MASARYK UNIVERSITY
FACULTY OF INFORMATICS

OpenCL Cryptographic Library

MASTER THESIS

Bc. Martin Preisler

Brno, May 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Martin Preisler

Advisor: RNDr. Adam Rambousek

Acknowledgement

Hereby I would like to thank my thesis advisor RNDr. Adam Rambousek and technical consultant Dr. Ir. Nikolaos Mavrogiannopoulos for valuable pointers, encouragement and support.

I would also like to thank Ing. Peter Vrabec for the unique opportunity to work on this project at Red Hat, Inc.

Last but not least, I am very grateful to Adam Preisler, Bc. Jan Štěpnička and Red Hat, Inc. for lending me various hardware for testing.

Abstract

Modern GPUs are devices with very high parallelism for a very low cost. Integer and logic instruction support enable us to use them for many workloads unrelated to rendering. Cryptographic algorithms like AES or Blowfish can benefit from being executed on the system's GPU. Such execution off-loads work from the main CPU, freeing it to do other tasks on a server system. For bulk encryption and decryption the whole operation can often be faster as well. This thesis describes implementation of an OpenCL library of commonly used ciphers — AES-ECB, AES-CTR, AES-GCM and Blowfish-ECB. Integrations of the library with existing software are included. The library provides abstractions that enable easy implementation of additional ciphers in the future.

Keywords

OpenCL, cryptography, AES, Blowfish, ECB, CTR, GCM, CUDA, GPU Compute, GPGPU

Contents

1 Introduction

*GPU*s advanced rapidly by adopting and optimizing 16bit floating point operations. This enabled real-time 3D rendering. At first all rendering was done using a so called *fixed pipeline*. With *fixed pipeline* developers were tied to a limited set of functionality. As *GPU*s got more complex and more and more rendering features were bolted into the *fixed pipeline*, there was a need to program *GPU*s directly. Low-level *GPU* programming has been introduced. The *GPU* programs were called *shaders* because they enabled developers to implement various different shading models not available in *fixed pipeline*. This led to a revolution in the graphics rendering world. Many new rendering techniques were being created every month. At this stage shaders still had to fit into a pipeline. There were two types — *vertex shaders* and *pixel shaders*.

Despite this limitation, *shaders* could be used to solve other problems than just rendering — *n-body simulations*, *heat transfer problems* and others. Solving these was a hack at the time. Developers had to adapt the problem to fit the pipeline and solving a problem meant rendering an image that represented the result. For iterative simulations the results image was fed back as a texture into the algorithm to advance another step. Big focus on floating point operations meant that integer processing was incomparably slow or even absent. This was a non-issue for many problems but it made it difficult or even impossible to solve some problems. 16-bit floating point is not a good replacement for 32-bit integers. It was also quite difficult to implement for these *GPU*s. There were many different shading languages, all quite low-level, similar to assembly. Writing a portable solution to a problem meant writing many different low-level shaders or relying on a meta-language like *NVIDIA Cg*.

Cg was a collaboration between *NVIDIA* and *Microsoft*. It promised to simplify development by enabling developers to write in a familiar C-like high-level syntax that was compiled into low-level shader code. Despite focus on *NVIDIA* hardware the *Cg* language was quite usable on many different *GPU*s. In 2009 *OpenGL ARB* announced *GLSL* — an official high-level shading language of *OpenGL*. Similarly, *Microsoft* introduced *HLSL* as an official high-level shading language of *DirectX*. Most developers moved to either *GLSL* or

HLSL and *Cg* was discontinued in 2012 [5]. Shader development was quite common at this stage but using *GPU*s for generic problem solving was still difficult.

NVIDIA reacted to these events and started *Compute Unified Device Architecture — CUDA* [15]. *CUDA* brought a high-level *C*-like language, fast integer and logic instructions, and much better debugging tools to *GPU Compute* developers. Two years later, *OpenCL* was introduced as a vendor-neutral alternative to *CUDA*. *OpenCL* was well received by *AMD* and *Intel* because it finally enabled *GPU Compute* outside *NVIDIA* hardware. *CUDA* and *OpenCL* allowed developers to implement their problem solvers without having to fit them into a rendering pipeline. At this stage *GPU Compute* took off and started to be used even outside laboratories for video encoding, video compositing, finance simulations, machine learning, and more. [9].

One of suitable tasks for *GPU Compute* is symmetric block cryptography. Each block can usually be processed by a different thread which makes the problem a perfect fit for massively parallel *GPU*s.

In this thesis we will explore ways to implement well-known cryptographic algorithms like *AES* and *Blowfish* on *GPU* hardware. We will focus on usage outside of a laboratory on commonly available hardware. *OpenCL* will be our main tool to avoid vendor lock-in.

2 Motivation

A lot of processing power is spent for cryptographic operations in a typical server workload. Processing power that may be needed for generating the content. Special instructions for many popular cryptographic algorithms are commonly available in modern *CPU*s [10]. These instructions offer tremendous speed-up but it still costs cycles to encrypt and decrypt. In a typical headless server environment the *GPU* is either idle or completely absent. In this thesis we explore the idea to use this powerful idle hardware to lower the *CPU* load and possibly speed-up encryption and decryption. Modern *GPU*s are cheap, relatively energy efficient and readily available due to high consumer demand fueled by modern computer games [22].

While there are a few projects implementing one or a few specific cryptographic algorithms in either *CUDA* or *OpenCL*, we could find none that provide a stable *API* and is suitable for production. From our research none of those projects reached a mature state where there are stable maintained releases with packages available in common *Linux* distributions. Many were extremely bleeding edge and optimized for one particular device and use-case, focused solely on achieving the best performance. While this project does not aim to be the best performing or use the lowest amount of memory, it is designed to be portable, readable and usable.

Another motivation is to create a test bed to implement more cryptographic algorithms in *OpenCL* in the future. A lot of setup code has to be written to even safely compile an *OpenCL* kernel and transfer all the data. This project should enable developers to focus on just the *OpenCL* kernel.

3 Prior Art

There are many other similar projects. In this section we summarize some of them.

3.1 AES Encryption in GPU Gems 3

The first prior art we looked at was this chapter from a well known book about various *GPU* techniques from 2007 [31]. It was released roughly at the time when integer processing began to be available on consumer *GPU*s. Instead of using *CUDA* or *OpenCL* it uses plain *OpenGL* low-level shaders because *CUDA* was only in its infancy at the time. The researchers use new features in *NVIDIA GeForce 8* like *Transform Feedback Mode* and *Typed Registers*. *AES* key schedule is done on the *CPU*. *AES* processing is implemented as a vertex shader and as a fragment shader. Both approaches are compared with fragment shader decisively winning in performance.

For large buffers the researchers achieved throughput of *53 MB/s* with vertex shader and *95 MB/s* with fragment shader.

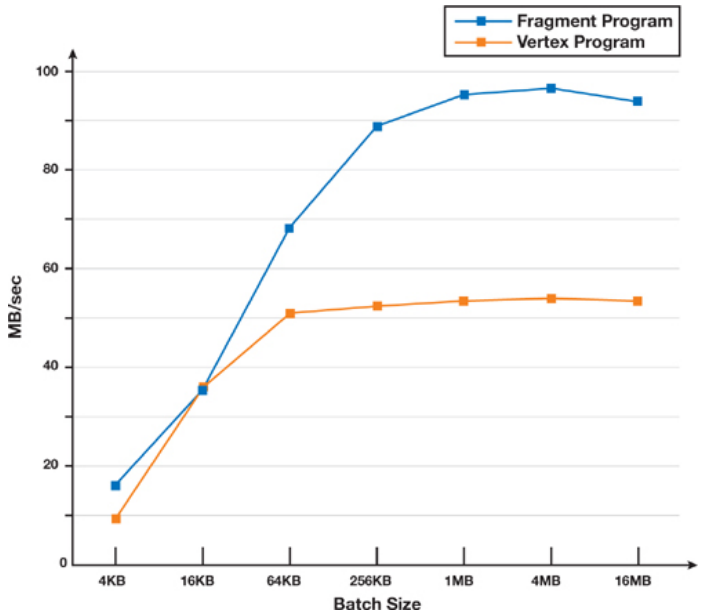

Figure 3.1: *GPU Gems 3 AES* throughput comparison

3.2 SSLShader

Keon Jang et al. [29] explore *SSL* protocol acceleration on commodity *GPU*s. The paper presents a hypothesis that lack of cheap *SSL* hardware accelerators may be preventing wide *SSL* adoption. The researchers present *SSLShader* — a transparent *SSL* proxy. The authors show that latencies increase when using the *GPU* but the throughput is high enough for the solution to be useful. In conclusion the paper states that common *GPU*s are a viable alternative to specialized *SSL* accelerator hardware and may be the driving force for wider *SSL* adoption in the future.

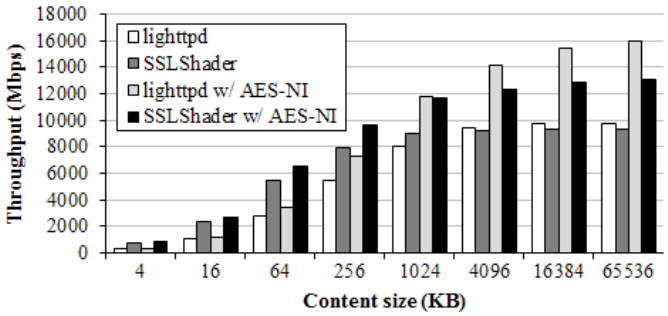

Figure 3.2: *SSLShader* reported throughput

Unfortunately, there is no way to verify claims of the researchers. The software has not been publicly released, the researchers claim on the website [21] that they are in the process of commercializing it.

3.3 Acceleration of AES Encryption on CUDA GPU

Keisuke Iwai et al. [28] provide a concise introduction to *CUDA* hardware and then explore *AES* acceleration on *NVIDIA GTX 285*. Performance of different memory arrangements are compared, as well as different granularities — number of threads operating on one *AES* block. The final throughput is *4400 MB/s* measured with *256 MB* plain-text size. The best performing arrangement operated on one *AES* block per thread and used shared memory for *T-box* allocation.

While those numbers are amazing, we have to keep in mind that the researchers did not factor in memory transfers and other setup costs. Overlapping

data transfers is mentioned as a possible way to avoid most of memory transfer costs.

In conclusion the researchers write that *CUDA GPU*s in *PC*s and even laptops seem to be suitable cryptographic accelerators. Unfortunately, the source code is not provided.

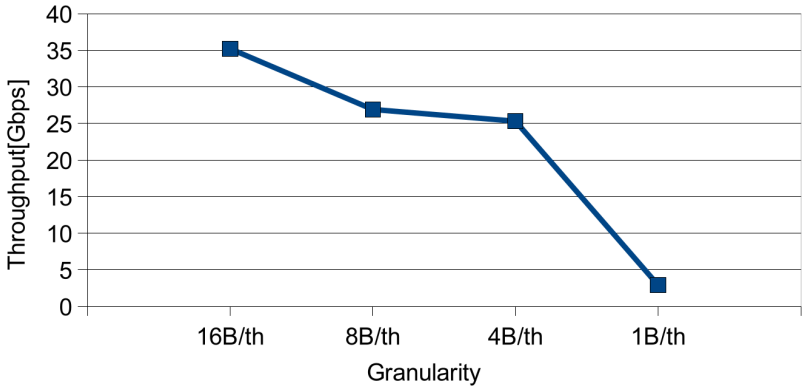

Figure 3.3: AES throughput comparison depending on granularity

3.4 Bulk Encryption on GPUs

In this article from 2011, Salman Ul Haq et al. [33] focus on encryption of big buffers during transmission or storage. Therefore big latencies are tolerated in favor of large throughput. The use-case relates to the usage of specialized *cryptographic accelerator* hardware and authors suggest that modern *GPU*s may be a potential replacement. *ECB* and *CTR AES* modes are implemented in *OpenCL* as part of the paper.

Authors do key expansion in *local memory* using the *GPU*, which is a very interesting and novel approach. Unfortunately no performance comparison between *CPU* key schedule and *GPU* key schedule is available.

The final solution reportedly has throughput of approximately *4000 MB/s* on *ATI Radeon HD 5870*.

4 Design

In this chapter we outline key design decisions before implementation starts. We also briefly discuss *GPU Compute* and *symmetric cryptography* basics.

4.1 Readability and Portability versus Performance

Performance tuning is device-specific, especially with massively parallel devices like *GPU*s. The goal of this project is not to achieve the best performance for any particular hardware, but provide decent performance on many different hardware configurations. That means that readability, portability and correctness are favored over small performance improvements. Device or platform specific performance optimization should be enabled or disabled using macros. Readability of the code is favored, functions are preferred over macros and other preprocessor tricks, longer variable names are preferred over single letter names.

The ideal result library will not require any performance considerations from the user but test and tune everything by itself. Auto-learning performance-sensitive variables — local work size, block size, etc. — is preferred over asking the user to set them manually.

4.2 Choosing GPU Compute API

Before starting any design we needed to choose a *GPGPU API*. Prior art mentioned in Chapter 3 used mostly *CUDA* or *OpenCL*. We set the following critical requirements for the *API*:

- cross-platform
- vendor-neutral
- patent unencumbered or at least royalty-free

This ruled out *DirectCompute* by *Microsoft* because it is only available on *Windows* [6]. Three major *API*s for doing general purpose computations on the *GPU* remained, which we summarize in the rest of this section.

4.2.1 CUDA

CUDA from 2007 is the oldest *API* of the three [15]. It is *NVIDIA*-only and focused on *NVIDIA* hardware architecture. It is very powerful, mature, cross-platform and widely used by the industry. Many high-level libraries are offered for *CUDA*, even commercial ones. Debugger and profiler tools offer high productivity and are stable on *Linux* from the author's limited testing.

