command many times was under 3 ms, for most message sizes only affecting the third most significant digit or less. Hence four runs of the same message size were deemed sufficient, and the average value was registered. As a standalone implementation much control is available over the workload supplied to the OpenCL version. The more real-world adapted OpenSSL engine is controlled by an application that can't be managed in the same way. Therefore measurements on both standalone and engine implementation were made.

In the host program there are as mentioned three functions. The function doing the real calculation is the host run function and that is the one that should be measured. This is because the host set-up function is only called once when initializing the application, and the host tear-down function is called only when not having any more work to do.

In the main function the *timespec struct* was used to register the systems real-time clock. This struct has a resolution of 1 nanosecond. Before and after calling host run function the clock value was stored. The clock difference was calculated and written to a file. The same struct was used in the main function for measuring the plain C implementation.

For reference to the OpenCL engine, the internal AES implementation of OpenSSL was used. To supply the implementations with work, the *enc* application included in the toolkit was started from the command line. This application was given some options to decide which implementation to use and the size of the message. To time this application execution, another tool called *time* was used, which is a command line tool reporting the time taken for the specified command. The recorded value is the average of four runs.

The intended usage of the application is to decrypt incoming video streams from network cameras. They're encrypted in CBC mode since ECB is considered unsafe as described in section 3.2. CBC mode decryption is just marginally more calculation-intensive, due to the added XOR outside the main AES algorithm.

Thus the results from CBC decryption are very similar to those of ECB decryption. Because of the intended usage and similar results, only charts for CBC decryption are shown.

The first chart in figure 3.8 shows time taken for small messages by the standalone implementations. The x-axis is input message size and y-axis is the execution time decrypting the message. Here the linear time increase for plain C version can be seen, when the message size gets bigger, this comes from the sequential execution. The fact that the line for plain C starts in the axes' origin, is because of the lack of buffer copying and kernel enqueuing. Timings for the OpenCL version starts at almost 2 ms even for the smallest message size. The overhead time comes from that the API function calls in the host run function are always run no matter how small the message is. Depending on the platform and runtime used, these API function calls are taking different amount of time but always add some overhead. These function calls vary in time because of the structure described in 2.3.1, causing a deviation from a linear line, i.e. 4 kB takes longer than 8 kB for OpenCL on GPU.

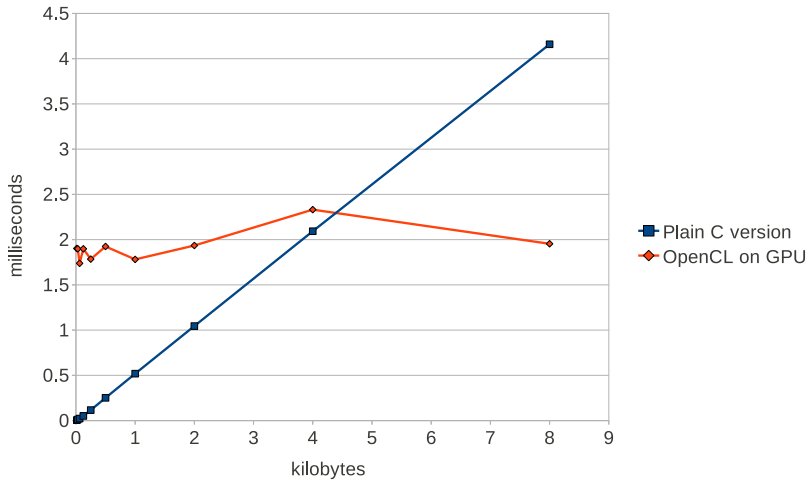

Figure 3.8: Crossover point for AES decryption in CBC mode.

At just over 4 kB there is a crossover point, meaning that the OpenCL version is faster than the plain C version for bigger messages. As the message size is increased, the overhead is proportionately less of the total time. The OpenCL version also gets almost linear time increase for big messages, although with much smaller inclination. This can be seen in figure 3.9.

As figure 3.9 also plots the OpenCL on CPU version, it can be seen that it's only marginally faster than plain C. This version uses all eight cores of the CPU while

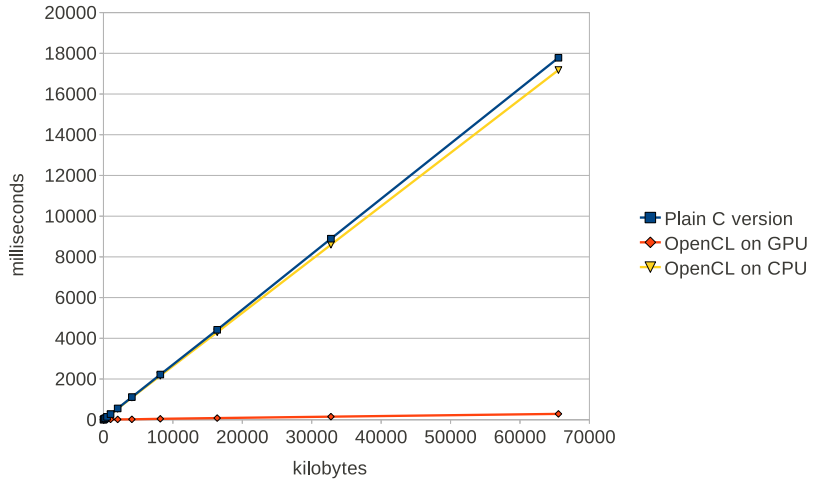

Figure 3.9: Time taken for AES decryption in CBC mode.

the plain C version only uses one core. It's somewhat baffling that eight cores take as long time as one core in performing the same amount of work. This issue will be discussed later in the report in section 5.2.

By dividing the execution times of two implementation, the speed difference is given as result. This is a sort of relative performance measurement. In figure 3.10 it's plotted how much faster OpenCL is on GPU and CPU compared to plain C on CPU. The x-axis is input message size and y-axis is how many times faster the OpenCL implementation is. At the smallest message sizes the overhead affects the relative performance, which is less than one. The speed up of using a GPU levels out at around 60, when big enough messages are being calculated. This is when all compute units of the GPU are used and work-groups have to wait for free compute units. As expected the CPU OpenCL version has a steady ratio around one, meaning it's just as fast as the plain C version.

The charts for AES encryption/decryption in ECB mode were similar to these. In CBC encryption where each cipher block is calculated sequentially, the OpenCL versions are a lot slower since the API function calls are issued between each cipher block. This adds the overhead for every new cipher block resulting in long calculation times.

When the execution time for the engine was measured the results weren't in favor of OpenCL. Figure 3.11 shows that the engines are much slower, also for big message sizes. When examining why this was the case, it was noticed how OpenSSL uses the engine. No matter how big the message size is, OpenSSL only lets the engine calculate 4 kB at a time by dividing the message in smaller fragments. From the standalone it's known that 4 kB is just under the crossover point, where the OpenCL versions are slower than plain C. Adding the overhead of calling the engine for every 4 kB and that OpenSSL's internal AES implementation is

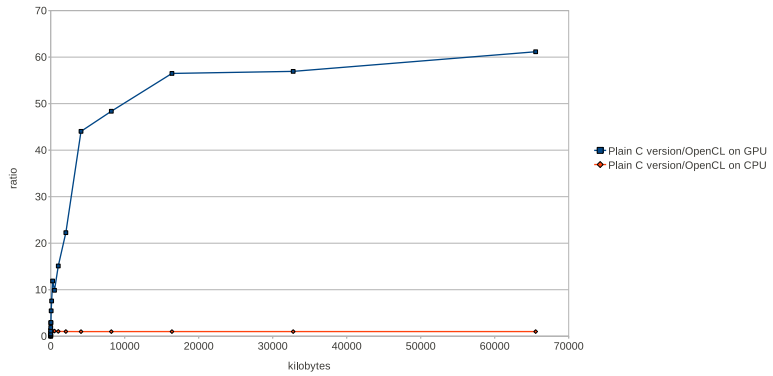

Figure 3.10: Times faster the OpenCL implementation is over plain C implementation in CBC mode AES decryption.

further optimized, the lowered performance of OpenCL is explained. This limitation in the application layer above the OpenCL host was an unexpected result.

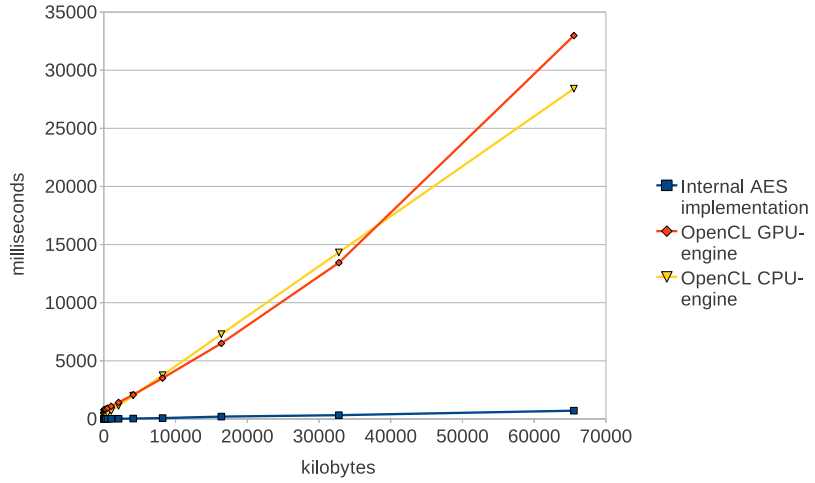

Figure 3.11: OpenSSL execution times for engine implementation and internal AES implementation for CBC mode AES decryption.

The OpenCL specification allow for distribution of the workload over multiple devices, with intent to shorten the total execution time. A quick modification of the host program was done to utilize both GPUs in the system, by letting the devices calculate half the message each. This resulted in no performance increase and again the developers at ATI were consulted, they gave the explanation to this

in the developers forum on ATI's website. The current ATI SDK and OpenCL runtime can't handle two devices in parallel, meaning that the devices do work one at a time. Due to this sequential behavior the results from using two devices do not contribute any knowledge and are left out.

The OpenCL-implemented standalone was also tested on the workstation computer equipped with a Nvidia Geforce 9400GT graphics adapter. Since Nvidia has specified that the CUDA SDK doesn't support a CPU as compute device [16], the host had to be slightly rewritten to only execute on the GPU. The computational results from executing on the Nvidia device was correct.

Chapter 4

Application 2: Barrel distortion filter plug-in

After the implementation of the OpenSSL engine was considered completely finished, work on the second application started.