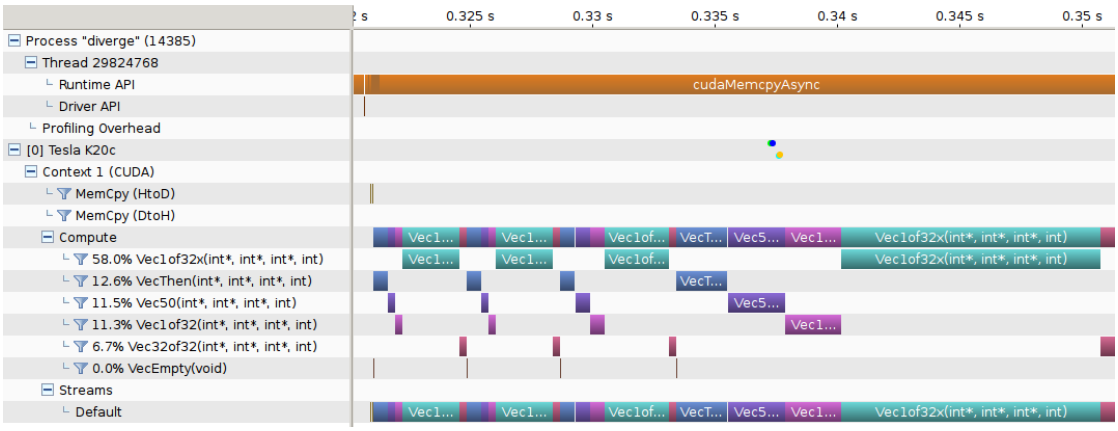

Figure 4.1: CUDA nvprof GUI [17]

It is quite well researched and there is a high amount of resources available for it on *NVIDIA* website [15] and elsewhere. Unfortunately it is as vendor-locked as can be. There is only one implementation and it does not seem this will change in the future.

4.2.2 OpenGL Compute Shaders

> Recent graphics hardware has become extremely powerful and a strong desire to harness this power for work (both graphics and non-graphics) that does not fit the traditional graphics pipeline well has emerged. To address this, this extension adds a new single-stage program type known as a compute program. This program may contain one or more compute shaders which may be launched in a manner that is essentially stateless. This allows arbitrary workloads to be sent to the graphics hardware with minimal disturbance to the GL state machine.

Figure 4.2: Section from ARB_compute_shader specification [18]

The most bleeding edge of all the considered options. ARB_compute_shader has been introduced by *AMD* in 2012 [18]. It uses the familiar *GLSL* syntax. Unfortunately it requires an *OpenGL* context which may be impractical in a headless server scenario[1]. That *OpenGL* context has to support *OpenGL* version 4.2 which is too recent for practical purposes, it would lock us out of a lot of commonly available hardware.

We could not find any profiler or debugger tools specifically for *compute shaders* but *GLSL* tools usually can be used.

4.2.3 OpenCL

OpenCL was initially developed by *Apple Inc.* with contributions from *AMD*, *IBM*, *Intel*, *NVIDIA* and *Qualcomm*. The specification was later submitted to the *Khronos Group* which ratified and publicly released it on December 8, 2008 [12]. It is an open, royalty-free standard by the *Khronos Group*. Many different companies and other members [13] participate in development of the standard.

Fundamental concepts are similar to *CUDA* with differences in nomenclature. *OpenCL* can target way more hardware than *CUDA*. Among other devices *OpenCL* can leverage *AMD*, *NVIDIA* and *Intel GPU*s. Even targeting *FPGA*s is possible with *OpenCL*. Furthermore, it is available for many operating systems — *Microsoft Windows*, *Linux*, *MacOS X* and even *Android* [12].

1. in this context, headless server is a server with no monitor, keyboard or other interfaces

The tooling seems less mature than with *CUDA* but the *API* is not locked to any hardware vendor. The single biggest downside in our opinion are the tools. We discovered *gRemedy gDEBugger* which looked promising. Then we discovered that *AMD* bought the company behind it and renamed the product to *AMD CodeXL* [2]. After downloading *CodeXL* we found that many of the features are only available with *AMD GPU*s. It is perfectly understandable but unfortunately made the tool unusable for this project because we lacked *AMD* hardware.

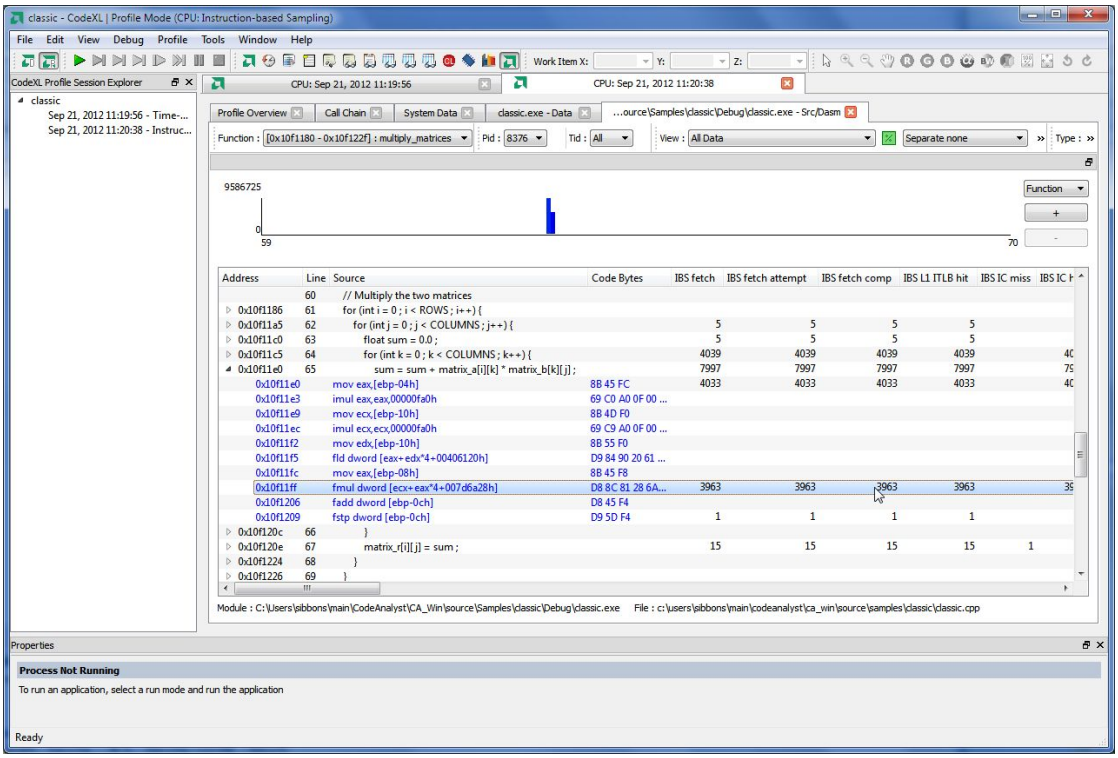

Figure 4.3: AMD CodeXL [1]

NVIDIA seems to have supported *OpenCL* profiling and debugging in the past, for example in the *CUDA Toolkit 3.1* release [7]. Unfortunately, these old releases are unsupported and cannot be used with new drivers. The profiler tool was rewritten from scratch in *CUDA Toolkit 5.0* and since then there is no official *OpenCL* support in the profiler [8]. When attempting to profile *OpenCL* code using *nvvp* or *nvprof* the tools output an error message "Warning: No

CUDA application was profiled, exiting". Our attempts to find an alternative profiler for *OpenCL* and *NVIDIA* hardware have failed.

The only profiler option for *Intel* hardware on *Linux* seems to be the *Intel VTune™ Amplifier* [11]. Unfortunately, the tool is not freely available. The basic version costs *$899*.

Despite difficulties with the tooling, *OpenCL* was chosen for this project in the end. It is not perfect but it fits all our requirements. Its future also looks promising with *NVIDIA*, *AMD, Intel*, and others all backing the standard.

4.3 Programming Language

The traditional language choice for a project like this is *C*. While the *C* standard does not define any *ABI*, the *ABI*s are defined by the platform vendors. These *ABI*s are kept stable for backward compatibility. The calling conventions are simple and well known which makes it possible to call library functions from other languages. The main drawback of *C* lies in low productivity when compared to higher level languages. Another important drawback is that *C* projects are prone to a category of security-sensitive bugs related to memory management.

The *OpenCL API* is written in *C* but it is usable from *C++* projects and bindings are available for many other languages. We briefly examined *PyOpenCL [30]* and *Ruby-OpenCL* [20]. Both bindings offer abstraction and ease of use incomparable to the *C API* — program sources can be interleaved with the main application, data transfers are convenient and safe, *OpenCL* errors are reported as exceptions. Using *Python* or *Ruby* would however prevent us from integrating with other cryptographic libraries that are mostly written in *C*. While calling *Python* or *Ruby* code from *C* is possible it requires a lot of boilerplate, performance suffers and it opens a large category of new problems we wanted to avoid.

In the end we chose *C++11* because it felt like the right compromise between *C* and a high-level scripting language. *C++11* is suitable due to its high productivity, portability and low run-time performance costs. While being a relatively new standard, compiler support is decent on all target platforms [4]. The mem-

ory model and other fundamental design features of *C++11* are similar to *C*. This makes integration with *GnuTLS*, *OpenSSL* and other libraries possible by writing a thin opaque-pointer wrapper in *C* — we can use *extern "C"* to enforce the *cdecl* calling conventions.

4.4 Target Platforms

The main goal of the library is not to be bound to any platform or hardware but in practice we can only test on several popular platform and hardware combinations. The choice of platforms and hardware was mainly driven by what we had available. It should cover most of common hardware.

List of benchmarked configurations:

- *Main Desktop*: Intel i7-920, 6 GB RAM, NVIDIA GTX 460, Fedora 21
- *Best Desktop*: Intel i5-760, 8 GB RAM, NVIDIA GTX 580, Windows 8.1
- *Laptop 1*: Intel i7-4600U, 12 GB RAM, Intel HD 4000 GPU, Fedora 21
- *Laptop 2*: Intel i7-3615QM, 16 GB RAM, NVIDIA GT 650M, Windows 8.1

All benchmark data are from the *Main Desktop* computer unless stated otherwise.

Unfortunately, we could not secure a consumer *AMD GPU* on time to test with. Since we are not using any *NVIDIA*-specific features it is likely that the project works as is with *AMD GPU*s or requires very small changes. To our best knowledge the code is portable and works on *Apple MacOS X* but we did not have the hardware to test that hypothesis.

4.5 Symmetric Cryptography Primer

This project will mainly focus on symmetric block ciphers because they are widely used and can usually be parallelized. These ciphers transform plain-text blocks into cipher-text blocks. Let us look at how the ciphers are defined.

$$E_K(P) := \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$$

$$D_K(C) := \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$$

E_K is the encryption mapping, D_K is the decryption mapping,
K is the key, P represents the plain-text, C represents the cipher-text,
k is the key-size, n is the plain-text and cipher-text size.

Figure 4.4: Formal definition of a block cipher

Observe that the size of plain-text and cipher-text matches when using symmetric block ciphers. For each possible key K, E_K and D_K are permutations — bijective mappings. The following has to hold for the cipher to be useful: $D_K(E_K(P)) = P$.

Symmetric block ciphers can operate in various modes. The most basic one is *ECB*. In *ECB*, each block is encrypted separately. This is very easy to implement and relatively easy to debug. If two aligned 128bit blocks in plain-text are exactly the same, the cipher-text blocks will also be the same. Therefore, 128bit aligned patterns in plain-text are visible in the cipher-text.

Figure 4.5: AES-ECB reveals patterns [31]

CTR is a big upgrade over *ECB* security-wise. Instead of encrypting blocks, a counter is encrypted. The counter is different for every block. Furthermore, the counter starts from an initial value that is recommended to be different for every operation. The resulting encrypted counter is used to *XOR* the plain-text block. Two equal 128bit aligned plain-text blocks are highly unlikely to result in exact same cipher-text blocks. Two equal plain-texts encrypted with different

initial counters cannot be recognized as the same from the cipher-text. The *CTR* mode is widely used in practice.

GCM is very similar to *CTR* but adds authentication. This makes it harder for the attacker to forge cipher-text blocks without being recognized. *GCM* is widely adopted because of its performance and efficiency. It is used in *TLS 1.2*, *IPSec*, *IEEE 802.11ad* and others. It is also part of *NSA Suite B Cryptography*.

Another important mode worth mentioning is *CBC*. *CBC* hides patterns in plain-text quite well and is widely used in practice. When encrypting, every block depends on a result from the previous block. This makes *CBC* encryption hard to parallelize and unsuitable for this project.

4.5.1 AES

Originally called the *Rijndael algorithm,* it is a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, or 256 bits. Number of rounds depends on key size and can be 11, 13 or 15. Each round consists of four transformations: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. The first and final rounds differ slightly from the other rounds.

It is the winning cipher of the *AES* contest [25], replacing *DES* as the encryption standard in the *US* and around the world. It has a relatively fast key schedule and utilizes a key-independent *SBox*. To this date no practical cryptoanalysis techniques have been found that can break *AES*. *AES* is widely used in security protocols. It is perhaps the most popular and most researched symmetric block cipher in the world. We have therefore made it the main focus of this project.

4.5.2 Blowfish

This is a symmetric-key block cipher from 1993 by Bruce Schneier [34]. It is much older than *AES*. Compared to *AES*, *Blowfish* has a smaller block size of 64bits instead of 128bits. Key size can vary from 32bits to 448bits. Important distinction from other ciphers is a much slower key schedule. The cipher is therefore suitable for high throughput where the key is not changed very often.

The cipher is built around the *Blowfish Feistel* function, which uses four

separate *SBoxes*. The *SBoxes* are key-dependent and have to be regenerated whenever the key is changed.

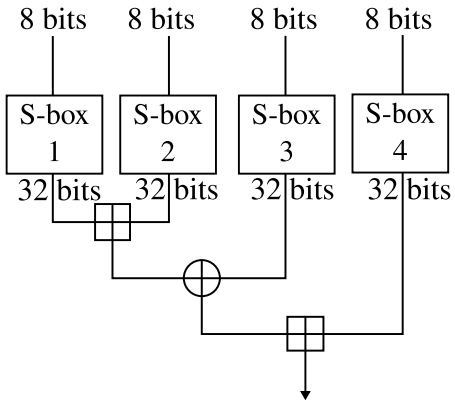

Figure 4.6: Blowfish Feistel function [3]

4.6 Amdahl's Law

An important theorem that allows us to argue about asymptotic speed-up when parallelizing algorithms. Ironically, it was written to argue for the validity of using a single processor for a large-scale computations [27]. We will frequently use the following corollary:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

N is the number of processors,
P is portion of the problem that is parallel,
$S(N)$ is the speed-up.

Figure 4.7: Corollary of Amdahl's law

In essence we reason about theoretical speed-up after adding an arbitrary number of additional processors. Since our use-case usually involves *GPU*s with hundreds of processors it is very important for the speed-up to trail off as late as possible. An important outcome of this corollary is that we need to parallelize as much of the problem as we can.

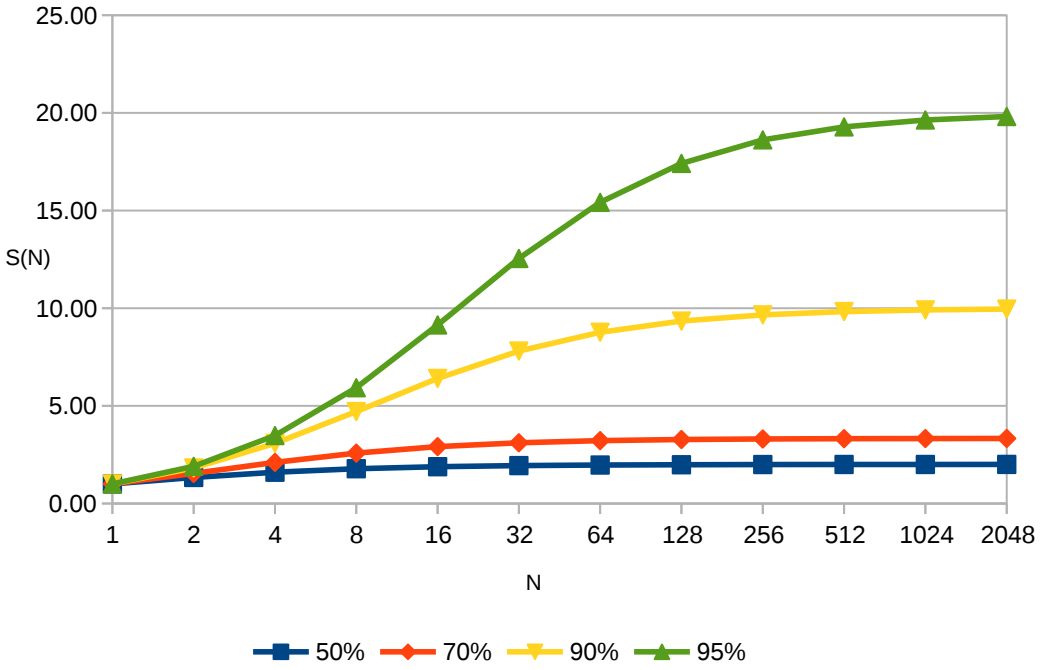

Figure 4.8: Speed-up with different P values

In the context of this project the parallel part is the *OpenCL* kernel itself. The serial part is the initialization, key schedule, kernel compilation, memory transfers and de-initialization. Initialization, kernel compilation and de-initialization are fixed costs give or take. The memory transfer takes longer the more memory we need to move to the *OpenCL* device but also gets more efficient the more memory we are moving. Key schedule varies depending on the algorithm but is usually either fixed or depends on the key size.

We therefore observe that we get better P values and therefore better speed-up for larger amounts of data.

4.7 GPU Compute Basics

OpenCL is the compute *API* we used for this project. In *OpenCL*, one or more compute devices are exposed as *OpenCL* devices. These may be *CPU*s, *GPU*s, *FPGA*s or others. Each devices has a queue that is used to schedule kernel executions.

The compute task we want to solve has to be decomposed into work-items. These work-items form a work-group. The kernel is set up to solve this work-group when it is executed. For many common tasks a loop can easily be transformed into an *OpenCL* kernel as long as the computations are not interdependent.

```
// C99
void serial_mul(
  int n,
  const float* a, const float* b,
  float* c)
{
  for (int i = 0; i < n; ++i)
    c[i] = a[i] * b[i];
}

// OpenCL
__kernel void opencl_mul(
  __global float* a, __global float* b,
  __global float* c)
{
  const int i = get_global_id(0);
  c[i] =  a[i] * b[i];
}
```

Figure 4.9: *C99* serial code compared to *OpenCL* kernel code

After writing the *OpenCL* kernel source code we use the *API* to build it and execute it on an *OpenCL* device we choose. The model is similar to *OpenGL* and *GLSL*, the driver supplied by the vendor gets the high-level source code and builds it into device specific low-level code.

5 Implementation

In this chapter we will focus on how we implemented the library to perform encryption and decryption on a set of cryptographic algorithms to pass test vectors. The resulting library is called *oclcrypto*.

Only very high-level optimization is done in this chapter.

5.1 OpenCL Abstraction

The first step was to write a very thin abstraction over the *OpenCL C API* that would allow us to execute kernels and transfer data with ease. While *OpenCL* provides a *C++ API* we did not believe it was suitable for this project. The reason is that it is merely a wrapper around the *C API* and provides no additional functionality. Something that would query the system for available devices, compile the kernels on demand and continuously check for *OpenCL* errors was necessary.

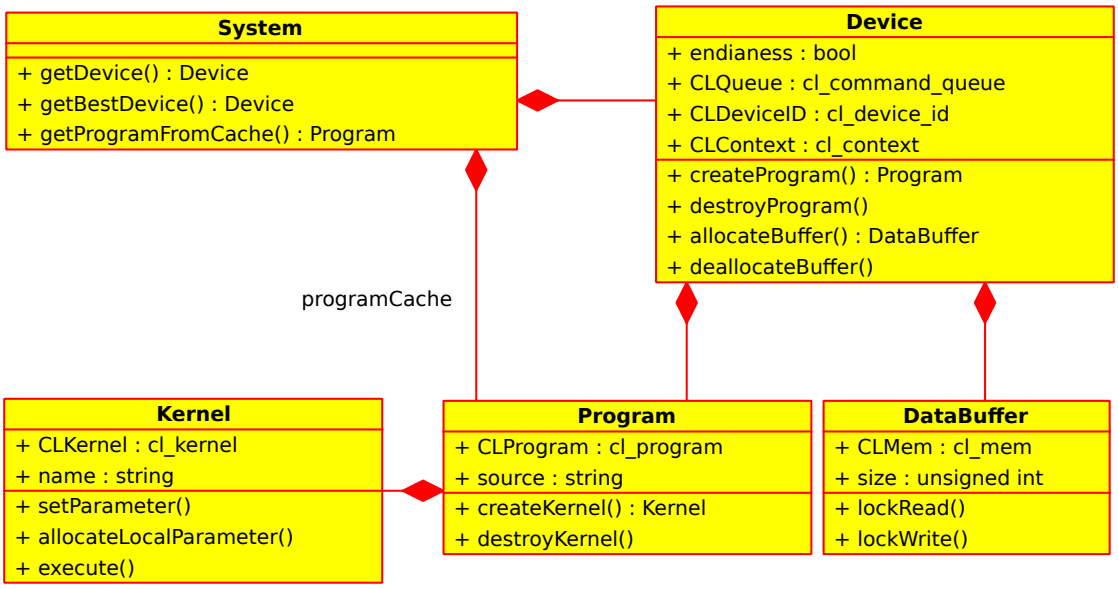

Figure 5.1: OpenCL abstraction overview

The abstraction layer had to be thin enough to achieve good performance but high-level enough to make *OpenCL* easy to use and safe. We did not focus

on providing all features available in *OpenCL*. Rather, we made the abstraction powerful just enough to serve our needs.

oclcrypto::System

The central part of the library is the *oclcrypto::System* class. When constructed it queries the system for available devices and stores them in a map for later use. It serves as a hub that keeps pointers to available resources and their usage. It also allows its users to cache programs per device, this prevents multiple compilations of the same *OpenCL* source code.

While it can be used as a singleton, it is not a true singleton. Creating multiple instances is designed to work fine and there is no static method to get the last instance.

oclcrypto::Device

Represents one *OpenCL* device. This may be a *GPU*, *FPGA*, or even a *CPU*.

In the *oclcrypto API Device* owns *Program*s that have been compiled on it. In turn, each *Program* owns *Kernel*s which represent global functions in it. *Device* itself does not do anything to avoid recompiling the same programs all the time, this is a responsibility of the *oclcrypto::System* class. *Device*s also allocate and deallocate buffers — represented by *oclcrypto::DataBuffer*.

Different *OpenCL* devices cannot share memory or compiled kernels. That is why *OpenCL* does not allow the same kernel and data to be run on multiple devices. Exactly one device always needs to be chosen to do the processing.

Each device has a command queue. To execute an action on it an action definition has to be placed in the queue. The queue can be processed either serially or out-of-order. For our use-case we decided that the hassle of out-of-order queue execution was not worth it. It may be something to research in the future.

oclcrypto::DataBuffer

Exposes global or constant memory data buffer on an *OpenCL* device to the user. Buffers can be shared between different programs and kernels but not between different *OpenCL* devices.

Reading and writing is achieved via a class template called *DataBufferReadLock* or *DataBufferWriteLock* respectively. These templates implement a common *C++* design paradigm called *RAII*. They automatically lock the buffer and are designed to unlock it when they go out of scope.

oclcrypto::Program

One compiled *OpenCL* source file. Has one or more kernels. Kernels are the only entry point to *OpenCL* programs. They take constants or *DataBuffer*s as input. Let us take the *AES* program as an example. It contains definitions of the *SBoxes* and various helper functions. Then it has the *ECB* encryption and decryption kernels, *CTR* encryption kernel and *GCM* encryption kernel. That is four kernels in total in one *OpenCL* program.

Transferring input data, executing a kernel and transferring output data back are all actions in the command queue of an *OpenCL* device.

Compiled programs cannot be transferred between devices. One *OpenCL* source file has to be compiled multiple times if user wants to execute the code on multiple devices.

5.2 GPU Architecture

Before we start implementing the algorithms we need to research how a typical *GPU* works. That way we can avoid anti-patterns in *OpenCL* kernel design.

The main goal of a *GPU* is to generate graphics content. In the past this was mainly texture mapping, image processing, polygon rasterization and geometry transformations. This and even other problems of computer graphics involve a lot of data being processed with the same function. For this reason *GPU*s are designed to be capable of processing a lot of data in parallel.

Let us now look at how a *GPU* is designed. When compared to a *CPU* a *GPU* usually has many more processors but each one of them is less powerful than a typical *CPU* processor. The *GPU* processors tend to have relatively low clock-speeds and usually do not have dedicated memory or instruction decoder. They are less independent than a *CPU* processor. On most *GPU* architectures

a lot of the processors are executing the same instruction with different data. This paradigm is called *SIMT*. This is not very flexible but works well with typical *GPU* workloads. Fewer instruction decoders make the *GPU*s less complex, cheaper to produce and make thread scheduling simpler. At the same time it causes surprising issues when optimizing for performance. See Section 5.2.1 for more details.

In the remainder of this section we will focus on *NVIDIA Fermi* [16] architecture, others may differ a bit but also share a lot of traits. The *NVIDIA GTX 460* that we are doing most of testing on is one of the *Fermi* devices.

A *Fermi GPU* consists of several *streaming multiprocessors*. Each *streaming multiprocessor* consists of 32 *CUDA* cores. Each core has one *ALU* and one *FPU*. Processing happens in groups of 32 threads that are called *warps* in the *CUDA* terminology. If processing in less than 32 threads is scheduled the scheduler spins up 32 threads and some of them are executing instructions on dummy data. Dummy data is filtered and thrown away when processing finishes.

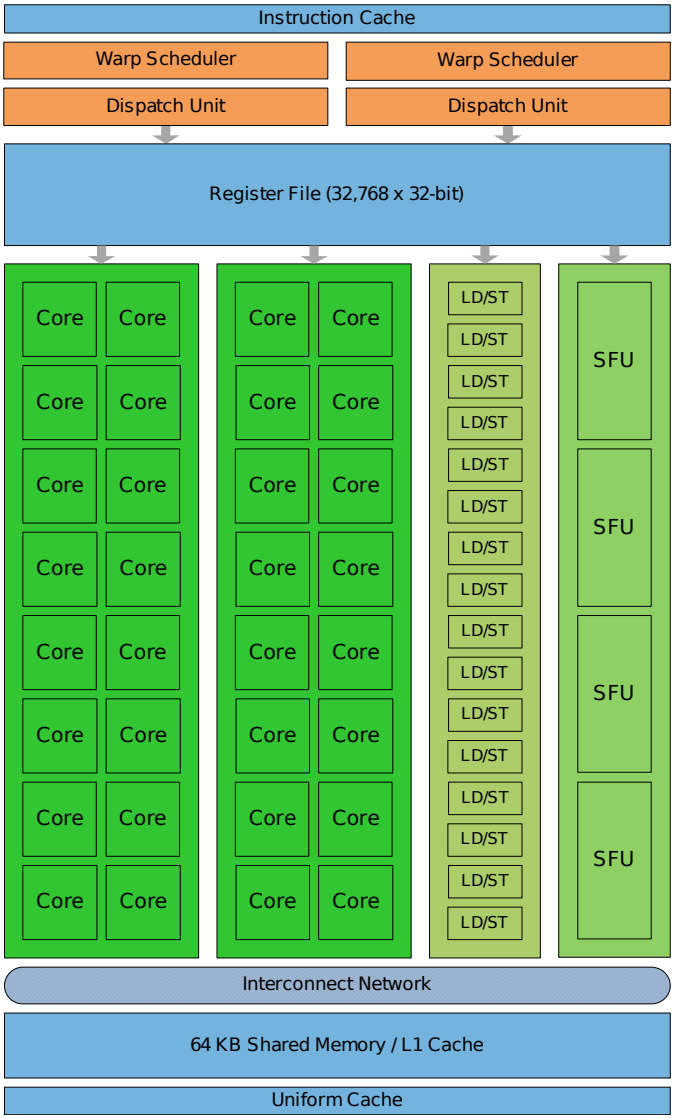

Figure 5.2: High-level diagram of NVIDIA Fermi [16]

5.2.1 Specifics of GPU Threads

GPU threads are very lightweight when compared to *CPU* threads but they are also less flexible. On a *CPU* each thread may do completely different instructions. On a GPU any instruction divergence can slow the entire execution down a lot. In the example in Figure 5.3 all of the threads will process the floating point add section but only one thread will actually use the data. The data from

threads that do not have local id 0 will be thrown away. That is 31 threads out of 32 that are doing extra processing for no reason. This means that many early out optimization techniques do not apply on *GPU*s. In fact, early out optimization can slow the kernels down in some cases because the executions diverge.

```
1  float a = 0;
2  float b = 0;
3  float c = 0;
4
5  // ... processing
6
7  if (get_local_id(0) == 0)
8      a = b + c;
9
10 // ... more processing
```

Figure 5.3: Example of divergence

There has to be instruction convergence in every thread of every warp. In case of a divergence some of the threads are kept idling until re-convergence.