For the second application it's desired to process data with a "smallest-size" big enough to hide the overhead from host set-up and kernel enqueuing, to ensure that as many benefits as possible is drawn from the OpenCL device. The application should also be of interest to Axis, and easily integrated into their existing products. For these reasons an image processing application was considered. Many image processing algorithms can be implemented in a massively parallel manner that operate on a per-pixel basis. Hence enabling us to utilize as good as all processing elements at the same time, if the image size is big enough.

After some discussions with people at Axis it was decided that a filter plug-in for the GStreamer library [9] was a suitable implementation. The choice of image processing algorithm fell on a barrel distortion filter. It's used to correct positive radial distortion effects that occur in video from cameras equipped with wide-angle lenses. The implementation of a plug-in for the GStreamer library was considered to be a realistic task since the library has a strictly modular structure, ensuring that an integration of an OpenCL host can be made with minimal changes to the original application. GStreamer is rich of example code and also well documented. Both when it comes to writing applications that uses the GStreamer library and also writing plug-ins for the framework, which would help in developing a plug-in.

4.1 Background on GStreamer

GStreamer is a framework for creating streaming media applications [9]. It's main use case is for building different kind of media players for streaming content. Thanks to its highly modular structure, virtually any kind of data can be streamed through the framework but audio and video is most common.

The main concept of GStreamer is focused around the *pipeline*, through which data flows from its source (e.g. a file or network streaming) and towards the endpoint which receives the media and presents it (e.g. show video on screen

and/or play audio). The pipeline is built up by combining *elements* that process the data in different ways. For example if the application shows a video stream from a network source, the pipeline starts with an element that receives the raw data from the stream. After that a demuxing element separates the audio and the video, passing the video on to an element that decodes the video. Optional elements which does some filtering can be used on the decoded video. Finally the stream is passed into an endpoint element, displaying the video on the screen.

Some of the most fundamental objects in GStreamer need to be explained

Element The most important object of GStreamer, the pipeline building block. Several elements are chained to build up the pipeline and every element plays its part. For example acting as a file or streaming data source, decoding audio/video or applying some kind of filter on the audio/video data.

Pad The input and output primitives of an element, used to connect different elements with each other. It can be seen as either the plug or the port of an element, through which data flows into the element or out from the element. Pads through which data flows into the element are called *sink pads* and pads through which data flows out from are called *source pads*.

Pipeline Is a so called *bin* representing a group of elements connected to each other. A pipeline bin is always a top-level bin, which also controls the playing state of the stream i.e. playing, paused, stopped etc.

Buffer Is one of many ways in which GStreamer communicates with e.g. overlaying applications and between elements in the pipeline. The buffer object is passed on between every element in the pipeline and contains a segment of data that each element process.

Plug-in Is an application object which encapsulates one element and its pads, it also contains all details that neighboring elements need to know about it. The combination of plug-ins is what builds up the core functionality of a GStreamer instance.

To use a specific plug-in there are generally two methods that can be used. The *first* is to write an application based on the GStreamer framework that sets up the entire pipeline in the source code. This would include allocating all the elements in the pipeline and defining how they should be connected. How the source of the video stream should be accessed and how the resulting output video should be presented is also defined. This implementation could e.g. be used as a standard media player application that plays audio/video over the network or from local files.

The *second* method is to simply set up the pipeline with a command line application called `gst-launch`. With this program, the user lists all the elements in order that should be used in the pipeline and gives some optional arguments to them, called *options* by GStreamer.

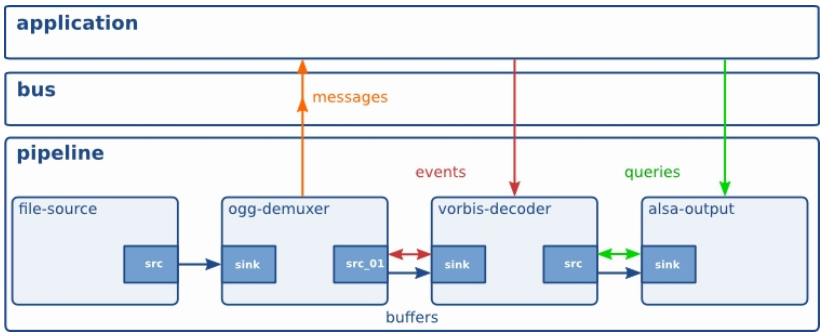

Figure 4.1: Conceptual image of a GStreamer pipeline and communication paths.

4.2 The lens distortion correction algorithm

Lens distortion is a phenomena that appears on images taken through lenses. It's known that all lenses contribute to some amount of distortion to the captured image. The added distortion can range from very small barely visible to very large, like the type of barrel distortion handled in this application. *Barrel distortion* is a kind of radial lens distortion which manifests when taking images through wide-angle lenses. It displaces the image points inwards to the center of the image, with more distortion as the distance to the center increases. This is creating an optical effect where originally vertical lines become bent, and reminds of the slits in a wooden barrel [19]. Figure 4.3 shows a barrel distorted video frame together with it's distortion filtered counterpart.

The process of correcting lens distorted images is desirable in many application fields, such as 3D measurement and video compression. Several ways to model the distortion of an image have therefore been developed, where *Brown's model* is the most popular one today. Thanks to Brown's model it can be determined where any pixel in the distorted image would be placed in the plane of a image taken without lens distortion. The model describes the relation between the pixels in a distorted image and its undistorted counterpart.

For each pixel in the distorted image, the model distributes its color in the undistorted image. Usually the undistortion is done pixel wise in the undistorted image, and the model needs to be inverted to start in the undistorted domain. The model inversion is no straight forward task and not easily implemented in software.

de Villiers et al. [6] have proved a method for using Brown's model in its original form for inverse distortion (also called *undistortion*) by using appropriate coefficients in the equation of the model. Thus Brown's equation can be used for generating undistorted images, instead of using more computational-intensive methods such as approximating the inverse of Brown's model.

Originally, Brown's model specifies infinite number of terms. It was chosen to limit it to an equation with 3 radial terms, which was proven to be sufficient according to de Villiers et al. and J. Park et al. [6, 19]. With the coordinates

in the undistorted image and radial distortion coefficients the distorted point is given. The equation derived is called the *undistortion algorithm* and can be found in figure 4.2.

$$\begin{pmatrix} x_d \\ y_d \end{pmatrix} = \begin{pmatrix} x_u \\ y_u \end{pmatrix} \left(1 + r_u^2 K_0 + r_u^4 K_1 + r_u^6 K_2\right)$$

(x_d, y_d)	=	Coordinates in distorted image
(x_u, y_u)	=	Coordinates in undistorted image
r_u	=	Distance from image center point given by: $r_u = \sqrt{(x_u - x_c)^2 + (y_u - y_c)^2}$
K_i	=	Coefficient for radial distortion parameter

Figure 4.2: The undistortion algorithm, based on Brown's model

4.3 Implementation

4.3.1 Application specifications

The final goal for the implementation of the undistortion algorithm is usage in video frame processing. The key functionality should be separated into a function that takes a pointer to a memory object containing one image frame, processes it and finally puts the resulting image in an output buffer. The implementation should initially support the RGBA (red, green, blue and alpha channel) color space, allowing straight forward manipulation of single pixels and adding up pixel values to create a new color value.

The progress will be similar to the OpenSSL engine, starting with implementation of the undistortion algorithm in a sequential manner written in plain C. This implementation will be denoted the *reference implementation*. Then follows the creation of a parallelized OpenCL-application which utilizes either the CPU or GPU as compute device. To compare the versions, the user of the resulting plug-in must be able to choose which of the implementations to use.

The implementation in plain C is called the reference implementation since it should remind of existing commercial software which doesn't draw any benefits from certain external hardware acceleration. It will be used as a starting point for comparison with the OpenCL implementation to partly investigate how much work is required to port it into OpenCL and partly to measure the performance gains from using the OpenCL implementation instead. Attempts were made to find existing plug-ins for GStreamer that could perform barrel distortion filtering but they were without success.

The implementation process will be split up in three parts:

1. Writing a standalone application which implements the undistortion algorithm in plain C and in OpenCL.
2. Writing a plug-in which acts as a wrapper for the functions that implement the undistortion algorithm and initially doesn't do any processing on the data.
3. Integrating the implemented undistortion algorithm from the standalone into the plug-in and verify its functionality.

The test setup consisted of an Axis 223 network camera equipped with a wide-angle lens, connected over 100 Mbps LAN to the test computer. The computer will put the data stream through a GStreamer pipeline, filtering and displaying the video.

4.3.2 Writing the standalone

The first task was implementing the barrel undistortion algorithm in plain C, the studies of the algorithm used is described more in detail in section 4.2. The source code for the reference implementation and OpenCL host and kernel can be found in appendix A.4, A.5 and A.6, respectively.

The lens distorts the captured image along the radial distance from the center of the image. In short, the relative movement of the pixel is calculated from a polynomial where the pixel's distance to the center of the image is the input value. Thus the formula for relative movement is

$$\begin{pmatrix} d_x \\ d_y \end{pmatrix} = \begin{pmatrix} x_u \\ y_u \end{pmatrix} \left(r_u^2 K_0 + r_u^4 K_1 + r_u^6 K_2 \right)$$

.

The input value r is normalized to work on all image resolutions. The polynomial is then multiplied with the distance to the center of image to find the relative movement distance d. This distance is then added to the current pixel's position to find the distorted point. The four pixels surrounding the distorted point are linearly filtered to find the color of the current pixel. Those operations will be performed in the following steps.

1. Get current pixel coordinates (x_u, y_u) in undistorted image.
2. Calculate distance to center (x_c, y_c) using $r = \sqrt{(x_u - x_c)^2 + (y_u - y_c)^2}$.
3. Normalize r using half of the diagonal, i.e. distance from center to corner of image using $r = \frac{r}{\sqrt{x_c^2 + y_c^2}}$.
4. Calculate the radial distortion polynomial $m = K_3 r^6 + K_2 r^4 + K_1 r^2$.
5. Get distorted location $(x_d, y_d) = (x_u, y_u) + m(x_u - x_c, y_u - y_c)$.
6. Filter linearly the four pixels surrounding (x_d, y_d) to get color of (x_u, y_u).
7. Write the color of the pixel to the output buffer.

As a starting point for the implementation the source code from Luis Alvarez et al. [12] was used, which starts in the distorted domain. This code was rewritten to be used in parallel execution later on. In the original form, one value was read and distributed to four pixel writes. This works in a sequential execution, but not in parallel execution where many writes to one pixel should occur at the same time interfering with each other. The code was changed so that values from four readings were filtered and generating one write. Many readings of one pixel can occur at the same time without undefined behavior. In the parallel execution in OpenCL, this single write is necessary so that each of the work-items can operate independently of the others.