5.2.2 Memory Access

Memory has to be read and written in specific patterns to avoid hitting *slow paths*. The common ideal pattern is that a group of 32 threads read consecutive memory where each thread reads 1/32 of it, the first thread reading the first 1/32 of the memory, the second reading the next, ... [16]. Even if we do read sequentially we still have to make sure the memory is aligned. Misaligned access causes the warp to read additional memory that is not used. In case of *AES* this is a non-issue, the blocks are 128bit which is aligned on all of our target hardware.

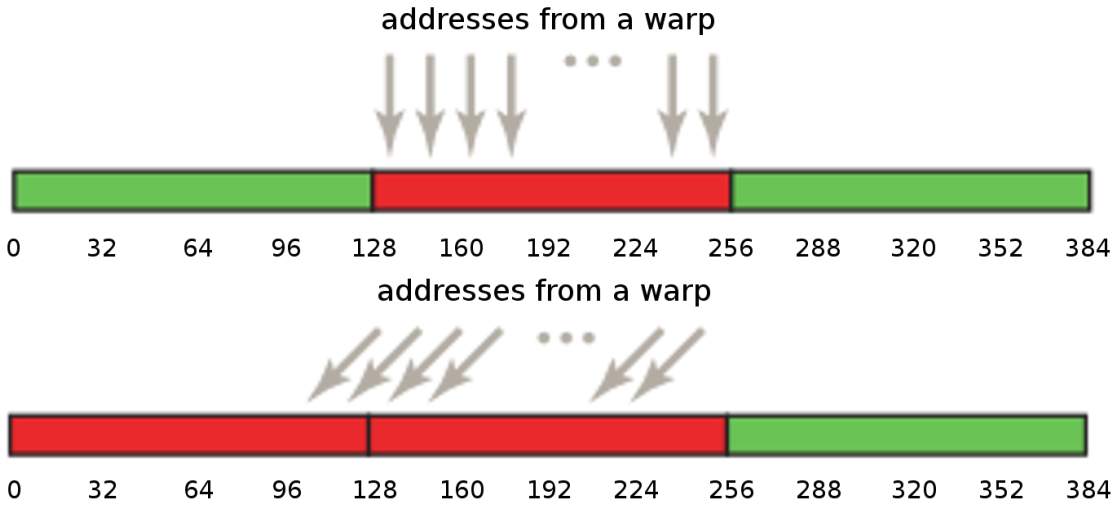

Figure 5.4: Aligned vs. unaligned memory access [16]

Local memory on the *GPU* is accessed via *memory banks*. Accessing the same *memory bank* from multiple threads results in a *bank conflict*. When a *bank conflict* occurs the accesses have to be serialized which slows down performance. The amount of memory banks differs between *GPU*s, *NVIDIA GTX 460* has 32 memory banks.

5.3 Benchmark Suite

To measure performance we implemented a benchmark suite. This suite includes synthetic tests that measure throughput of all the implemented algorithms. Memory transfers from host to device are included in the measurements unless stated otherwise. Tests are run multiple times and averaged to improve measurement precision.

Timing methods used in other publications greatly vary, most publications focusing on performance do not factor in memory transfer and other setup costs. All benchmark results in this project factor in all setup costs unless stated otherwise. In practical applications users usually want processed data in host device *RAM* so we consider measuring with memory transfers closer to real-world usage. For some use-cases users may be able to hide memory transfers performance cost completely by doing it in parallel of encryption or decryption

of another independent data-set [33].

There are a few unexpected issues that result in useless benchmark times. Many *GPU*s change their clock-speeds depending on their load. This is done to lower heat dissipation and energy consumption. When we first start processing, the *GPU* runs at a low clock-speed. After less than a second it increases the clock-speed to a middle level and a short while after that it runs at the highest, full design clock-speed. This makes the first benchmarked task seem slower than it actually is. To counteract this effect the *GPU*s are warmed up first — a dummy task is executed on them. Immediately after the dummy task is finished we run the real benchmark. Instead of having to code a dummy task we simply ran the benchmarks twice in a row to ensure the *GPU* runs at full design clock-speeds. The *nvidia-settings* utility was used to verify the clock-speeds.

5.4 Memory Transfer Between Host and OpenCL Device

A typical use-case involves copying data from host *RAM* to device memory, executing the kernel, and then copying results back to host *RAM* for further use. Optimizing for fast memory transfers is clearly important. The theoretical limit on 16-lane *PCI-E 2.0* is *8 GB/s* in both directions [35]. Design limit on *DDR3-1333* is 10.66 GB/s. *NVIDIA GTX 460* — the main GPU we are testing with — has *GDDR5*. The design speed of *GDDR5* is higher than of *DDR3-1333* so we do not need to consider it as a bottleneck.

The initial implementation used generic memory transfers. After the initial memory transfer code has been written, these were the numbers we got from our benchmark.

	64 kB	256 kB	1 MB	4 MB
GPU to CPU (average MB/s)	31.321	32.970	33.229	33.186
CPU to GPU (average MB/s)	37.656	39.911	40.137	39.792

Table 5.1: Initial implementation memory transfer performance

It turns out that memory transfer was the biggest overall bottleneck after the initial implementation was completed. The first step for improvement is very trivial — remove out of bounds safeties for release builds. This saves a lot

of cycles and enables the compiler to aggressively optimize. Reads and writes of locked data buffers get optimized to *memcpy* calls.

	64 kB	256 kB	1 MB	4 MB
GPU to CPU (average MB/s)	291.582	318.691	341.806	339.364
CPU to GPU (average MB/s)	239.343	262.314	275.879	271.146

Table 5.2: Memory transfer performance with safety checking off and full optimization

DataBuffer transfers were no longer the main bottleneck but we decided to push the performance a little bit further. The last optimization came from moving from generic memory buffers to *pinned memory*. In our tests we discovered that it is worth it for all but the smallest buffer sizes. When using *pinned memory* in *oclcrypto* we let the *OpenCL* driver from the vendor allocate shadow memory for device buffer and we tell it to map and unmap it when we want to access it. That means that the *OpenCL driver* can allocate the memory in whichever way is most performant on that architecture. The speed-up over generic buffers is quite incredible, especially when reading back big buffer sizes.

	64 kB	256 kB	1 MB	4 MB
GPU to CPU (average MB/s)	65.874	449.561	810.093	822.656
CPU to GPU (average MB/s)	59.671	234.991	306.883	312.932

Table 5.3: Pinned memory transfer performance

This performance was still not close to the theoretical limit of *8 GB/s*. At this point we decided that any other optimization for memory transfers did not have a decent cost / benefit ratio. Memory transfers stopped being the bottleneck for our use-cases.

Figure 5.5: Memory transfer speeds comparison charts

5.5 Cryptographic Algorithm Context

The goal of this library is to hide the added complexity of *GPU* computing from the user. *Cryptographic Context* classes are written to do just that. User is expected to construct them, set their parameters, execute them and collect resulting data. Besides having to read data from *GPU DataBuffers* all the *GPU* complexity is hidden. Reading from *GPU DataBuffers* is made a bit more convenient with *C++* operator overloading.

The high level cryptographic *API* is centered around *oclcrypto::System*. Users

are expected to construct at least one *System* class and reuse it for all their cryptographic needs. Each cryptographic algorithm has its own context class that needs to be setup and then executed. Users are encouraged to reuse the same context class in case they are using the same key for multiple chunks of input. Reusing the context enables the library to skip key schedule if key remains the same.

Typical life-cycle of a cryptographic context involves construction, setting a *key*, setting an *initialization vector* or *nonce*, setting the input data, executing the encryption or decryption and finally copying the resulting data back for further use. Since the key schedule usually does not depend on the processing mode, each cryptographic algorithm has a base class that does the key schedule and then a class for each supported mode.

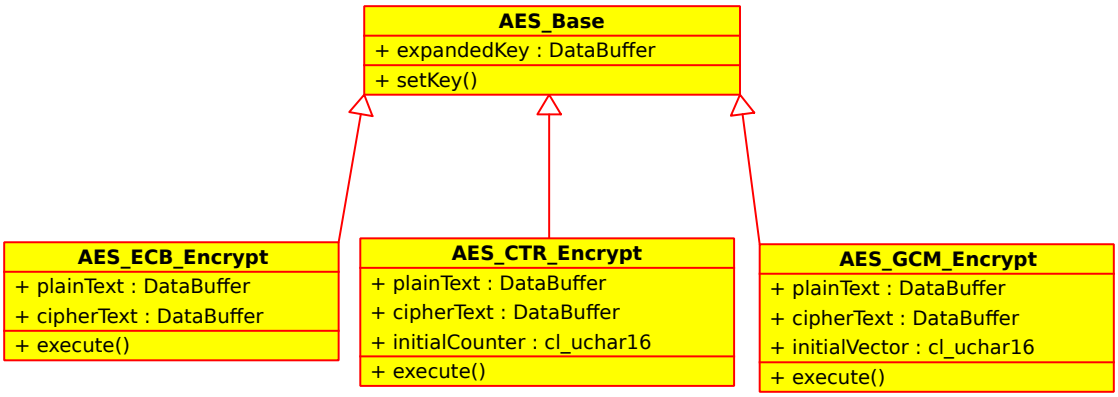

Figure 5.6: Various AES modes share the key expansion code

```
1 oclcrypto::System system;
2 // we assume user wants to use the first device available
3 oclcrypto::Device& device = system.getDevice(0);
4 oclcrypto::AES_ECB_Encrypt context(system, device);
5 context.setKey(key);
6 context.setPlainText(plaintext, plaintextSize);
7 context.execute(256); // use 256 threads
8 auto cipherText = context.getCipherText()->loadRead<unsigned char>();
```

Figure 5.7: Cryptographic context usage example

To avoid *API* usage mistakes, the encrypt and decrypt contexts have method names with plain-text and cipher-text instead of just input and output.

5.6 Regression Testing

Correctness is obviously very important in cryptographic algorithms. Since this project involves optimizing them for speed we need to make sure we do not break functionality. I have chosen *boost::test* to implement test cases for the cryptographic algorithms.

The *NIST* test vectors were added to the test suite to avoid regressions when changing the kernels. Furthermore we have added a few round trip smoke tests.

The entire test suite is executed by running *oclcrypto-test*.

```
$ ./oclcrypto-tests
Running 25 test cases...
*** No errors detected
```

Figure 5.8: Expected test suite output

As algorithms were added test vectors were added to the test suite. As a result, the project has a high test coverage.

5.7 Storage of OpenCL Kernel Source Code

Existing projects related to *OpenCL* (see Chapter 3) typically store the kernel sources in separate files. At run-time, these files are opened, their contents are read and passed to the *OpenCL API* for compilation. In our experiments this proved to be problematic. The shared object library had to know where to load the sources from. This is a source of problems that we wanted to avoid.

Instead of storing the kernel sources separately we chose to build them into the shared object. Just using *static const char* variables directly worked quite well but editing was very awkward and inefficient. Syntax highlighting was not available, special characters had to be escaped which made the code harder to

read, any refactoring required awkward indentation and quote character fixes. The solution that was used in the end involves loading separate *OpenCL* kernel source files and processing them into *C/C++* files with a single *static const char** variable. This seemed to be the best of both worlds. Files can be edited separately, yet the sources are inbuilt into the shared object. The complexity of getting file paths of the *OpenCL* kernel right are moved from the user to the person building the library.

The end solution is not perfect and it is quite easy to find test vectors that result in invalid *C/C++* code being generated. The *C/C++* code is generated at configure time instead of compile time, this can lead to unexpected issues when rebuilding the project. Since this does not affect our *OpenCL* kernel sources and was not the focus of this project we decided to ignore these issues.

5.8 OpenCL Memory Access

OpenCL has multiple types of device memory.

- global
- constant
- shared
- local
- private

The *OpenCL* memory types are related to how a *GPU* is designed, see Section 5.2. Peak throughput as well as latency varies tremendously between different memory types. But choosing the right type of device memory is not enough, there are several memory access anti-patterns in *OpenCL* and *CUDA* that may not be obvious to a *CPU* programmer. Even worse these pitfalls differ between *GPU* architectures. See Section 5.2.2 for more.

Fortunately, the one thread per *AES* block kernel naturally avoids many of these pitfalls when reading plain-text — the global data is read sequentially. We have to take extra care when reading the *AES expanded key* or *Blowfish P array*

or *SBoxes* though. The initial implementation read all these from global memory and suffered from low throughput and bank conflicts. This was improved when the data was first copied into local memory, then read. See Section 6.1 for more.

5.9 AES-ECB

AES-ECB was the first algorithm we implemented because it is a building block for both *CTR* and *GCM*. In itself *ECB* is not very useful because patterns in the cipher-text may reveal patterns in the plain-text.

We used the official *NIST FIPS 197* specification [32] to draft the first implementation. The first draft is just the *NIST* pseudo-code coded with *OpenCL* syntax and processing one *AES* block with one *OpenCL* thread. We decided against using heavily optimized *AES* because it is harder to debug and harder to analyze.

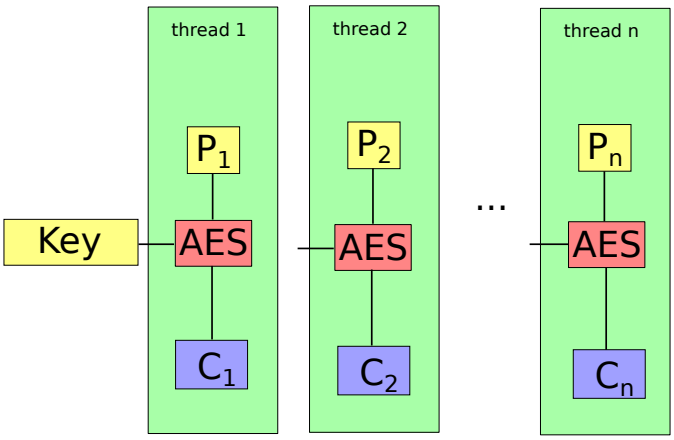

Figure 5.9: AES-ECB processing chart

Two look-up tables are required for *Sbox* and *InverseSbox* operations. Six look-up tables are required for the *Galois Field* multiplication which is used in *MixColumns* and *InverseMixColumns*. All eight tables are kept in device memory, access to the data is cached. Total memory cost is $8*256 = 2048$ bytes, which is a reasonable cost considering we can then do all the mentioned operations in

```
__kernel void AES_ECB_Encrypt(
    __global uchar16* plainText, __global uchar16* expandedKey,
    __global uchar16* cipherText,
    unsigned int rounds, unsigned int blockCount)
{
    int idx = get_global_id(0);
    if (idx < blockCount)
    {
        uchar16 state = plainText[idx];
        state = AES_AddRoundKey(state, expandedKey[0]);

        for (unsigned int i = 1; i < rounds - 1; ++i)
        {
            state = AES_SubBytes(state);
            state = AES_ShiftRows(state);
            state = AES_MixColumns(state);
            state = AES_AddRoundKey(state, expandedKey[i]);
        }

        state = AES_SubBytes(state);
        state = AES_ShiftRows(state);
        state = AES_AddRoundKey(state, expandedKey[rounds - 1]);
        cipherText[idx] = state;
    }
}
```

Figure 5.10: AES-ECB OpenCL kernel

constant time. Furthermore, we can use the same instructions every time which is necessary because of the way *GPU*s work. See Section 5.2.1 for more.

The algorithm is naturally parallelizable by block — one thread encrypts or decrypts one *AES* block. In case of *AES-ECB* decryption simply performs inverse of all the steps of encryption in reverse order.

Several straightforward optimization steps were available for the first prototype. We can use the *restrict* keyword for non-overlapping memory buffers. The *read_only* and *write_only* keywords can be used to decorate inputs and outputs. This helps the compiler optimize more aggressively. The performance numbers looked quite promising. As we can see from Figure 5.11 the speed clearly increases as plain-text sizes increase. Larger key sizes cause more *AES* rounds to be processed, so the speed decreases as *AES* key size increases.

	4 kB	16 kB	64 kB	256 kB	1 MB	4 MB
ECB-128 MB/s	2.256	8.908	34.835	79.933	123.244	128.204
ECB-192 MB/s	2.285	9.043	33.310	72.023	106.990	110.899
ECB-256 MB/s	2.277	9.087	31.633	65.513	94.524	97.673

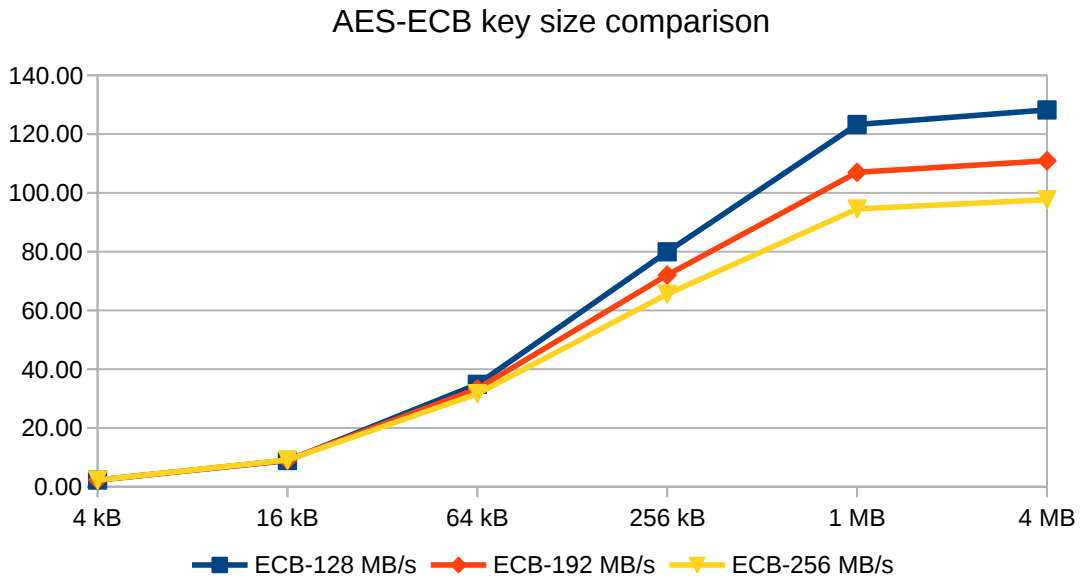

Figure 5.11: AES-ECB performance

5.10 AES-CTR

Before doing any further optimization we decided to go ahead and implement the *CTR* mode. *CTR* is not just an important mode to benchmark, it is also a practical way to encrypt and decrypt files and is therefore very suitable for our case studies.

The *AES-CTR* kernel is just a small step away from *ECB*. Instead of encrypting the plain-text, a counter is incremented and then encrypted with given key. Instead of always starting from zero the counter starts from a value called *Initial Counter* or also *Initial Vector*. The encrypted counter is then used to *XOR* the plain-text block. The advantage is that patterns in plain-text do not show in the cipher-text. Extra care has to be taken to avoid reusing initial counters as that can expose the key!

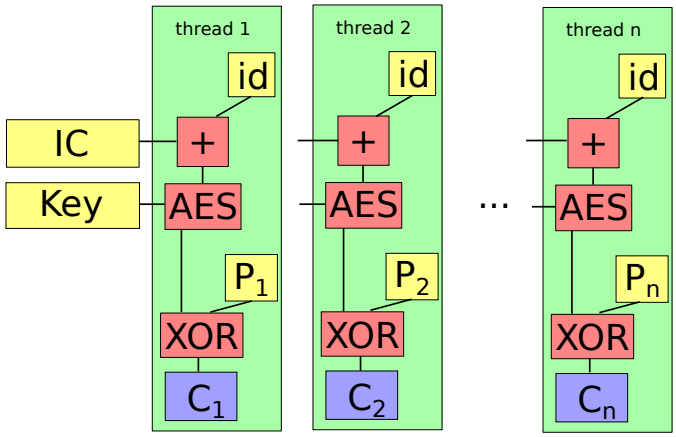

Figure 5.12: AES-CTR processing chart

Incrementing the counter in a generic way turned out to be a difficult task. The counter is a 128bit number and there is simply no widely available function in *OpenCL* that can add two 128bit numbers with proper carry and overflow. Incrementing a 128bit counter by one would be reasonably fast and easy to do but that is not enough for our use case. We cannot use any results from any of the threads as that would create dependencies and slow processing down. Instead we need a generic 128bit integer addition.

The first version we implemented looked like this:

```
1  void AES_CTR_IncrementIC(uchar16* ic, unsigned int id)
2  {
3      // TODO: This will not carry over the last half!
4      //       We need some sort of a uint4_add function.
5
6      // because of endianess we need to flip
7      unsigned long last = (unsigned long)ic->sfedcba98;
8      last += id;
9      uchar8* last_uchar8 = (uchar8*)&last;
10     // and then flip it back
11     ic->s89abcdef = last_uchar8->s76543210;
12 }
```

There are several issues with this implementation. First of all it does not

carry over the last half, as the comment says. It also assumes *endianess* of the device which goes against our requirements to be hardware neutral.

We will focus on fixing *endianess* first. If the device is *little endian* we can compile the program with *LITTLE_ENDIAN* macro defined. Using *#ifdef* in the *OpenCL* source we can choose the appropriate code path.

Fixing the carry-over proved to be much more difficult. One possible solution is to copy the last half, add to it, then check if the new version is lower than the old version. If it is we need to add the carry-over to the first half. This requires many more instructions and only affects cases where the *IC* is chosen to have high least significant bits. We have decided to ignore this issue and instead recommend using suitable *IC*s.

After the counter increment was implemented, we took code from the *ECB* kernel and used it to encrypt the counter.

	4 kB	16 kB	64 kB	256 kB	1 MB	4 MB
CTR-128 MB/s	2.263	9.072	34.971	87.186	139.519	145.386
CTR-192 MB/s	2.258	8.959	34.603	77.764	118.955	123.517
CTR-256 MB/s	2.262	9.047	32.656	70.260	103.500	107.357

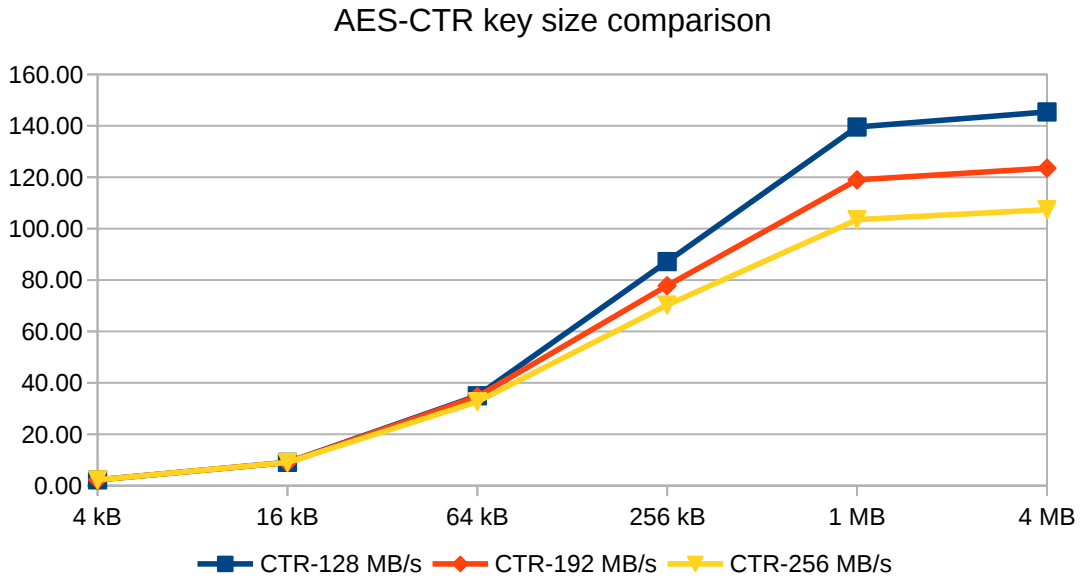

Figure 5.13: AES-CTR performance

5.11 AES-GCM

The encryption and decryption is just a small step from *AES-CTR*. We decided not to include a processing chart for *GCM* as it just a variation of the previous processing modes. A big difference between *CTR* and *GCM* counter processing is that *GCM* allows us to assume that the last 32 bits of *Initial Counter* are zeros. This helps a lot with a fast hardware implementation of counter incrementing. It also means that the counter will wrap around after *4294967295* blocks. But since that is more than *17 GB* of plain-text this is not a big issue. In practical use plain-text of that size will not be transferred at once.

The encryption and decryption are easy to implement. However we also have to do authentication as part of *AES-GCM*. And *GCM* authentication cannot be easily parallelized. The authentication tag for block *n* depends on authentication tag of block *n-1*. See figure 5.14. This creates a chain of dependencies between all the blocks and really stalls the processing. It may be possible that there is something better than serial processing — like a reduction — available but we could not find any. Since the authentication is serial it makes sense to always do it on the *CPU*. The serial part of the algorithm is too large for a big speed-up with massive parallelization — see Section 4.6 for more about *Amdahl's Law*.

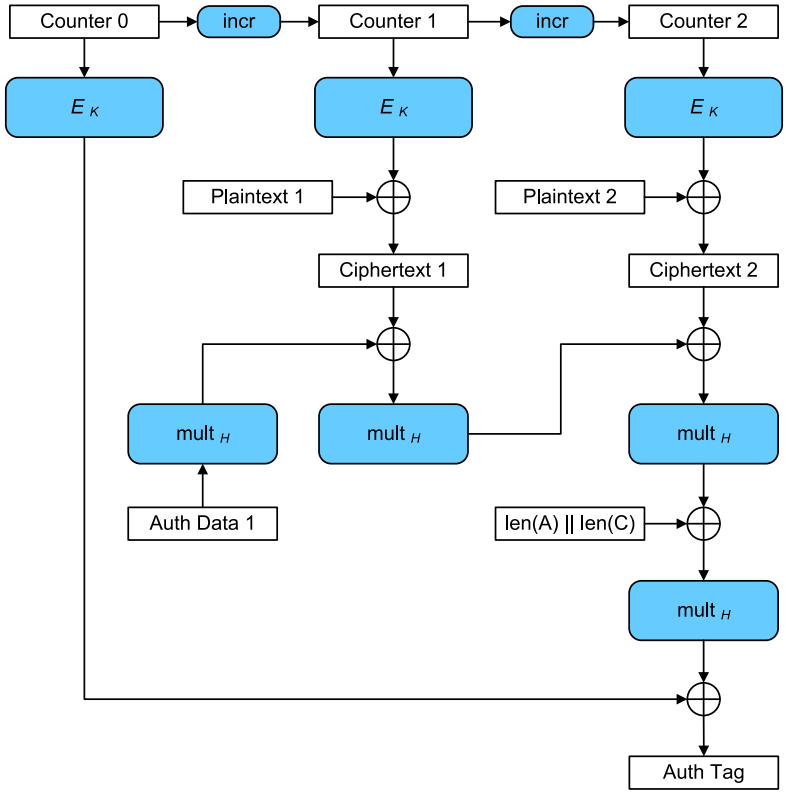

Figure 5.14: AES-GCM diagram

5.12 Performance Comparison of AES Modes

Before we measured and looked at the numbers we expected to find that all three *AES* modes have roughly the same performance. Surprisingly, the performance numbers of *AES-CTR* and *AES-GCM* are consistently better than *AES-ECB*. The difference is significant. It is very difficult to say the reason for this with absolute certainty but *AES-CTR* and *AES-GCM* are most likely more cache friendly on *NVIDIA GTX 460*. If possible we would back this up with data from *nvprof* but the tool refuses to profile *OpenCL* programs. *CTR* and *GCM* modes of *AES* are almost identical so the performance numbers are very close.