The reference implementation of the above steps were made by putting them inside a double loop which iterated through all of the image's pixels. In each iteration the color value of the corresponding output pixel was calculated.

For the standalone implementation a simple test program was written, which loaded a bitmap file and sent its raw pixel data to the function performing the undistortion. This requires that the raw data sent to the undistortion function must be in RGBA format with 8 bits depth for each channel. Since Brown's model with three coefficients is used, those must also be supplied as the application doesn't calculate them on its own. The coefficients for the lens used were determined beforehand using a known test pattern. Figure 4.3 shows the difference between a correct image and a distorted one. The function that performs undistortion implemented in plain C is named `undistort_image_bgra` and can be found in code appendix A.4.

Figure 4.3: Top: Correct image without barrel distortion. Bottom: Image viewed through a wide-angle lens causing distortion.

Once the standalone reference implementation was finished, work moved on to the OpenCL implementation. In this case many parts could be reused from the host program developed for the OpenSSL application. The implementation of the host set-up function is briefly described in the following steps:

1. Obtain the first available platform and obtain a GPU or CPU device inside this platform.
2. Create a context connected to the chosen device.
3. Create a command queue connected to the context and the chosen device.
4. Create memory buffers for the input and output images with a size of width*height pixels and 32 bits per pixel.
5. Create a program object with the source code from the loaded kernel .cl file.
6. Create a kernel object from the program object, and set the following kernel arguments through the `clSetKernelArg` command
    - Polynomial coefficients for the algorithm.
    - Pointers to the input and output memory buffers.
    - Normalization distance from center to image corner.

When designing the function executing the image processing algorithm, the two-dimensional structure that represents an image can be used. A two-dimensional *index space* is set up where the dimensions represent the width and the height of the image. The set-up was done by setting the values of element 0 and 1 in the `global_size` vector to the corresponding width and height of the image. This index space was used inside the kernel to fetch the position of the pixel it operated on. Thus allowing the pixels to be accessed in a more intuitive way, since one location in the index space directly corresponds to a pixel in the video frame.

As mentioned earlier, the work-group sizes should always be evenly divisible by 64 which is the wavefront size when working with ATI GPUs. The local index space for each work-group should be dimensioned so that the work-groups fit evenly in the width and height of the images. Local index space was dimensioned to 32x2 by setting element 0 and 1 in the `local_size` vector. This gave a work-group size of 64 and also made it evenly divisible with the video resolutions that the resulting plug-in should support. With this knowledge, the steps for executing the algorithm on an OpenCL device can be specified

1. Set the global and local sizes of the two-dimensional index space.
2. Copy the input image data to device memory by enqueuing this action to the command queue.
3. Enqueue the number of compute kernels for execution defined by the global and local index sizes, letting this action wait for the preceding buffer write command to finish before its started.

4. Enqueue a read from the device memory into the output image buffer, letting this action wait until the preceding kernel enqueue has finished its execution.

Once the host was finished, the process of porting the reference implementation into OpenCL kernel code was easy and quick.

In the kernel code the for-loops were removed, which meant that each work-item would operate on exactly one pixel. The coordinates of the pixel is given by the location of the work-item in the two-dimensional global index space. The use of OpenCL's vector data types were introduced to easily store intermediate position values containing both x and y coefficients. Vectors were also used to represent the contributing RGBA values from the four pixels surrounding the distorted point (`float2` and `float4` respectively). This was all that was needed to implement the undistortion algorithm for parallel execution since all the work-items work on one pixel independently of each other. The function in the host that executes the kernel is named `undistort_image_bgra` in the source code.

4.3.3 Debugging and verification of the standalone implementation

The standalone implementations, both in plain C and in OpenCL, were executed through the debugging tool GDB and the program analysis tool valgrind. This was to verify the stability of the application and also whether the OpenCL implementation can be debugged and analyzed. More details on valgrind can be found in section 3.3.3.

The debugging session described in ATI's stream programming guide was followed to verify that one can step through the kernel code, this worked without any problems.

When analyzing the standalone through valgrind, it was first executed using the OpenCL implementation with CPU as compute device. Over 1 000 000 errors were reported by valgrind and a memory leak of 16 920 bytes classified as definitely lost. Those numbers are similar to the ones recorded when doing the same step in the OpenSSL application. When running the standalone using the reference implementation in plain C through valgrind, the tool reported zero errors and no memory leaks.

As stated in the OpenSSL application case it was discovered that large error amount is caused by compatibility issues between the ATI Stream SDK and valgrind as acknowledged by ATI.

4.3.4 Writing the GStreamer plug-in

The goal with the GStreamer plug-in was to make it simple to integrate the OpenCL host into it. This required that the plug-in could extract single frames of raw data from the video stream, send them to the OpenCL host for processing and finally put the processed frame back on the GStreamer pipeline. Since the standalone OpenCL application was implemented to only support RGBA color space, it would be required for the plug-in to supply the host with this format.

Fortunately, it was quite easy to start with plug-in development in GStreamer thanks to their plug-in template generator. This is a program that generates a complete source for a simple plug-in, that initially doesn't do any work, it just pushes the incoming buffer objects to the element's source pad. The function inside the plug-in that triggers the actual processing on the video stream is called `chain`. It receives a buffer object of the type `GstBuffer` together with a reference to the plug-in class itself.

It was discovered that when the plug-in receives video data in a raw (i.e. not compressed) format, each call to the `chain` function that is triggered receives a buffer object which contains exactly one video frame. This was helpful for keeping the code simple since no more preparations on the buffer had to be made before sending it to the host.

Section 4.1 describes two methods of using plug-ins in GStreamer. In this case, the method which used the command-based program `gst-launch` to set up the pipeline and view the streaming media was chosen. To test the plug-in, this was the fastest and easiest way to get started and to test different configurations of the elements. The command used to test the simple pixel-shifting plug-in was

```
gst-launch -v souphttpsrc location=
http://[ip-address]/mjpg/video.mjpg?resolution=1600x900 !
multipartdemux ! jpegdec ! ffmpegcolorspace ! cldistort ! ximagesink
```

.

As seen above, each element in the list is separated by an exclamation mark. Firstly, a source element called *souphttpsrc* which retrieves a data stream from the test camera, followed by an element which demuxes the stream into video and audio. After that, an element that decodes MJPEG data into raw video data, followed by a color space converter and then the plug-in which is named *cldistort*. The cldistort element passes the filtered image frame to the final element called ximagesink which opens up a window displaying the resulting video.

4.3.5 Integrating standalone implementation into the plug-in

After completing the first simple implementation of the plug-in, integration of the code from the standalone OpenCL application began. The strategy was to follow the procedure from corresponding steps in the OpenSSL application, i.e. mapping the three functions in the host program to parts in the plug-in. The header files that defines the OpenCL host functions were copied from the standalone and included in the plug-in source.

The host set-up must be done in a function called `gst_cldistort_set_caps`. This is a function that negotiates specifications on the video stream with the previous connected element in the GStreamer pipeline. The negotiated width and height of the stream was sent to the host set-up function.

Calls to the appropriate implementation of the distortion algorithm was inserted in the `chain` function. If the user wants to use the reference implementation, its function `undistort_image_bgra` is called. If the user wants to use the OpenCL implementation with either CPU or GPU as compute device, the host run function `host_correct_image` is called. Both implementations take a

pointer to a memory buffer in which their resulting filtered image is placed, this buffer is finally pushed onto the plug-ins source pad so that it continues along the pipeline.

Deallocation of the OpenCL objects that were created for the host instance is done by the host tear-down function `host_clean`. This function is called when the plug-in changes its state to inactive. This is handled in the plug-in function `gst_cldistort_change_state`.

To open a GStreamer instance with the resulting plug-in, almost the same `gst-launch` command as mentioned in section 4.3.4 was used. The difference is an added option to the cldistort plug-in which tells it which of the implementations to use. For example, to use the plug-in with the OpenCL implementation on a GPU device, the command was

```
gst-launch -v souphttpsrc location=
http://[ip-address]/mjpg/video.mjpg?resolution=1600x900 !
multipartdemux ! jpegdec ! ffmpegcolorspace !
cldistort mode=oclfiltgpu ! ximagesink
```

.

4.4 Results

To assess the process of integrating OpenCL implemented code in existing applications, some knowledge of coding the target-applications is required. In this case it was mainly about learning how the GStreamer framework works and how to write simple plug-ins for GStreamer. This showed to be a nontrivial task since GStreamer introduces many new concepts of how data is processed that the developer needs to learn. GStreamer is in turn based on the GLib library. GLib is a general-purpose utility library, which provides many abstractions for e.g. object orientation, threading and type conversions [8]. Learning those concepts also required some extra work. To get started with developing the plug-in, studies were carried out on the GStreamer application writing manual as well as source code from the included GStreamer plug-in package.

Thanks to the experiences gained from the OpenSSL-application, the construction of the OpenCL host went quickly and the code became simpler and easy to understand. The procedure of setting up the host when it comes to obtaining references to a platform, a device, creating the context, queue, kernel and finally memory objects was done in almost the exact same way as in the OpenSSL application. This indicates that the OpenCL learning curve is steep but very short. Once the developer understands the concept of creating the host and managing kernel objects and memory objects, both applications are created almost the same way no matter their differences.

The integration of the OpenCL host into the GStreamer plug-in was also an easy task since GStreamer uses a buffer concept similar to the one in OpenSSL engines. A processing-function in the plug-in is called from the parent functions which provides the processing-function with a buffer object or pointer which contains the data to be processed.

The first version of the plug-in with OpenCL-acceleration was produced in less than two hours of work on the integration. That work resulted in a plug-in which could produce a working distortion filtering with the original frame rate preserved. The only work that remained was to expand the plug-in functionality so that the user can choose which implementation to perform filtering with via GStreamer's plug-in option feature. Due to the lack of experience in developing GStreamer plug-ins, this part was also the one that took the longest time since studies of the plug-in writing had to be carried out.

4.4.1 Performance results

The ATI equipped test computer was used for performance measurements and debugging and verification.

The first performance measurements to investigate was the execution time of the image processing function in the standalone programs, both with the reference implementation and with the OpenCL implementation. This will be useful since those are making the most time consuming work when integrated into the GStreamer plug-in. The execution time was measured by taking timestamps right before and after the function was called and then calculate the elapsed time. Time was measured both for the host execution on its own and for the entire chain with host set-up, host kernel run and host tear-down.