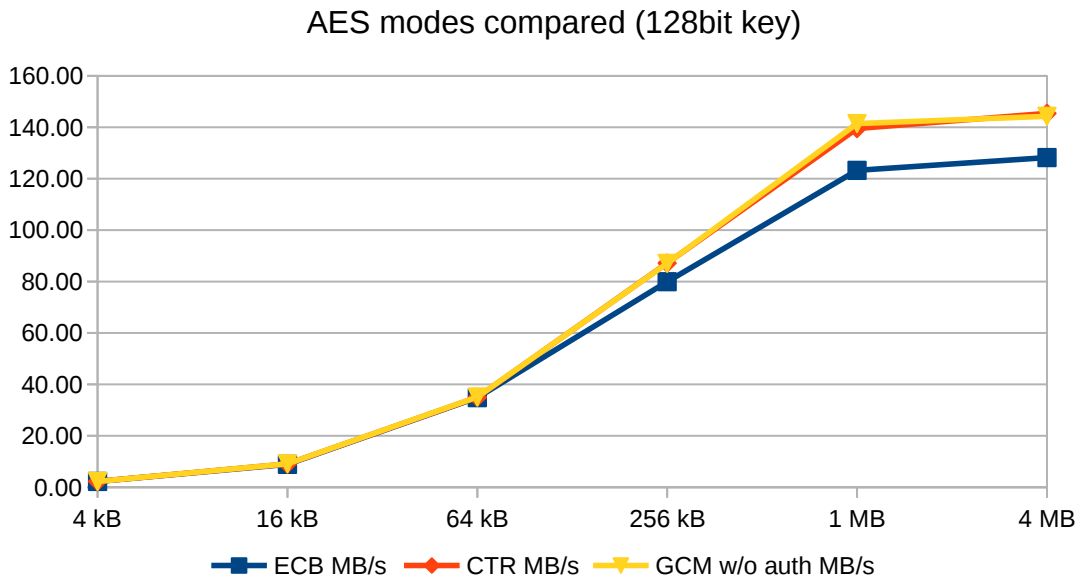

Figure 5.15: AES modes performance comparison

5.13 Blowfish-ECB

Because of the smaller block-size *Blowfish* has better occupancy for smaller plain-text sizes than *AES* — two times as many threads can be used for the same plain-text size. We therefore expect the *ECB* mode to be at least twice as fast as *AES-ECB*.

Our initial implementation stored the key-dependent *SBoxes* in global memory.

Blowfish was implemented mainly for performance comparison purposes. In practice it is rarely used compared to *AES*, the smaller block-size makes it more performant but also less secure than *AES*. For this reason we only implemented the *ECB* mode. Other modes can be added quite easily if there is demand for them. Typical *AES* key sizes were used for comparison purposes.

	4 kB	16 kB	64 kB	256 kB	1 MB	4 MB
ECB-128 MB/s	2.261	8.958	34.913	124.610	352.996	394.437
ECB-192 MB/s	2.274	8.962	34.988	124.756	352.479	394.417
ECB-256 MB/s	2.278	8.967	34.926	124.668	352.085	395.181

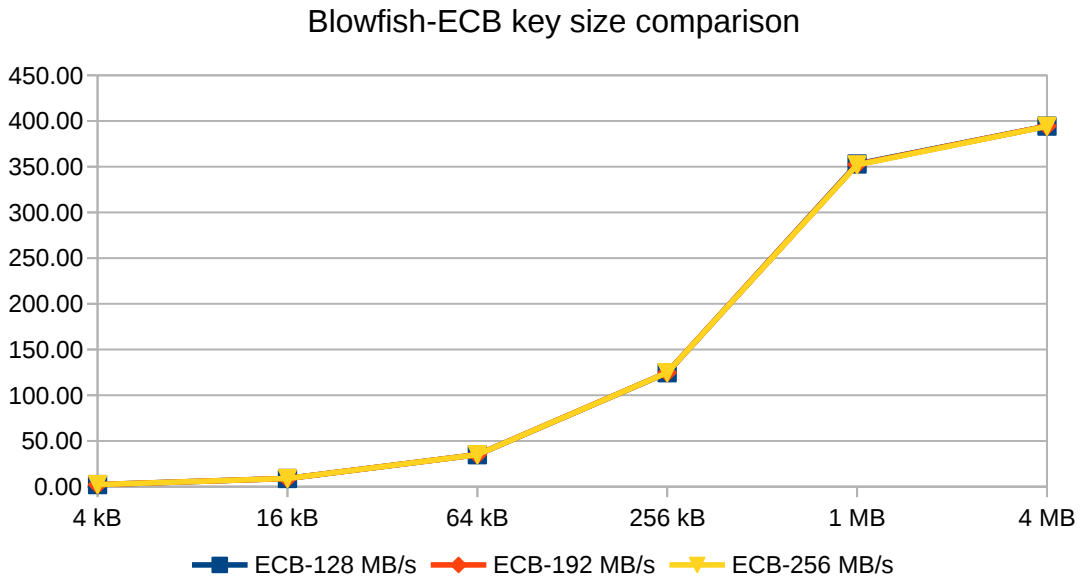

Figure 5.16: Blowfish-ECB performance

As we can see from Figure 5.16, performance is independent of key size. The reason for this is that the key size only affects the key schedule and the key schedule for *Blowfish* is defined in such a way that it takes roughly the same time regardless of key size [34]. Let us now compare the performance to our *AES* implementation.

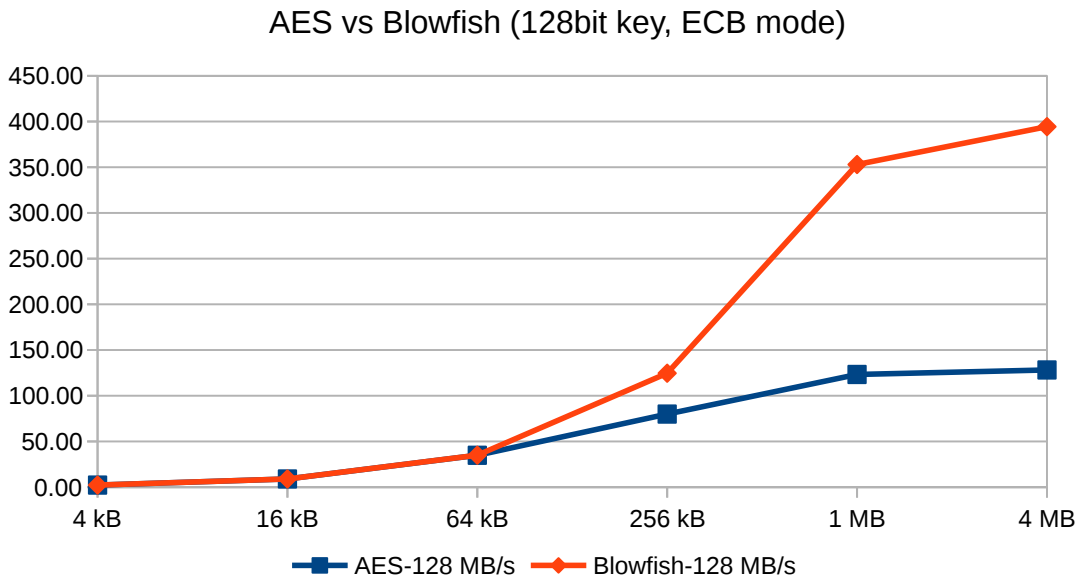

Figure 5.17: AES vs. Blowfish performance

We can see that the performance is noticeably faster than *AES*. The main reason for this is higher occupancy — twice as many of the *GPU* threads are being used for the same plain-text size — and lower amount of table look-ups while processing. Our implementation of *Blowfish* on average uses a lower amount of table look-ups than the *AES* implementation. It is also likely that *Blowfish* is more cache friendly on *GPU*.

6 Performance Evaluation

In this chapter we optimize the *OpenCL* kernels and evaluate performance on various hardware setups.

6.1 Optimizing Memory Access

After the basic implementation was completed we started to optimize. The first logical step was to optimize memory access in the kernels. The initial prototypes used registers and global memory directly. Using local memory for resources shared between threads in a warp offered potential for better performance. Local memory has much lower latencies and is much faster than global memory. See Section 5.2.2 for more. To copy global memory to local in a device-agnostic way we used *async_work_group_copy*.

```
__local uchar16 localExpandedKey[15];

event_t cacheEvent;
cacheEvent = async_work_group_copy(
    localExpandedKey,
    expandedKey,
    rounds,
    cacheEvent
);

const int global_id = get_global_id(0);
uchar16 state = plainText[global_id];
wait_group_events(1, &cacheEvent);
```

Figure 6.1: Using local memory in AES-ECB kernel

We expected dramatic speed-up when copying expanded key to *local memory* and using the local copy in all threads of a warp. The difference between using *__constant* and this solution was noticeable for small plain-text sizes but as the buffers got bigger there was no difference.

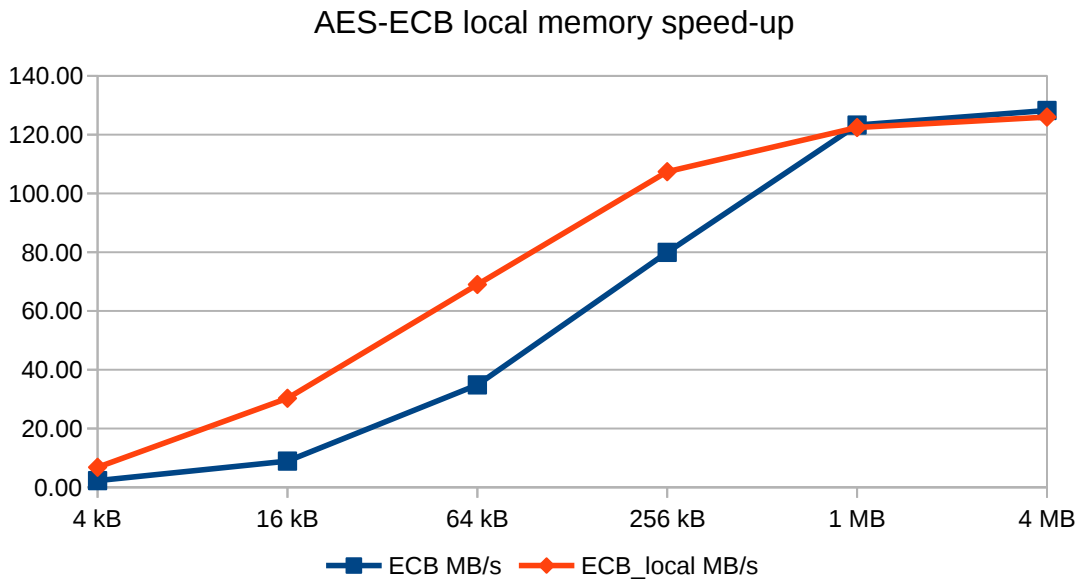

Figure 6.2: AES local memory speed-up

Unfortunately, without access to solid profiler tools we can only speculate why the difference is so small for large buffer sizes.

We also managed to get a noticeable speed-up in *Blowfish* copying the *P array* and *SBoxes* to *local memory*, again especially for smaller buffer size. We can only speculate that larger buffer sizes hide memory transfer latencies.

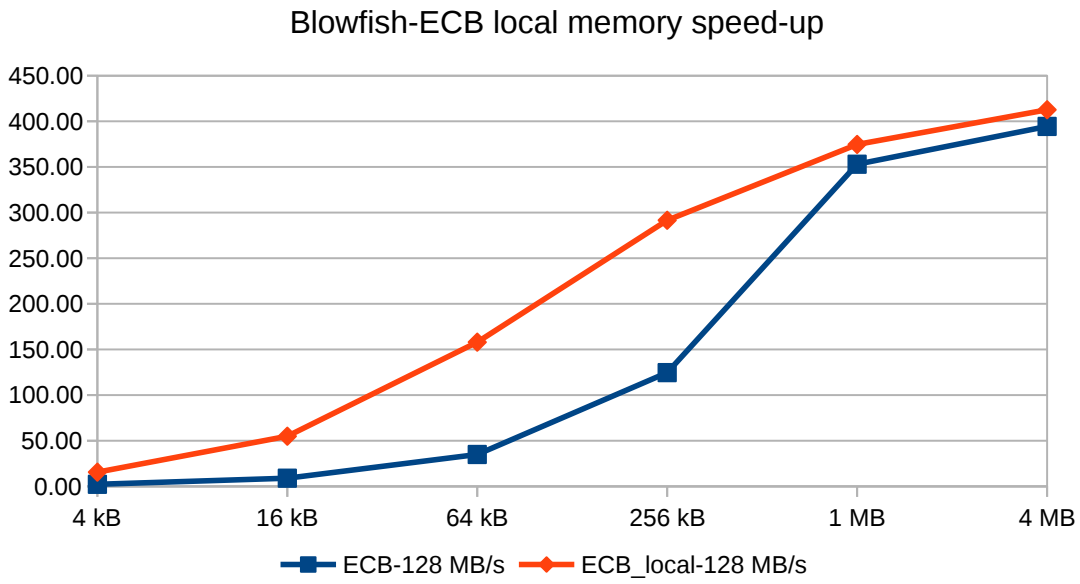

Figure 6.3: Blowfish local memory speed-up

6.2 Performance Comparison Across Hardware

All charts and data up to this point have been measured on *Intel i7 920* with *NVIDIA GTX 460* on *Fedora Linux 21*. This chapter contains comparison with other hardware on other platforms. We will only show charts in this section to save space, for raw data please see the attached *benchmark_results.ods* file.