A series of test images of various sizes were generated to use as in-data to decide at which in-data size the OpenCL implementation became faster than the reference implementation. It was discovered that the reference implementation was slightly faster than the OpenCL implementation for sizes up to around 20 Kb. For all larger sizes the OpenCL implementation grew almost exponentially faster than the reference implementation. A chart displaying the measured times for small sizes where the crossover point can be seen is displayed in figure 4.4 and a chart for larger image sizes is shown in figure 4.5.

When the GStreamer plug-in was considered finished with a working integrated OpenCL implementation, some measurements was made on it. Since it's about video streaming, it's not so easy to make measurements as on the standalone applications. Therefore it was chosen to measure the properties found most important in video streaming, namely frame rate and CPU load. Tests were carried out on the plug-in using both the reference implementation and the OpenCL implementation with GPU and CPU as compute device, separately.

The tests were carried out by opening up a video stream from the test camera and then measuring the frame rate and the CPU load caused by the GStreamer process. The frame rate was measured by inserting code in the plug-in that made it print out the frame rate every second. An average frame rate of over more than 15 seconds is what is presented here. The CPU load of the computer that was displaying the video was observed during the streaming, and the average was registered. Note that this is coarse value only indicating the stress on the system's CPU. In table 4.1 observations of frame rate and CPU load are presented for the highest image size.

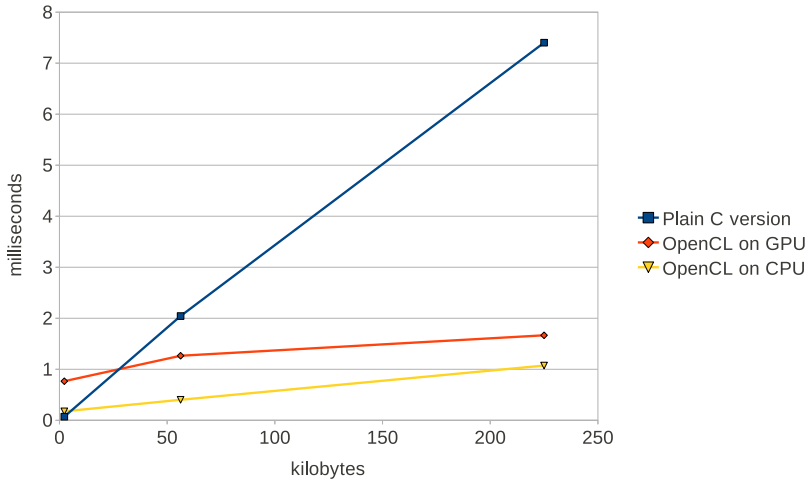

Figure 4.4: Performance measures for the standalone undistortion implementations, small image sizes.

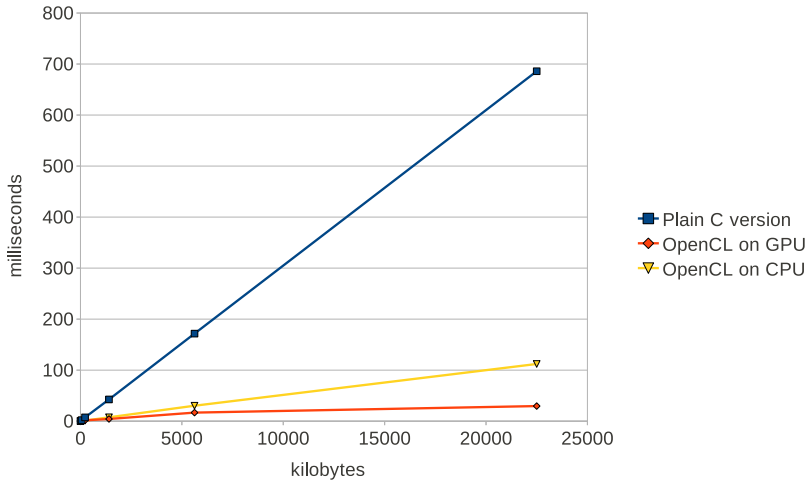

Figure 4.5: Performance measures for the standalone undistortion implementations, large image sizes.

Table 4.1: Observed plug-in performance with resolution 1600x900.

Implementation	FPS	% of CPU time per core	% of CPU time for all cores
No filter	13,45	24	3,7
Reference	5,01	98	12,2
OpenCL w. GPU	13,49	28	4,5
OpenCL w. CPU	12,36	23	23

When it comes to the timings for the standalone implementations, it's showed that the reference implementation is faster than the OpenCL implementations only for the smallest image size, just as revealed in the OpenSSL application. This continues to prove that OpenCL needs a certain minimal size of data to process to be able to hide the overhead from transferring data to the device and enqueue the kernels.

Regarding the plug-in performance it's interesting to see that even the OpenCL implementation using the CPU as compute device can sustain the cameras original frame rate for the largest image size. Even though the CPU doesn't perform as good as the GPU on larger image sizes, it still outperforms the reference implementations by several orders of magnitude due to that OpenCL utilizes all the 8 cores of the CPU which unlocks its full performance. The OpenCL implementation using CPU as compute device also performs much better in this application than with the AES application.

Chapter 5
Discussion and Conclusions

5.1 Discussion

5.1.1 Platform independence

With the notion that OpenCL is platform independent, the level to which this corresponds with the development experiences will be discussed here. Three aspects can be looked at when describing platform independence: design decisions, code portability and programming language knowledge.

Design decisions

Before the introduction of the OpenCL standard, developers had to be familiar with the specific hardware architecture they were writing software for. This knowledge is still required to make design decisions on OpenCL applications, for correct execution on the specific device.

One important difference in GPU architectures manifests itself in the wavefront size, described in section 2.3. For Nvidia GPUs, the wavefront size is 32 while in GPUs manufactured by ATI the size is 64. If defining a work-group smaller than the wavefront size, it'll result in under-utilized wavefronts with processing elements executing empty instructions. A work-group size of 32 will use a Nvidia GPU in the right way, but only get half the potential performance on an ATI GPU.

A critical thing to know about the hardware architecture is the supported data buffer types and sizes of the index space. OpenCL specifies a minimum set that should work on all platforms, but vendors can also choose to support larger sizes if their hardware does. For example, the CPU used in this thesis doesn't support image type buffers when the used GPU does. This means that code which utilizes image buffers will function on the GPU but not on the CPU.

The CPU does however support an bigger index space in three dimensions, with the size (1024,1024,1024) compared to the size (256,256,256) that the GPU is limited to. This leads to a similar problem with code that maps the entire index space on the CPU, it will not function if executed on the GPU.

Many of these architecture specific limitations can however be overcome with a more complex host program. There are API functions for querying the devices

for their capabilities. If something isn't supported, then the host program can chose a default value guaranteed by the OpenCL specification to work. This will however add more overhead time to the host because of the extra API function calls. And some limitations can't be disregarded, e.g. the use of images.

Code portability

Since each vendor is responsible for their API implementation to comply with the OpenCL specification, execution of an OpenCL application should give the same results independent of device vendor. Code written in this thesis did execute correctly on all tested devices, including the Nvidia device, but with big performance differences. The AES OpenCL application run on CPU was far from it's potential performance, when the GPU performed very well with the same code.

This proves that code optimized for one device may be unsuitable for an other device. Moving a working OpenCL application to a different and potentially better hardware will not guaranty better performance.

Still the definition of platform independence that OpenCL claims to be, is regarding correct execution. With this in mind, the OpenCL standard is considered to meet its requirements on platform independence.

Programming language knowledge

Most vendors today supporting OpenCL did provide their own SDK and programming language before the introduction of OpenCL. This forced the developer to learn a new language when programming for a new platform. This could be a substantial amount of time in the development process, adding cost and delays to the application.

There was a big difference in development time for the kernel code and host program, between the two applications, with the second being considerably shorter. This indicates that the programming language doesn't have to be relearned for a new device or application. Compared to developing applications in plain C, the use of OpenCL only caused a slight time increase. If added time is the concern when considering usage of OpenCL, it can be disregarded for a developer with previous acquaintance with OpenCL.

The knowledge in the OpenCL programming language can be used completely independent of device and vendor.

5.1.2 Host program

Looking at the results from the two applications, the writing and understanding of the host program was considered to be a quite easy task. OpenCL defines a clear structure of how to set up, use and finally tear down the host. This structure could then easily be applied when writing both applications and can probably be used when writing host programs for a large amount of other existing software as well.

The ability to integrate the host program into existing software was also considered to be an easy task. One instance of the host can be stored in a memory-allocated struct which can be flexibly handled as a kind of session variable.

It's also important to know that some of the performance tuning for the specific device can be made in the host. Due to this, the developer should not only tune the kernel code but also the host code when porting an OpenCL application between different platforms.

The fact that OpenCL defines a clear structure of the host program could also limit or complicate the integration of the host into several existing applications. If the application does not execute in a way that lets the developer call the set-up, run and tear-down functions in a natural way, this could complicate the integration of the host.

It's also worth noticing that the set-up phase of the host could take a significant amount of time. In the first application the host set-up function was measured, taking around 0.7 seconds using one device and around 1.4 seconds using two devices. This could be an important consideration since the time delay could slow down applications which requires the processing of data to start immediately after application startup.

5.1.3 Kernel code

Writing the kernels in OpenCL C code was one of the most important tasks when developing the applications. The fact that a significant amount of functionality in standard C is left out, didn't complicate the writing of the kernels. The added data types and functionality made the language intuitive and very efficient for writing data parallel functions.

Still, considerations had to be made to make sure that the work-items execute correctly in a massively parallel fashion. Work-items interfering with each other can also occur in OpenCL if they happen to access the same local or global memory space in a conflicting way. Developers experienced with threaded programming should feel familiar with how work-items behave in parallel, since synchronization primitives such as barriers also exists in OpenCL.

Memory access between the work-items was also made intuitive, thanks to the introduction of the index space which helps the programmer control which parts of the memory work-items should access and when.

5.1.4 Workload and algorithm considerations

It was discovered that the size of the workload and the computational intensity of the implemented algorithm is a very important consideration. The computational intensity defines how many arithmetic operations that has to be made on each data element.

Fairly little calculations per pixel was made in the lens distortion application. When correcting a 1600x900 image on the GPU with a total execution time of about 20 ms, only 2 ms were calculations, the rest was memory transfer over PCI-Express bus and API function calls. The AES application however had a very calculation-intensive algorithm, where the time for memory transfer was as

little as 1 % of total execution time. This shows that the transfer of data and API function calls adds overhead that even can be bigger than the actual calculation. It could even be so that the added overhead is greater than the reduced calculation time, in which case there is no need for an OpenCL implementation at all.

The expected workload size and calculation-intensity of an algorithm could be used as guidelines, when considering if the algorithm should be implemented in OpenCL.

The characteristics of the implemented algorithm can also lead to significant performance differences. This could be seen in the AES application case where the OpenCL implementation performed much worse when using the CPU as compute device compared to using the GPU as compute device. When using the CPU, the measured performance was similar to the performance of the sequential reference implementation, even though all cores of the CPU were utilized. In the lens distortion application however, both the GPU and CPU delivered good performance.

To find out why the CPU performed so bad in the AES application case, the kernel source code can be compared with the code from the lens distortion application. The notable differences are the use of the local memory and the calling of `barrier(CLK_LOCAL_MEM_FENCE)`, these were not used in the second application. The local memory area is an abstraction that is mapped to on-chip memory in the GPU, but to off-chip RAM in the CPU. Therefore the memory latencies of the local memory are longer on the CPU, adding to the total execution time of the AES application.

Also the use of barriers is adding to the total time. When the currently executing work-item in a CPU core reaches a barrier a context switch is performed, which takes much longer time than on a GPU. Since the barriers are called twice in each AES round, the total delay aggregated from the context switches makes the execution on CPU significantly slower.

5.1.5 Higher level software limitations

In a real-world situation, the host program and kernel will seldom be used in a standalone executable. Instead they will be integrated in other higher level software such as plug-ins, engines etc. using their functionality.

This puts a lot of requirements on the higher level software if it should enable the OpenCL host to fully utilize the performance of the compute device. The consequences of too small data block sizes can be seen in the OpenSSL application case. Here it was discovered that OpenSSL partitions the data to be encrypted into fragments sent into the engine for processing. Measurements showed that those fragments where too small for the OpenCL device to gain any benefits over the built-in OpenSSL implementation.

The circumstances showed not to be optimal in the lens distortion application either. The camera used for generating the video stream only provided a frame rate of 13-14 fps, which even at the highest video resolution couldn't max out the CPU used as compute device. This is also why the CPU managed to perform real-time frame rates on par with the GPU.

Despite this, the OpenCL application performed very well with barrel distortion filtering and the filtering computations can be moved from the CPU to the GPU. The CPU was greatly unburdened and could perform other important system operations.

5.1.6 Performance delivered by the devices

As seen in the performance result sections of both applications, the OpenCL-implemented algorithms does perform impressively well when executed on the GPU device. The AES standalone performed up to 60 times faster than its corresponding reference implementation and the lens distortion filter performed almost 25 times better. Worth noticing is that those applications were implemented and tested in almost optimal conditions. When running the standalones, sufficient data sizes were used and the algorithms were highly appropriate for parallel execution. But considering the very large performance gains, the circumstances don't even have to be optimal to gain a significant performance increase.

It's also satisfying to see that the OpenCL-implemented distortion filter performed very well on the CPU. This indicates that OpenCL could be used as the new standard method of implementing parallelized software on CPUs instead of common APIs such as pthreads or OpenMPI. Threaded implementations using those APIs were not tested in this thesis however. It will remain to see how well OpenCL on CPUs perform in comparison with well-written threaded implementations.

5.1.7 Vendor API implementation

Performance of the implementation

The supplied API implementation from the vendor reduces the developer's control over the device, as the developer can't alter the behavior of the API implementation. The developer just have to rely on that the API implementation is optimized, and that no significant performance is lost compared to developing with device-specific language or assembler code.

Section 2.3 describes the chain of abstraction layers that must be traversed from the call of an API function, to the point where the corresponding instruction is executed on the device hardware. When measuring the time for those calls, they showed to take very long time compared to the time of the actual work executed on the hardware. If these long API function call times comes from the many abstraction layers, or from lack of optimizations in ATI's implementation remains unknown for now.

Debugging features of the implementation

As seen in section 3.3.3 and 4.3.3, ATI's implementation has some compatibility issues with analysis tools such as valgrind. This could be a serious flaw since it complicates the procedures of maintaining stability in software that has an

OpenCL host integrated. Since it's an issue recognized by ATI, it's possible that it will be fixed in upcoming versions of the SDK.

When debugging and analyzing the standalone OpenCL-implementations on the Nvidia platform, no errors were reported by valgrind but some memory leaks were. Also, Nvidia's API implementation doesn't support the CPU as compute device. This shows that the choice of vendor of the compute device could be critical and must be done with great care. One should take all the features and limitations of the different vendors API implementation into consideration when choosing vendor of the compute device.

The debugging features via GDB was also tested and considered stable on both ATI's and Nvidia's platforms. It was satisfying to see that one can step through both the host and kernel code when debugging OpenCL software.

5.1.8 Deployment of OpenCL applications

If OpenCL is to be used in more widely spread consumer products, it must be easy to install software with OpenCL-functionality on all platforms supported by OpenCL. This means that each computer running the software must have a OpenCL-compliant device installed. The computer must also also contain device drivers with OpenCL support and libraries containing the API implementation from the vendor of the device.

Experiences from this thesis shows that the library containing the vendor's API implementation usually comes with the vendor's SDK. If the SDK isn't installed the application developer can choose to include the necessary vendor libraries with its software.

The support for OpenCL is expected to move from inclusion in SDKs to inclusion in device drivers that always need to be installed on the system. This would result in an easier deployment of OpenCL software.

5.2 Conclusions

From the application implementations, results and previous discussion we draw the following conclusions.

- OpenCL is platform independent in regards of the written code and programming language knowledge. But if the expected performance is to be obtained, device specific optimization is needed.
- The structure of the host program is flexible enough to be integrated into existing software.
- The OpenCL C programming language used to write kernels is mature and easy to learn if previously knowing standard C.
- A device's potential performance can be utilized by OpenCL, if the workload is suitable and the algorithm can be parallelized.
- The software using the OpenCL host can greatly limit the performance, depending on how it uses the host.

- The API implementation supplied by the device vendor affects the performance and the stability analysis of the OpenCL application.
- Libraries containing the vendor's API implementation should be easy to deploy if a suitable installation package is created.

Chapter 6
Summary

6.1 Comments

The purpose of this thesis was to investigate practical software development with OpenCL, focused on more commercial grade software. The work was carried out with very few problems encountered, almost surprisingly smooth. During this thesis some important discoveries were made, one of them was the current differences in the various vendors' implementations of the OpenCL API. Another being that OpenCL applications require certain conditions to perform optimally, e.g. workload sizes and which algorithms to conduct on which hardware.

With this in mind, the authors of this thesis consider the goals of the thesis to be fulfilled. The authors also hope that Axis will gain benefits of this work when developing their future products. Also that it could be an inspiration to other software vendors planning to incorporate new technologies for hardware acceleration into their products.

6.2 Future work

The era of programming heterogeneous systems is still relatively young and lots of research needs to be done in the years to come. Studies made during this thesis, on discussions around the use of OpenCL, found that the industry is very optimistic regarding the future of the standard. One significant future goal would be a wide-scale integration of OpenCL into many operating systems, not least mobile systems.

Its already confirmed that some of the market leading manufacturers of mobile platforms and architectures will incorporate OpenCL support in their future hardware [20]. The standard's embedded profile in combination with the ability to move computational intensive tasks to the more energy-efficient GPUs makes OpenCL very interesting for usage on mobile devices. Therefore it's suggested that the next step in research on OpenCL will be focused towards mobile platforms and software.

References

[1] AMD. AMD Brook+ documentation. http://developer.amd.com/gpu_assets/AMD-Brookplus.pdf.

[2] ATI. Stream SDK OpenCL programming guide. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx.

[3] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, pages 1–33, 2010.

[4] Citizendum. The AES Competition. http://en.citizendium.org/wiki/AES_competition.

[5] J. Daemen and V. Rijmen. The Rijndael Block Cipher specification. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[6] J. de Villiers, F. Leuschner, and R. Geldenhuys. Centi-pixel accurate real-time inverse distortion correction. *Proceedings of the 2008 International Symposium on Optomechatronic Technologies*, 7266:1–8, 2008.

[7] O. Gervasi, D. Russo, and F. Vella. The aes implantation based on opencl for multi/many core architecture. *2010 International Conference on Computational Science and Its Applications*, pages 129–134, 2010.

[8] GNOME. GLib reference manual. http://library.gnome.org/devel/glib/.

[9] GStreamer. Application development manual. http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/manual.pdf.

[10] C. Hulbert. Plain C implementation of AES. https://github.com/chrishulbert/crypto/raw/master/c/c_aes.c.

[11] Internet Engineering Task Force. The Transport Layer Security protocol. http://www.ietf.org/rfc/rfc2246.txt.

[12] L. G. Luis Alvarez and J. R. Sendra. Algebraic lens distortion model estimation. *Image Processing On Line*, 2010. http://dx.doi.org/10.5201/ipol.2010.ags-alde.

[13] Mozilla. The Secure Socket Layer protocol. http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt.

[14] National Institute of Standards and Technology. Block Cipher Modes of Operation. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[15] National Institute of Standards and Technology. The Advanced Encryption Standard specification. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[16] Nvidia. CUDA Programming Guide for CUDA Toolkit. http://developer.nvidia.com/object/gpucomputing.html.

[17] OpenSSL. Cryptography and SSL/TLS toolkit. http://www.openssl.org.

[18] OpenSSL. Engine API. http://www.openssl.org/docs/crypto/engine.html.

[19] J. Park, S.-C. Byun, and B.-U. Lee. Lens distortion correction using ideal image coordinates. *IEEE Transactions on Consumer Electronics*, 55(3):987–991, 2009.

[20] Qualcomm. Adreno GPU roadmap. http://developer.qualcomm.com/sites/default/files/IQ-Tech-Track-AdrenoGPUandPerformanceTools.pdf.

[21] The Khronos Group. The OpenCL standard. http://www.khronos.org/registry/cl/.

[22] The Valgrind Developers. valgrind, GPL-licensed tool suite for debugging and profiling of Linux programs. http://valgrind.org/info/about.html.

[23] S. Trenholme. The Rijndael key schedule. http://www.samiam.org/key-schedule.html.

[24] R. Tsuchiyama et al. *The OpenCL programming book*. Fixstars Corporation, 2010.