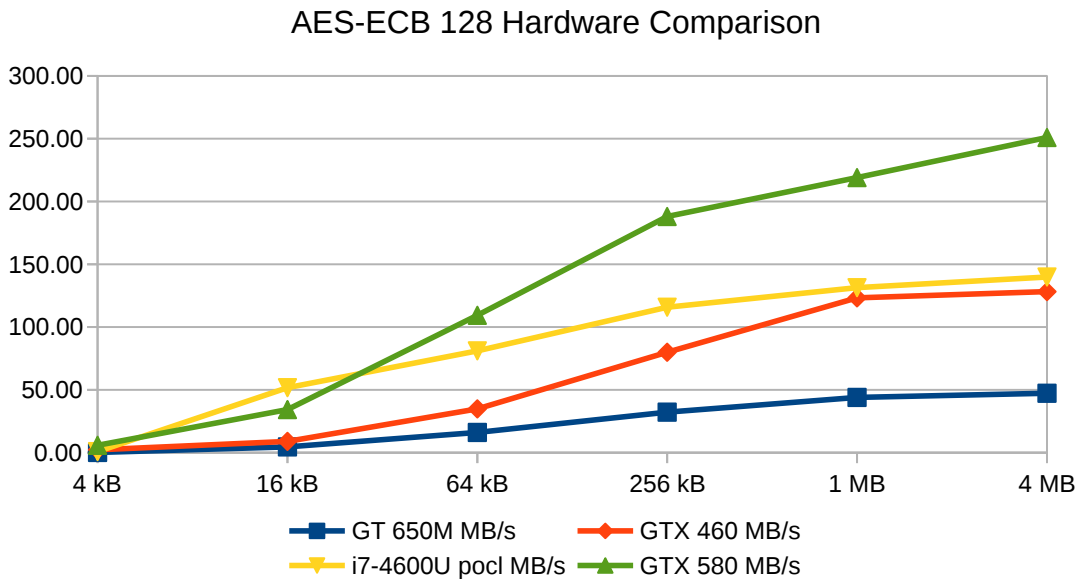

Figure 6.4: AES-ECB 128 hardware comparison

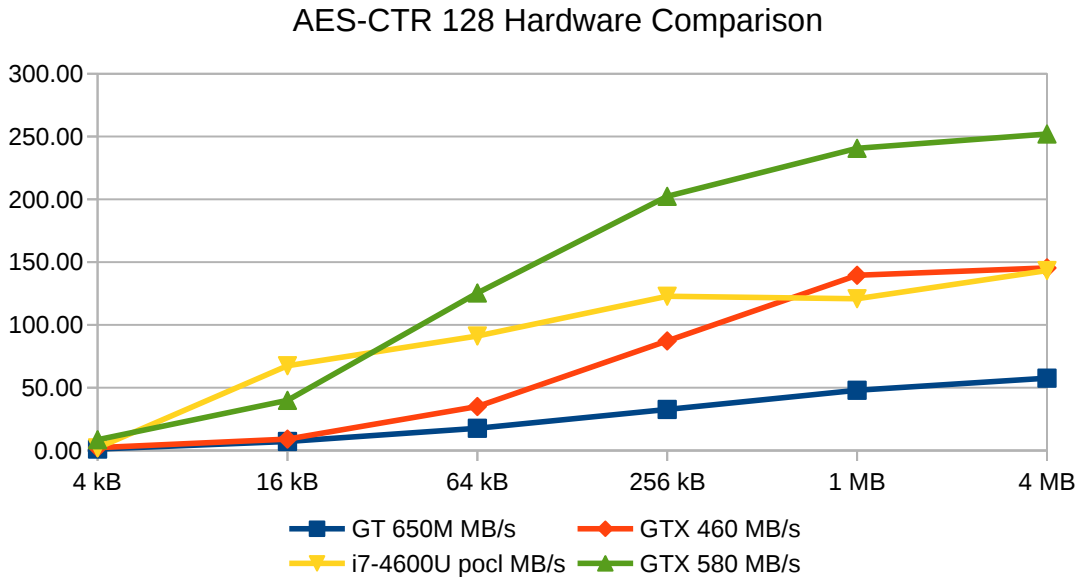

Figure 6.5: AES-CTR 128 hardware comparison

These results show quite clearly that there is future potential as *GPU*s get

faster and faster. *NVIDIA GTX 580* clearly outperforms our main *GPU* by a significant amount. Both *GTX 460* and *GTX 580* are quite old. There are far better performing *GPU*s on the market that we have not tested with. It is very surprising that *portable OpenCL* results from an *Intel i7-4600U* — a mobile *CPU* — are so close to the *GTX 460*. Not having to do memory transfers is a clear advantage in favor of the *CPU* but the *CPU* has just two real cores.

Let us look at latencies — the time between input data transfer starts and result data transfer ends. Predictably, latencies increase as buffer sizes increase.

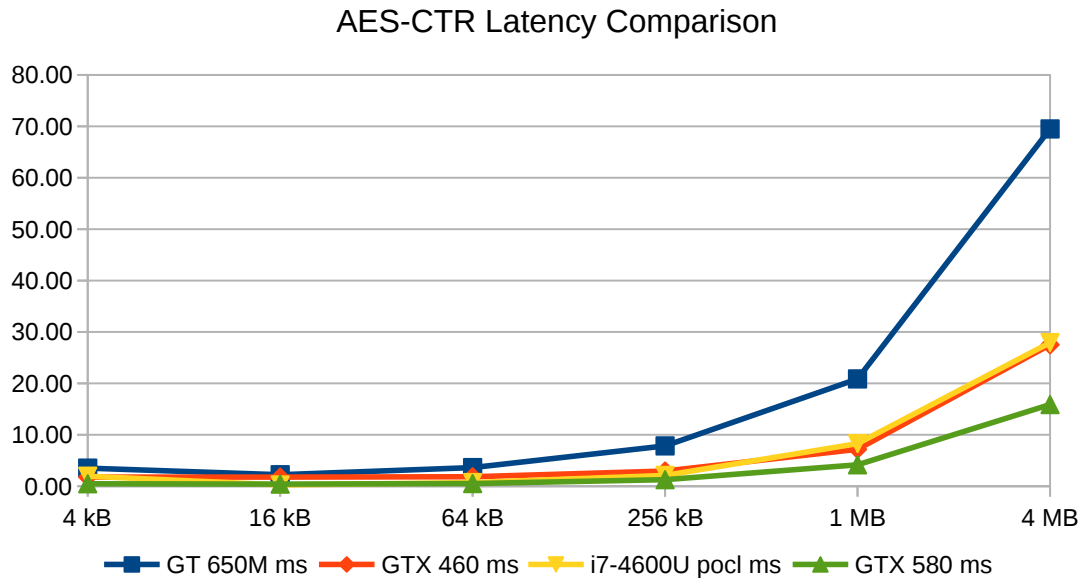

Figure 6.6: AES-CTR 128 latency comparison

We can observe that our solution gets more efficient as latencies increase. This suggests that the high performance can be exploited for bulk processing but is less useful for real-world cryptographic protocols.

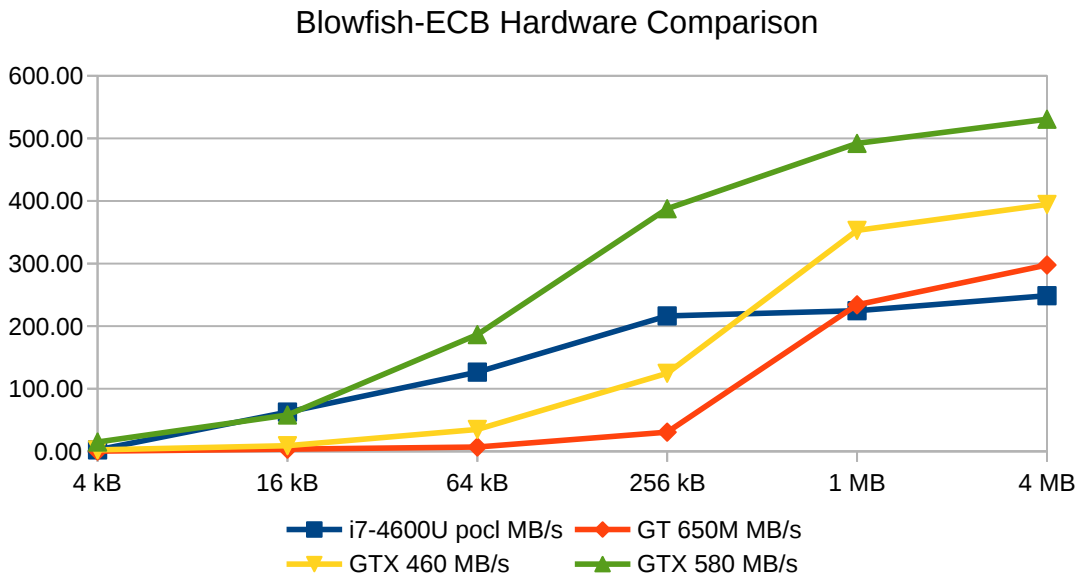

Figure 6.7: Blowfish-ECB hardware comparison

Blowfish-ECB paints a slightly different picture. Even *NVIDIA GTX 650M* — the slowest *GPU* we tested — is faster than *Intel i7-4600U* for large buffer sizes. The most likely reason is that *Blowfish* offers twice the occupancy of *AES* and overall is a simpler algorithm.

These numbers clearly show that cryptography on *GPU* can be very fast but also exhibits high latencies. For some use-cases like bulk encryption we do not care but for anything real-time — like hard drive encryption — this would be a big issue.

Let us also compare memory transfer speeds of all tested hardware since memory transfers take a significant amount of time with the *GPU*s. The *NVIDIA GTX 650M* is excluded for lack of data. Portable *OpenCL CPU* results are excluded because with pinned memory that is a *NOOP*.

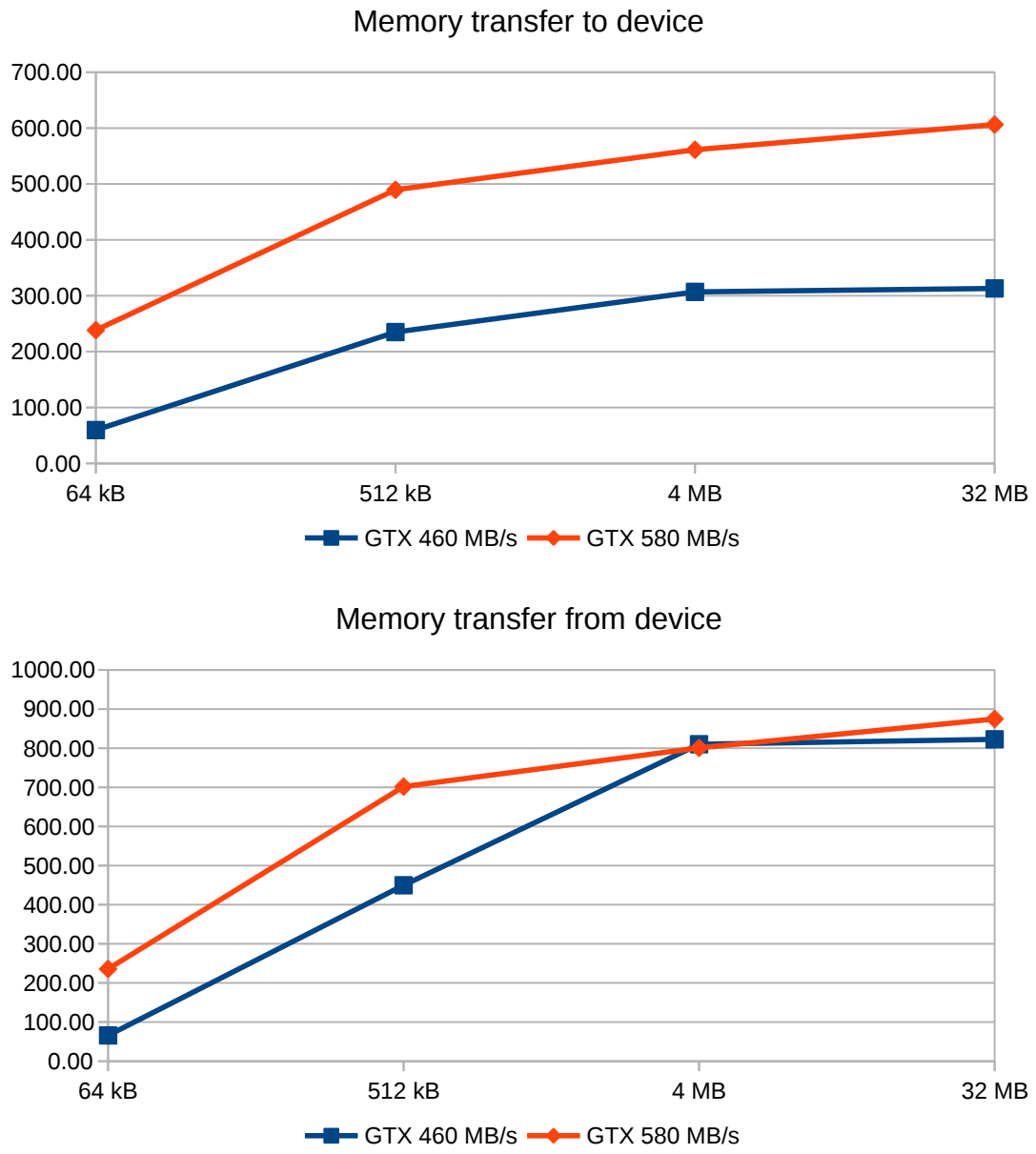

Figure 6.8: Memory transfer hardware comparison

These charts suggest that the graphics card is the bottleneck when transferring memory. It would be interesting to run these benchmarks with a modern *PCI-E 3.0 GPU* to confirm.

7 Integrations

7.1 oclcrypto-cli

As part of this project we created a small command line tool with an argument syntax inspired by *openssl enc*. It allows users to encrypt or decrypt files using algorithms available in *oclcrypto*. It is the simplest practical application of the library. It does not require user to choose any *OpenCL* device, instead it will choose the first available. The only required arguments are input and output files and key.

```
$ oclcrypto-cli aes-ecb-enc helohelohelohelo \
  plain.txt cipher.txt
$ oclcrypto-cli aes-ecb-dec helohelohelohelo \
  cipher.txt plain_copy.txt
$ diff -u plain.txt plain_copy.txt
```

Figure 7.1: oclcrypto-cli usage example

7.2 OpenSSL Engine Integration

To properly test the library in practice we decided to integrate it with *OpenSSL*. The *OpenSSL* engine *API* [19] was used because it is modular and allows for plugins to be loaded at run-time that can replace the inbuilt cryptographic functions. That seemed like a great idea at first. The API itself turned out to be not well suited for our use case.

First, let us outline how an ideal *API* would look like for *oclcrypto*. Before the cryptographic context class was initialized a function would be called with key size and buffer size. This function would be able to determine whether this cryptographic engine should be used, depending on the buffer size. For example if buffer size is just 128bits the function would decline to be used. That way we would never waste a lot of time setting all the *OpenCL* infrastructure up just to encrypt 128bits. Secondly, all the data would either be transferred in very big

chunks or even all at once. Transferring partial results out of *OpenCL* devices incurs a noticeable overhead.

Unfortunately the *OpenSSL* engine *API* allows neither of these. When initializing the context class we do not get the key size or the amount of data that will be processed in the future. First the key is set, without any information about future buffer sizes. Given key is expanded, the results are transferred to *OpenCL* data buffers. When the data starts coming in later, it is too late to perform another *CPU* key schedule and process the data on the *CPU*, we would be duplicating work. We conclude that the *API* is too low-level for our use case. Maybe low-level is not the right word for it but it surely is not suitable for *CPU / GPU* auto-negotiation.

The work in progress *OpenSSL* engine integration is optional, only compiled when *OpenSSL* headers are found at configure-time and can be found in the *openssl_engine* folder of the project.

8 Security Concerns

Gaming enthusiasts drive the sales of modern consumer *GPU*s [22] so it is no surprise that *GPU* manufacturers optimize almost exclusively for computer games. Driver releases are often focused solely on improving performance of a newly released game. Needless to say there is not a big push towards correctness and security in what is mainly gaming hardware. *GPU* vendors even provide multiple versions of various instructions, one version is correct and adheres to the standard and one is faster but less precise [24].