Appendix A

Code Appendix

A.1 AES Reference implementation

```
void change_order(byte *state)
{
  int i;
  byte temp[16];
  memcpy(temp,state,16);
  for(i=0;i<16;i++)
    state[i] = temp[change_order_table[i]];
}

// XOR's all elements in a n byte array a by b
void xor(byte *a, byte *b, int n) {
  int i;
  for (i=0;i<n;i++)
    a[i] ^= b[i];
}

// XOR the current cipher state by a specific round key
void xor_round_key(byte *state, byte *keys, int round) {
  xor(state,keys+round*16,16);
}

// Apply and reverse the rijndael s-box to all elements in an array
void sub_bytes(byte *a,int n) {
  int i;
  for (i=0;i<n;i++)
    a[i] = lookup_sbox[a[i]];
}
void sub_bytes_inv(byte *a,int n) {
  int i;
  for (i=0;i<n;i++)
    a[i] = lookup_sbox_inv[a[i]];
}
```

```
// Perform the core key schedule transform on 4 bytes,
// as part of the key expansion process

void key_schedule_core(byte *a, int i) {
  byte temp = a[0];
  a[0]=a[1];
  a[1]=a[2];
  a[2]=a[3];
  a[3]=temp;
  sub_bytes(a,4);
  a[0]^=lookup_rcon[i];

}

// Expand the 16-byte key to 11 round keys (176 bytes)
void expand_key(byte *key, byte *keys) {
  int bytes=16;
  int i=1;
  int j;
  byte t[4];
  memcpy(keys,key,16);
  change_order(keys);
  while (bytes<176) {
    memcpy(t,keys+bytes-4,4);
    key_schedule_core(t, i);
    i++;
    xor(t,keys+bytes-16,4);
    memcpy(keys+bytes,t,4);
    bytes+=4;


    for (j=0;j<3;j++) {
      memcpy(t,keys+bytes-4,4);
      xor(t,keys+bytes-16,4);
      memcpy(keys+bytes,t,4);
      bytes+=4;
    }
  }
}

// Apply / reverse the shift rows step on the 16 byte cipher state
void shift_rows(byte *state) {
  int i;
  byte temp[16];
  memcpy(temp,state,16);
  for (i=0;i<16;i++)
```

```
    state[i]=temp[shift_rows_table[i]];
}
void shift_rows_inv(byte *state) {
  int i;
  byte temp[16];
  memcpy(temp,state,16);
  for (i=0;i<16;i++)
    state[i]=temp[shift_rows_table_inv[i]];
}

// Perform the mix columns matrix on one column of 4 bytes
void mix_col (byte *state) {
  byte a0 = state[0];
  byte a1 = state[1];
  byte a2 = state[2];
  byte a3 = state[3];
  state[0] = lookup_g2[a0] ^ lookup_g3[a1] ^ a2 ^ a3;
  state[1] = lookup_g2[a1] ^ lookup_g3[a2] ^ a3 ^ a0;
  state[2] = lookup_g2[a2] ^ lookup_g3[a3] ^ a0 ^ a1;
  state[3] = lookup_g2[a3] ^ lookup_g3[a0] ^ a1 ^ a2;
}

// Perform the mix columns matrix on each column of the 16 bytes
void mix_cols (byte *state) {
  mix_col(state);
  mix_col(state+4);
  mix_col(state+8);
  mix_col(state+12);
}

// Perform the inverse mix columns matrix on one column of 4 bytes
void mix_col_inv (byte *state) {
  byte a0 = state[0];
  byte a1 = state[1];
  byte a2 = state[2];
  byte a3 = state[3];
  state[0] = lookup_g14[a0] ^ lookup_g9[a3] ^ lookup_g13[a2]
  ^ lookup_g11[a1];
  state[1] = lookup_g14[a1] ^ lookup_g9[a0] ^ lookup_g13[a3]
  ^ lookup_g11[a2];
  state[2] = lookup_g14[a2] ^ lookup_g9[a1] ^ lookup_g13[a0]
  ^ lookup_g11[a3];
  state[3] = lookup_g14[a3] ^ lookup_g9[a2] ^ lookup_g13[a1]
  ^ lookup_g11[a0];
}

// Perform the inverse mix columns matrix on each column of
```

```
// the 16 bytes
void mix_cols_inv (byte *state) {
  mix_col_inv(state);
  mix_col_inv(state+4);
  mix_col_inv(state+8);
  mix_col_inv(state+12);
}
```

A.2 OpenCL host for AES implementation

```
// Function that expands the key to an expanded key
void expandKey(unsigned char *key)
{
    int i;
    for(i = 16; i < 176; i++) {
        switch((i%16)) {
            case 0:
                key[i] = sbox[key[i-9]]^Rcon[i/16]^key[i-16];
                break;
            case 4:
            case 8:
                key[i] = sbox[key[i-9]]^key[i-16];
                break;
            case 12:
                key[i] = sbox[key[i-25]]^key[i-16];
                break;
            default:
                key[i] = key[i-1]^key[i-16];
                break;
        }
    }
}

// Function that expands the key and initiates the host_data_t
// struct
int host_init(host_data_t *hd, const unsigned char *key, int enc,
                                                              int nid)
{
    FILE *fp;
    char *source_str;
    size_t source_size;

    // Get kernel source
    fp = fopen("aes_crypto_kernels.cl", "r");
    if (!fp) {
```

```
        fprintf(stderr, "Failed to load kernel.\n");
     exit(1);
     }
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );

// Check availability of AMD platform
cl_uint numPlatforms;
cl_uint    numGPU;
cl_int status;
status = 0;

clGetPlatformIDs(0, NULL, &numPlatforms);

cl_platform_id platform_list[numPlatforms];
hd->platform = NULL;

clGetPlatformIDs(numPlatforms, platform_list, NULL);

int i;
for(i = 0; i < numPlatforms; i++) {
    char vendorstring[100];
    clGetPlatformInfo(platform_list[i], CL_PLATFORM_VENDOR,
                      sizeof(vendorstring), vendorstring, NULL);

    if(!strcmp(vendorstring, "Advanced Micro Devices, Inc.")) {
                                hd->platform = platform_list[i];
        break;
    }
}
if(!hd->platform) // Found no platform
    return 0;

// Check availability of GPU device, otherwise pick the CPU
clGetDeviceIDs(hd->platform, CL_DEVICE_TYPE_GPU, 0, NULL,
                                                        &numGPU);

if(numGPU > 3) {
    clGetDeviceIDs(hd->platform, CL_DEVICE_TYPE_GPU, 1,
                                            &hd->device, NULL);
}
else {
    clGetDeviceIDs(hd->platform, CL_DEVICE_TYPE_CPU, 1,
                                            &hd->device, NULL);
}
```

```
// Create context and command queue
hd->context = clCreateContext(NULL, 1, &hd->device, NULL, NULL,
                                                          &status);
ERRCHK(status, "Failed to create context")

hd->command_queue = clCreateCommandQueue(hd->context,
                                        hd->device, 0, &status);
ERRCHK(status, "Failed to create commandqueue")

// Create the expanded key
unsigned char expanded_key[176];
for(i = 0; i < 16; i++)
    expanded_key[i] = key[i];
expandKey(expanded_key);

// Create buffers for sBox or rsBox, and expanded key.
// Copy contents from their host pointer to device
if(enc) {
    hd->sBoxBuffer = clCreateBuffer(hd->context,
                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           sizeof(cl_uchar)*256, sbox, &status);
    ERRCHK(status, "Failed to create sBox buffer")
} else {
    hd->sBoxBuffer = clCreateBuffer(hd->context,
                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                          sizeof(cl_uchar)*256, rsbox, &status);
    ERRCHK(status, "Failed to create rsBox Buffer")
}

hd->key_buffer = clCreateBuffer(hd->context, CL_MEM_READ_ONLY |
                    CL_MEM_COPY_HOST_PTR, sizeof(cl_uchar)*176,
                                        &expanded_key, &status);
ERRCHK(status, "Failed to create key buffer")

// Create program from kernel source
hd->program = clCreateProgramWithSource(hd->context, 1,
                                    (const char **)&source_str,
                         (const size_t *)&source_size, &status);
ERRCHK(status, "Failed to create program object")

// Build the kernel program
status = clBuildProgram(hd->program, 1, &hd->device,
                                                NULL, NULL, NULL);
if(status != CL_SUCCESS) {
    char log[500];
    clGetProgramBuildInfo(hd->program, hd->device,
                    CL_PROGRAM_BUILD_LOG,sizeof(log),log,NULL);
```

```
        printf("Build log:\n%s\n",log);
    }
    ERRCHK(status, "Failed to compile kernel")

    // Create the kernel object
    if(enc) {
        if(nid == 418) {
            hd->kernel = clCreateKernel(hd->program,
                                        "aes_encrypt_ecb", &status);
            hd->kernel_nbr = 0;
        } else {
            hd->kernel = clCreateKernel(hd->program,
                                        "aes_encrypt_cbc", &status);
            hd->kernel_nbr = 2;
        }
    } else {
        if(nid == 418) {
            hd->kernel = clCreateKernel(hd->program,
                                        "aes_decrypt_ecb", &status);
            hd->kernel_nbr = 1;
        } else {
            hd->kernel = clCreateKernel(hd->program,
                                        "aes_decrypt_cbc", &status);
            hd->kernel_nbr = 3;
        }
    }
    ERRCHK(status, "Failed to create kernel object")

    // Set the initial kernel arguments
    status = clSetKernelArg(hd->kernel, 2, sizeof(cl_mem),
                                           (void *)&hd->key_buffer);
    ERRCHK(status, "Failed to set kernel arg 2")

    status = clSetKernelArg(hd->kernel, 3, sizeof(cl_mem),
                                           (void *)&hd->sBoxBuffer);
    ERRCHK(status, "Failed to set kernel arg 3")

    free(source_str);

    return 1;

}

int host_run(host_data_t *hd, const unsigned char *inptr,
             unsigned char *outptr, size_t len, unsigned char *iv)
{
    cl_int status;
```

```
cl_event event[2];

size_t globalSize;
size_t localSize;
size_t workBytes = len;
if(hd->kernel_nbr == 2) {
    globalSize = 4;
    localSize = 4;
} else {
    if(len%256)
        workBytes += 256-len%256;
    globalSize = workBytes/4;
    localSize = 64;
}

// Check if code exeeds the device implementation

// Create input buffer
cl_mem input_buffer = clCreateBuffer(hd->context,
                                                CL_MEM_READ_ONLY,
                     sizeof(cl_uchar)*workBytes, NULL, &status);
ERRCHK(status, "Failed to create input buffer")

// Write the data to device buffer
status = clEnqueueWriteBuffer(hd->command_queue, input_buffer,
                              CL_TRUE, 0, sizeof(cl_uchar)*len,
                                    inptr, 0, NULL, &event[0]);
ERRCHK(status, "Failed to write the data")

// Create output buffer
cl_mem output_buffer = clCreateBuffer(hd->context,
                                                CL_MEM_WRITE_ONLY,
                    sizeof(cl_uchar)*workBytes, NULL, &status);
ERRCHK(status, "Failed to create output buffer")

if(hd->kernel_nbr > 1) {
    cl_mem initial_vector = clCreateBuffer(hd->context,
                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           sizeof(cl_uchar)*16, iv, &status);
    ERRCHK(status, "Failed to create initial vector")

    status = clSetKernelArg(hd->kernel, 4, sizeof(cl_mem),
                                       (void *)&initial_vector);
    ERRCHK(status, "Failed to set kernel arg 4")
}

if(hd->kernel_nbr == 2) {
```

```
        cl_uint nbrblocks = len/16;
        status = clSetKernelArg(hd->kernel, 5, sizeof(cl_uint),
                                                (void*)&nbrblocks);
        ERRCHK(status, "Failed to set kernel arg 5")
    }

    // Set kernel args to the new memory buffers
    status = clSetKernelArg(hd->kernel, 0, sizeof(cl_mem),
                                           (void *)&output_buffer);
    ERRCHK(status, "Failed to set kernel arg 0")

    status = clSetKernelArg(hd->kernel, 1, sizeof(cl_mem),
                                            (void *)&input_buffer);
    ERRCHK(status, "Failed to set kernel arg 1")

    // Enqueue the kernel
    status = clEnqueueNDRangeKernel(hd->command_queue, hd->kernel,
                                1, NULL, &globalSize, &localSize,
                                       1, &event[0], &event[1]);
    ERRCHK(status, "Failed to enqueue kernel")

    // Read the result to application pointer
    status = clEnqueueReadBuffer(hd->command_queue, output_buffer,
                                CL_TRUE, 0, sizeof(cl_uchar)*len,
                                     outptr, 1, &event[1], NULL);
    ERRCHK(status, "Failed to read the result")

    clReleaseEvent(event[0]);
    clReleaseEvent(event[1]);
    clReleaseMemObject(input_buffer);
    clReleaseMemObject(output_buffer);

    return 1;
}

int host_clean(host_data_t *hd)
{
    clFlush(hd->command_queue);
    clFinish(hd->command_queue);
    clReleaseKernel(hd->kernel);
    clReleaseProgram(hd->program);
    clReleaseMemObject(hd->sBoxBuffer);
    clReleaseMemObject(hd->key_buffer);
    clReleaseCommandQueue(hd->command_queue);
    clReleaseContext(hd->context);

    return 1;
```

```
}
```

A.3 Kernel source for AES implementation

```
//the galois multiplication in the finit field
uchar gal_mult(uchar a, uchar b)
{
    uchar p = 0;
    for(unsigned int i = 0; i < 8; i++){
        if(b&1)
            p ^= a;
        uchar hiBitSet = (a & 0x80);
        a <<= 1;
        if(hiBitSet)
            a ^= 0x1b;
        b >>= 1;
    }
    return p;
}

//substitute each byte for corresponding byte in sbox
inline uchar4 sub_byte(__global uchar *sbox, uchar4 block)
{
    return (uchar4)(sbox[block.x], sbox[block.y], sbox[block.z],
                                                    sbox[block.w]);
}

//mix the columns of the block with galois matrix multiplication
uchar4 mix_col(__local uchar4 *block, __private uchar4 *gal_coeff,
                                        uint start, uint rowID)
{
    uchar4 c;

    c.x = gal_mult(block[start].x, gal_coeff[rowID].x) ^
        gal_mult(block[start+1].x, gal_coeff[rowID].y) ^
        gal_mult(block[start+2].x, gal_coeff[rowID].z) ^
         gal_mult(block[start+3].x, gal_coeff[rowID].w);

    c.y = gal_mult(block[start].y, gal_coeff[rowID].x) ^
        gal_mult(block[start+1].y, gal_coeff[rowID].y) ^
        gal_mult(block[start+2].y, gal_coeff[rowID].z) ^
         gal_mult(block[start+3].y, gal_coeff[rowID].w);

    c.z = gal_mult(block[start].z, gal_coeff[rowID].x) ^
        gal_mult(block[start+1].z, gal_coeff[rowID].y) ^
```

```
        gal_mult(block[start+2].z, gal_coeff[rowID].z) ^
         gal_mult(block[start+3].z, gal_coeff[rowID].w);

    c.w = gal_mult(block[start].w, gal_coeff[rowID].x) ^
        gal_mult(block[start+1].w, gal_coeff[rowID].y) ^
        gal_mult(block[start+2].w, gal_coeff[rowID].z) ^
         gal_mult(block[start+3].w, gal_coeff[rowID].w);

    return c;
}

//shift rows to the left
uchar4 shift_rows(uchar4 row, uint rowID)
{
    uchar4 r;
    switch(rowID){
        case 0:
            r = row.xyzw;
            break;
        case 1:
            r = row.yzwx;
            break;
        case 2:
            r = row.zwxy;
            break;
        case 3:
            r = row.wxyz;
            break;
        default:
            r = row;
            break;
    }
    return r;
}

//shift rows to the right
uchar4 shift_rows_inv(uchar4 row, uint rowID)
{
    switch(rowID){
        case 0:
            row = row.xyzw;
            break;
        case 1:
            row = row.wxyz;
            break;
        case 2:
            row = row.zwxy;
```

```
            break;
        case 3:
            row = row.yzwx;
            break;
        default:
            break;
    }
    return row;
}

//block that does aes cipher encryption
uchar4 block_cipher_encrypt(__global uchar4 *key,
        __global uchar *sbox,
        __local uchar4 *block0,
        __local uchar4 *block1,
        uchar4 in)
{
    uint localID = get_local_id(0);
    uint rowID = localID%4;

    uchar4 gal_coeff[4];
    gal_coeff[0] = (uchar4)(2, 3, 1, 1);
    gal_coeff[1] = (uchar4)(1, 2, 3, 1);
    gal_coeff[2] = (uchar4)(1, 1, 2, 3);
    gal_coeff[3] = (uchar4)(3, 1, 1, 2);

    //fill local block and add round key
    block0[localID]  = in^key[rowID];

    for(uint r = 1; r < 10; r++){
        block0[localID] = sub_byte(sbox, block0[localID]);

        barrier(CLK_LOCAL_MEM_FENCE);
        block1[localID] = shift_rows(block0[localID], rowID);

        barrier(CLK_LOCAL_MEM_FENCE);
        block0[localID] = mix_col(block1, gal_coeff,
                                          localID-rowID, rowID);
        block0[localID] = block0[localID]^key[r*4 + rowID];
    }

    block0[localID] = sub_byte(sbox, block0[localID]);
    block0[localID] = shift_rows(block0[localID], rowID);

    return block0[localID]^key[40 + rowID];
}
```

```
//block that does aes cipher decryption
uchar4 block_cipher_decrypt(__global uchar4 *key,
        __global uchar *sbox,
        __local uchar4 *block0,
        __local uchar4 *block1,
        uchar4 in)
{
    uint localID = get_local_id(0);
    uint rowID = localID%4;

    uchar4 gal_coeff[4];
    gal_coeff[0] = (uchar4)(14, 11, 13, 9);
    gal_coeff[1] = (uchar4)(9, 14, 11, 13);
    gal_coeff[2] = (uchar4)(13, 9, 14, 11);
    gal_coeff[3] = (uchar4)(11, 13, 9, 14);

    //fill local block and add round key
    block0[localID]  = in^key[40 + rowID];

    for(uint r = 9; r > 0; r--){
        block0[localID] = shift_rows_inv(block0[localID], rowID);

        block0[localID] = sub_byte(sbox, block0[localID]);

        barrier(CLK_LOCAL_MEM_FENCE);
        block1[localID] = block0[localID]^key[r*4 + rowID];

        barrier(CLK_LOCAL_MEM_FENCE);
        block0[localID] = mix_col(block1, gal_coeff,
                                          localID-rowID, rowID);
    }

    block0[localID] = shift_rows_inv(block0[localID], rowID);
    block0[localID] = sub_byte(sbox, block0[localID]);

    return block0[localID]^key[rowID];
}


__kernel void aes_encrypt_ecb(__global uchar4 *output,
                __global uchar4 *input,
                __global uchar4 *key,
                __global uchar *sbox)
{
    uint globalID = get_global_id(0);
    __local uchar4 block0[64], block1[64];
    output[globalID] = block_cipher_encrypt(key,sbox,block0,block1,
```

```
                                                    input[globalID]);
}

__kernel void aes_decrypt_ecb(__global uchar4 *output,
                __global uchar4 *input,
                __global uchar4 *key,
                __global uchar *sbox)
{
    uint globalID = get_global_id(0);
    __local uchar4 block0[64], block1[64];
    output[globalID] = block_cipher_decrypt(key,sbox,block0,block1,
                                                    input[globalID]);
}

__kernel void aes_encrypt_cbc(__global uchar4 *output,
                __global uchar4 *input,
                __global uchar4 *key,
                __global uchar *sbox,
        __global uchar4 *iv,
        uint nbrblocks)
{
    uint globalID = get_global_id(0);
    __local uchar4 block0[4], block1[4];
    uchar4 result;
    result = iv[globalID] ^ input[globalID];
    result = block_cipher_encrypt(key,sbox,block0,block1,result);
    output[globalID] = result;
    uint i;
    for(i = 1; i < nbrblocks; i++) {
        result = result ^ input[globalID + 4*i];
        result = block_cipher_encrypt(key,sbox,block0,block1,
                                                        result);
        output[globalID + 4*i] = result;
    }
}

__kernel void aes_decrypt_cbc(__global uchar4 *output,
                __global uchar4 *input,
                __global uchar4 *key,
                __global uchar *sbox,
        __global uchar4 *iv)
{
    uint globalID = get_global_id(0);
    __local uchar4 block0[64], block1[64];
    uchar4 result;
    result = block_cipher_decrypt(key,sbox,block0,block1,
                                                    input[globalID]);
```

```
    if(globalID > 3) {
        output[globalID] = result ^ input[globalID-4];
    } else {
        output[globalID] = result ^ iv[globalID];
    }
}
```

A.4 Lens distortion filter reference implementation

```
void undistort_image_bgra(float *a,
        unsigned char *image_in,
        unsigned char *image_out,
        int width, int height)
{
    int xu, yu, xd, yd;
    long pu, pd;
    float xd_float, yd_float, xu_float, yu_float, xd_delta;
    float yd_delta;
    float xc, yc;
    float r, r_delta, r_norm;
    float weight;
    float pt[4];

    xc = width/2.0;
    yc = height/2.0;
    r_norm = sqrt(xc*xc+yc*yc);

    for(yu = 0; yu < height; yu++) {
        for(xu = 0; xu < width; xu++) {
            pu = (yu*width+xu)*4;
            /* scale the image with a[3] to see more */
            xu_float = (xu-xc)*a[3]+xc;
            yu_float = (yu-yc)*a[3]+yc;

            /* translating radius r to relative movment r_delta */
            r = sqrt((xu_float-xc)*(xu_float-xc)+(yu_float-yc)*
                                                    (yu_float-yc));
            r = r/r_norm;
            r_delta = a[2]*r*r;
            r_delta = (r_delta+a[1])*r*r;
            r_delta = (r_delta+a[0])*r*r;

            // using r_delta to find distorted coordinates (xd,yd)
            xd_float = xu_float+(xu_float-xc)*r_delta;
            yd_float = yu_float+(yu_float-yc)*r_delta;
```

```
xd = (int) xd_float; /* integer part */
yd = (int) yd_float; /* integer part */
xd_delta = xd_float-xd; /* fraction part */
yd_delta = yd_float-yd; /* fraction part */

/* clamping coordinates to border */
if(xd < 0) {
    xd = 0;
    xd_delta = 0;
}
if(xd > width-2) {
    xd = width-2;
    xd_delta = 1;
}
if(yd < 0) {
    yd = 0;
    yd_delta = 0;
}
if(yd > height-2) {
    yd = height-2;
    yd_delta = 1;
}

/* contribution from upper left pixel */
pd = (yd*width+xd)*4;
weight = (1.0-xd_delta)*(1.0-yd_delta);
pt[0] = (float) weight*image_in[pd];
pt[1] = (float) weight*image_in[pd+1];
pt[2] = (float) weight*image_in[pd+2];
pt[3] = (float) weight*image_in[pd+3];

/* contribution from upper right pixel */
pd += 4;
weight = xd_delta*(1.0-yd_delta);
pt[0] += (float) weight*image_in[pd];
pt[1] += (float) weight*image_in[pd+1];
pt[2] += (float) weight*image_in[pd+2];
pt[3] += (float) weight*image_in[pd+3];

/* contribution from lower right pixel */
pd += width*4;
weight = xd_delta*yd_delta;
pt[0] += (float) weight*image_in[pd];
pt[1] += (float) weight*image_in[pd+1];
pt[2] += (float) weight*image_in[pd+2];
pt[3] += (float) weight*image_in[pd+3];
```

```
            /* contribution from lower left pixel */
            pd -= 4;
            weight = (1.0-xd_delta)*yd_delta;
            pt[0] += (float) weight*image_in[pd];
            pt[1] += (float) weight*image_in[pd+1];
            pt[2] += (float) weight*image_in[pd+2];
            pt[3] += (float) weight*image_in[pd+3];

            /* store the correct pixel value */
            image_out[pu] = (unsigned char) pt[0];
            image_out[pu+1] = (unsigned char) pt[1];
            image_out[pu+2] = (unsigned char) pt[2];
            image_out[pu+3] = (unsigned char) pt[3];
        }
    }
}
```

A.5 OpenCL host for lens distortion filter

```
#define MAX_SOURCE_SIZE (0x100000)
typedef struct {
    cl_context context;
    cl_kernel kernel;
    cl_command_queue command_queue;
    cl_mem in_data;
    cl_mem out_data;
} data_struct_t;

int host_setup(data_struct_t *ds, int gpu, float *a, int width,
                                                        int height)
{
    /* setup host */
    int i;
    FILE *source_fp;
    char *source_string;
    size_t source_size;
    char log[500];
    cl_int status;
    cl_platform_id platform;
    cl_device_id device;
    cl_program program;
    cl_float r;

    status = 0;
    r = sqrt((width*width+height*height)/4);
```

```
/* read kernel source */
source_fp = fopen("kernel_buffer.cl", "r");
if(!source_fp) {
    fprintf(stderr, "Failed to load kernel.\n");
    exit(1);
}
source_string = (char *) malloc(MAX_SOURCE_SIZE);
source_size = fread(source_string, 1, MAX_SOURCE_SIZE,
                                                    source_fp);
fclose(source_fp);

/* get platform and device */
platform = NULL;
status = clGetPlatformIDs(1, &platform, NULL);
ERRCHK(status, "Failed to get platform");

device = NULL;
if(gpu)
    status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
                                                &device, NULL);
else
    status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1,
                                                &device, NULL);
ERRCHK(status, "Failed to get device");

/* create context and command queue */
ds->context = clCreateContext(NULL, 1, &device, NULL, NULL,
                                                      &status);
ERRCHK(status, "Failed to create context");
ds->command_queue = clCreateCommandQueue(ds->context, device,
                                                  0, &status);
ERRCHK(status, "Failed to create command queue");

/* create image objects on device */
ds->in_data = clCreateBuffer(ds->context, CL_MEM_READ_ONLY,
                sizeof(cl_uchar4)*width*height, NULL, &status);
ERRCHK(status, "Failed to create input image buffer");
ds->out_data = clCreateBuffer(ds->context, CL_MEM_WRITE_ONLY,
                sizeof(cl_uchar4)*width*height, NULL, &status);
ERRCHK(status, "Failed to create output image buffer");

/* create program from kernel source */
program = clCreateProgramWithSource(ds->context, 1,
                                (const char **) &source_string,
                        (const size_t *) &source_size, &status);
ERRCHK(status, "Failed to create program");
```

```
    /* build the program */
    status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
    if(status != CL_SUCCESS) {
        clGetProgramBuildInfo(program, device,CL_PROGRAM_BUILD_LOG,
                                            sizeof(log), log, NULL);
        printf("Build log:\n%s\n", log);
    }
    ERRCHK(status, "Failed to build program");

    /* create kernel object */
    ds->kernel = clCreateKernel(program, "correct_image", &status);
    ERRCHK(status, "Failed to create kernel");

    /* set initial kernel arguments */
    status = clSetKernelArg(ds->kernel, 0, sizeof(cl_float4), a);
    ERRCHK(status, "Failed to set kernel arg 0, a");
    status = clSetKernelArg(ds->kernel, 1, sizeof(cl_mem),
                                             (void *) &ds->in_data);
    ERRCHK(status, "Failed to set kernel arg 1, in_data");
    status = clSetKernelArg(ds->kernel, 2, sizeof(cl_mem),
                                            (void *) &ds->out_data);
    ERRCHK(status, "Failed to set kernel arg 2, out_data");
    status = clSetKernelArg(ds->kernel, 3, sizeof(cl_float), &r);
    ERRCHK(status, "Failed to set kernel arg 3, r");

    clReleaseProgram(program);
    free(source_string);

    return 1;
}

int host_correct_image(data_struct_t *ds, unsigned char *image_in,
                   unsigned char *image_out, int width, int height)
{
    cl_int status;
    cl_event event;
    size_t global_size[2], local_size[2];

    // max workgroup size is 256, number of workgroups must be even
    global_size[0] = width;
    global_size[1] = height;
    local_size[0] = 32;
    local_size[1] = 2;

    /* write input image */
    status = clEnqueueWriteBuffer(ds->command_queue, ds->in_data,
                        CL_TRUE, 0, sizeof(cl_uchar4)*width*height,
```

```
                                                image_in, 0, NULL, NULL);
    ERRCHK(status, "Failed to write the input image");

    /* enqueue the kernel */
    status = clEnqueueNDRangeKernel(ds->command_queue, ds->kernel,
                              2, NULL, (const size_t *) &global_size,
                    (const size_t *) &local_size, 0, NULL, &event);
    ERRCHK(status, "Failed to enqueue the kernel");

    /* read corrected image */
    status = clEnqueueReadBuffer(ds->command_queue, ds->out_data,
                         CL_TRUE, 0, sizeof(cl_uchar4)*width*height,
                                          image_out, 1, &event, NULL);
    ERRCHK(status, "Failed to read the corrected image");

    /* release memory */
    clReleaseEvent(event);

    return 1;
}

int host_clean(data_struct_t *ds)
{
    /* clean up the context */
    clFlush(ds->command_queue);
    clFinish(ds->command_queue);
    clReleaseKernel(ds->kernel);
    clReleaseMemObject(ds->in_data);
    clReleaseMemObject(ds->out_data);
    clReleaseCommandQueue(ds->command_queue);
    clReleaseContext(ds->context);

    return 1;
}
```

A.6 Kernel source for lens distortion filter

```
__kernel void correct_image(float4 a, __global uchar4 *in_data,
                                  __global uchar4 *out_data, float r)
{
    long pu, pd;
    int2 size;
    size.x = get_global_size(0);
    size.y = get_global_size(1);
    int2 ui, di;
```

```
    float2 df, dd, uf;
    ui.x = get_global_id(0);
    ui.y = get_global_id(1);
    float2 c;
    c.x = size.x/2.0;
    c.y = size.y/2.0;
    float n, m, weight;
    pu = ui.y*size.x+ui.x;
    uf.x = (ui.x-c.x)*a.w+c.x;
    uf.y = (ui.y-c.y)*a.w+c.y;
    n = sqrt((uf.x-c.x)*(uf.x-c.x)+(uf.y-c.y)*(uf.y-c.y));
    n = n/r;
    m = a.z*n*n;
    m = (m+a.y)*n*n;
    m = (m+a.x)*n*n;
    df.x = uf.x+(uf.x-c.x)*m;
    df.y = uf.y+(uf.y-c.y)*m;
    di = convert_int2(df);
    dd = df-convert_float2(di);
    if(di.x < 0) {
        di.x = 0;
        dd.x = 0;
    } else if(di.x > size.x-2) {
        di.x = size.x-2;
        dd.x = 1;
    }
    if(di.y < 0) {
        di.y = 0;
        dd.y = 0;
    } else if(di.y > size.y-2) {
        di.y = size.y-2;
        dd.y = 1;
    }
    pd = di.y*size.x+di.x;
    weight = (1.0-dd.x)*(1.0-dd.y);
    float4 color = weight*convert_float4(in_data[pd]);
    pd += 1;
    weight = dd.x*(1.0-dd.y);
    color += weight*convert_float4(in_data[pd]);
    pd += size.x;
    weight = dd.x*dd.y;
    color += weight*convert_float4(in_data[pd]);
    pd -= 1;
    weight = (1.0-dd.x)*dd.y;
    color += weight*convert_float4(in_data[pd]);
    out_data[pu] = convert_uchar4(color);
}
```