New in GeForce 337.88 Game Ready WHQL drivers:

Game Ready — This 337.88 Game Ready WHQL driver ensures you'll have the best possible gaming experience for Watch Dogs.

Performance — Introduces key DirectX optimizations which result in reduced game-loading times and significant performance increases across a wide variety of games compared with the previous 335.23 WHQL.

Figure 8.1: *NVIDIA* 337.88 driver release announcement

One of the most common security issues in *GPU*s are information leaks. Information in this context can be part of a texture or other data buffer.

GPU vendors cut corners to maximize performance. This means that memory in the *GPU* may not be overwritten when it is being deallocated and future use of this part of memory may reveal past data. Unfortunately, *GPU* vendors are secretive to maintain their competitive advantage and do not generally release how their architectures work. Di Pietro et al. [26] go into detail about how data is leaked in a paper from 2013. In many instances memory buffers are left at the mercy of the driver and there is no way to be sure that their contents will be overwritten. The driver also handles memory protection. The operating system uses similar methods to protect memory but the protection methods are available for audit and subject to scrutiny.

Vasiliadis et al. [36] look at the issue from the opposite end and propose to store keys in *GPU* instead of *CPU* as a measure to avoid key leakage. Running kernels indefinitely is proposed as a way to avoid data buffer leakage. If the

memory is taken by a kernel the scheduler will not allow other kernels to read or write to it. Instead of storing the keys in global memory the keys are stored in registers. As long as the kernel is running the registers are not cleared. Unfortunately registers can only store a few secret keys, the storage space is small on consumer *GPU*s. To solve this, encrypted keys are stored in global memory and the encryption key is stored in registers. That way, the adversary can steal the encrypted key-store but cannot decrypt it.

Bernstein [23] provided a timing attack against *OpenSSL AES* implementation in 2005. Using table look-ups in *AES* makes the implementation susceptible to a timing attack if the attacker can encrypt or decrypt any data. Researching whether a timing attack is viable against our implementation is out of scope of this project. We can speculate that since we use look-up tables our implementation is susceptible. The author provides a list of problems of existing *AES* implementations and their solutions. Unfortunately the solutions are often not applicable to *OpenCL* kernels.

9 Areas for Future Improvement

9.1 Splitting up AES Block Processing

All of the algorithms in this project are parallelized per block. In case of *AES*, one *128bit* block is processed by a single thread. A *warp* of 32 threads processes 32 *AES* blocks. There is strong indication that *AES* one block per thread is the fastest arrangement [28]. However, this only applies for very large plain-text sizes where the entire *GPU* is utilized — occupancy is close to 100%. For small block sizes we end up using just a part of the *GPU*. It seems viable to use an arrangement with slightly lower throughput but higher occupancy for small plain-text sizes. Four threads per *AES* block is the first arrangement to explore. In this arrangement, a *warp* of 32 threads processes 8 *AES* blocks.

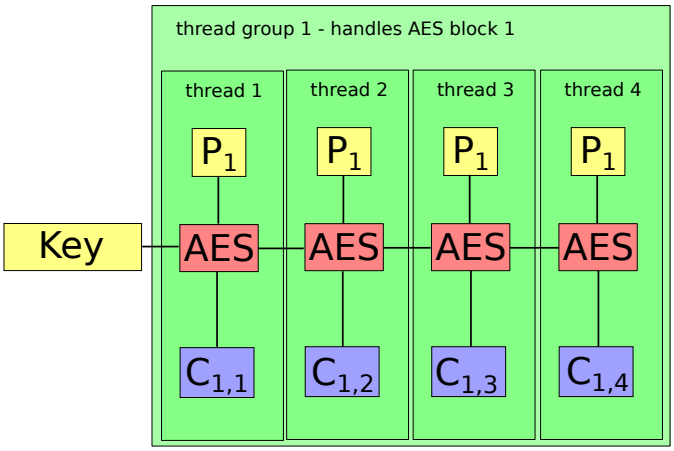

Figure 9.1: AES-ECB with 4 threads encrypting 1 AES block

The idea is fairly simple, however there are several obstacles. First of all, for each step of *AES-ECB* we need the previous step finished and we need its data. So all four threads have to cooperate very closely and wait for each other. The data has to be kept in *__local* memory for easy access. Second obstacle is that each of the four threads has to operate with slightly different instructions. This is not ideal for most GPU architectures that expect a massive amount of threads, each running exactly the same instructions with different input data.

The resulting algorithm should be faster for small buffer sizes because more of the *GPU* resources are used. However it would likely be slower for large buffer sizes due to the synchronization overhead.

9.2 Interleaving Data Transfer and Processing

Copying data in, processing and then copying data out resembles a lot of real-world use-cases. When used repeatedly on different data, it is possible to copy data to the *GPU* while the *GPU* is processing other data that was copied previously. This is called *Interleaved Data Transfer* and is available in both *OpenCL* and *CUDA*. Previous work suggests that interleaving data transfers can hide approximately *65%* of the data transfer cost in an *AES* encryption situation [28].

This project cannot use this technique at the time because it relies on the fact that device queues are processed in-order. This greatly simplifies the code-base at various places. The code-base would require extensive changes — like explicit locking — before *Interleaved Data Transfer* could be used.

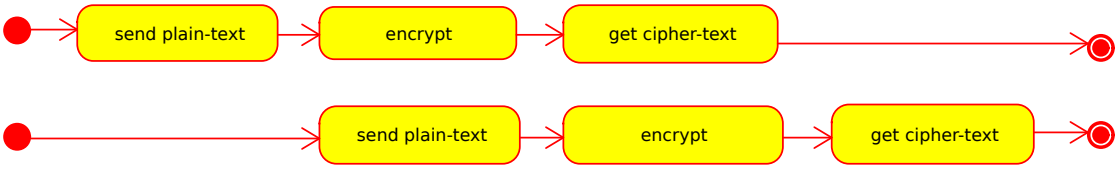

Figure 9.2: Example of *Interleaved Data Transfer*

9.3 Secure Key-Store

Our implementation stores the expanded key using global memory on the *GPU*. The global memory is readable from the host with very few restrictions. Anybody with permissions to use the *GPU* can read global memory. This renders our project unusable in a multi-user situation where users can use accelerated graphics. An adversary might read other users' keys by simply reading the global memory.

Switching to other key-store is out of scope of this project and would most likely decrease performance. A possible solution to store encrypted keys in

global memory and the key in registers is mentioned by Vasiliadis et al. [36].

9.4 Khronos Vulkan

The trend in graphics *API*s is to expose more of the internal complexity to the developer. Recent announcements of *Khronos Vulkan* [14] suggests that *GPU* vendors and developers both prefer the *API* to be closer to hardware.

It is likely that *OpenGL* will be replaced by *Khronos Vulkan* in the future. There is significant functionality overlap between *OpenCL* and *Vulkan*, therefore it may happen that *OpenCL* will also be replaced. It seems like a viable idea to explore the new *API* and perhaps even provide implementations of the *AES* and *Blowfish* kernels for it.

10 Conclusion

We have designed and implemented a portable open-source library for symmetric block encryption and decryption on a *GPU* or any other *OpenCL* device. The library supports *AES* and *Blowfish* ciphers, the implementations are covered by unit tests and measured by benchmarks. The library *API* processes all data on an *OpenCL* device but at the same time hides the complexity of *GPU Compute* from the user. Users can use the *API* with very little knowledge about *GPUs, OpenCL* or related technology.

The performance numbers we measured look promising. We have achieved *252 MB/s* of throughput with *AES-CTR* and *531 MB/s* with *Blowfish-ECB* on *NVIDIA GTX 580* including all set-up costs. Consumer *GPU*s used as *OpenCL* devices proved to be decent and cost-effective cryptographic accelerators. Efficiency of our solution increases as latencies increase, which makes the solution suitable for bulk processing rather than cryptographic protocols.

While there are problems and areas for improvements, we believe the solution presented can be used in real world applications as a substitute for dedicated cryptographic accelerators.

Nomenclature

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
CBC	Cipher Block Chaining
CPU	Central Processing Unit
CTR	Counter Mode
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DES	Data Encryption Standard
ECB	Electronic Code Book
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCM	Galois / Counter Mode
GDDR	Graphics Double Data Rate
GLSL	OpenGL Shading Language
GPGPU	General Purpose GPU Computation
GPU	Graphics Processing Unit
HLSL	High-Level Shading Language
IC	Initial Counter
IEEE	Institute of Electrical and Electronics Engineers
IPSec	Internet Protocol Security

IV	Initial Vector
NIST	National Institute of Standards and Technology
NOOP	No Operation
NSA	National Security Agency
NVIDIA Cg	short for C for Graphics
OpenCL	Open Compute Library
OpenGL	Open Graphics Library
OpenGL ARB	OpenGL Architecture Review Board
RAII	Resource Acquisition Is Initialization
RAM	Random Access Memory
SBox	Substitution Box
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SSL	Secure Sockets Layer
TLS	Transport Layer Security
XOR	eXclusive OR

Bibliography

[1] AMD CodeXL benefits in detail. http://developer.amd.com/tools-and-sdks/opencl-zone/codexl/codexl-benefits-detail/. Accessed: 2015-05-10.

[2] AMD picks up an OpenGL and CL tool makers as well as an RnD site. http://www.pcper.com/news/General-Tech/AMD-pick-OpenGL-CL-tool-makers-well-RD-site. Accessed: 2015-05-07.

[3] Blowfish F function diagram by Decrypt3 and DnetSvg. https://commons.wikimedia.org/wiki/File:BlowfishFFunction.svg. Accessed: 2015-05-10.

[4] C++ compiler support. http://en.cppreference.com/w/cpp/compiler_support. Accessed: 2015-05-08.

[5] Cg toolkit website. https://developer.nvidia.com/cg-toolkit. Accessed: 2015-04-27.

[6] Compute shader overview. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx. Accessed: 2015-03-27.

[7] CUDA toolkit 3.1 download notes. http://developer.nvidia.com/cuda-toolkit-31-downloads. Accessed: 2015-05-08.

[8] CUDA toolkit 5.0 release notes and errata. http://developer.download.nvidia.com/compute/cuda/5_0/rel/docs/CUDA_Toolkit_Release_Notes_And_Errata.txt. Accessed: 2015-05-08.

[9] GPU-acceleration applications for HPC industries. www.nvidia.com/content/gpu-applications/PDF/GPU-apps-catalog-mar2015.pdf. Accessed: 2015-04-27.

[10] Intel advanced encryption standard (AES) instructions set - rev 3.01. https://software.intel.com/en-us/articles/intel-

advanced-encryption-standard-aes-instructions-set/. Accessed: 2015-03-27.

[11] Intel VTune amplifier 2015. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed: 2015-05-08.

[12] Khronos - OpenCL - the open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. Accessed: 2015-03-27.

[13] Khronos promoter members. https://www.khronos.org/members/promoters. Accessed: 2015-05-16.

[14] Khronos releases Vulkan API. https://www.khronos.org/news/press/khronos-reveals-vulkan-api-for-high-efficiency-graphics-and-compute-on-gpus. Accessed: 2015-05-11.

[15] NVIDIA - about CUDA. https://developer.nvidia.com/about-cuda. Accessed: 2015-03-27.

[16] NVIDIA's next generation CUDA compute architecture - Fermi. http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf. Accessed: 2015-03-27.

[17] nvprof user's guide. http://docs.nvidia.com/cuda/profiler-users-guide/. Accessed: 2015-05-16.

[18] OpenGL ARB compute shader specification. https://www.opengl.org/registry/specs/ARB/compute_shader.txt. Accessed: 2015-03-27.

[19] OpenSSL engine API. https://www.openssl.org/docs/crypto/engine.html. Accessed: 2015-05-11.

[20] Ruby-OpenCL homepage. http://ruby-opencl.rubyforge.org/. Accessed: 2015-04-27.

[21] SSLShader website. http://shader.kaist.edu/sslshader/. Accessed: 2015-05-16.

[22] The state of PC graphics sales Q2 2014. http://www.anandtech.com/show/8446/the-state-of-pc-graphics-sales-q2-2014. Accessed: 2015-04-27.

[23] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[24] NVIDIA Corporation. CUDA programming guide. 2012.

[25] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2002.

[26] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA leaks: information leakage in GPU architectures. *arXiv preprint arXiv:1305.7383*, 2013.

[27] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, (7):33–38, 2008.

[28] Keisuke Iwai, Naoki Nishikawa, and Takakazu Kurokawa. Acceleration of AES encryption on CUDA GPU. *International Journal of Networking and Computing*, 2(1):131–145, 2012.

[29] Keon Jang, Sangjin Han, Seungyeop Han, Sue B Moon, and KyoungSoo Park. SSLShader: Cheap SSL acceleration with commodity processors. In *NSDI*, 2011.

[30] Andreas Klöckner. PyOpenCL homepage. http://mathema.tician.de/software/pyopencl/. Accessed: 2015-04-27.

[31] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007. Chapter 36: AES Encryption and Decryption on the GPU.

[32] NIST FIPS Pub. 197: Advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197:441–0311, 2001.

[33] Aamir Majeed Usman Aziz Salman Ul Haq, Jawad Masood. Bulk encryption on GPUs. http://developer.amd.com/resources/documentation-articles/articles-whitepapers/bulk-encryption-on-gpus/. Accessed: 2015-05-03.

[34] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption*, pages 191–204. Springer, 1994.

[35] PCI SIG. PCI Express base 2.0 specification, 2007.

[36] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. PixelVault: Using GPUs for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2014.

A oclcrypto README

A reusable *C++11* library for *OpenCL* hardware accelerated cryptography.

Compilation on Linux

Build dependencies (Fedora)

```
# if you do not have build tools installed already
yum install @development-tools

yum install cmake boost-devel ocl-icd-devel opencl-headers
```

(Replace yum with dnf if you use Fedora 22 or newer.)

Build dependencies (Debian or Ubuntu)

```
# if you do not have build tools installed already
apt-get install build-essential

apt-get install cmake libboost-all-dev \
  ocl-icd-opencl-dev opencl-headers
```

Configure step

Feel free to replace build/ with a build directory of your choice. Using the source directory is discouraged and may not work, only out-of-source builds are supported.

```
mkdir build
cd build/
cmake ../
```

Build step

```
cd build/
make
```

Install step

```
cd build/
sudo make install
```

The command above will install headers into $prefix/include/oclcrypto and libraries into $prefix/lib(64). There is no automatic uninstall step. You are recommended to use the install step while packaging.

Running unit the tests

```
cd build/
ctest –V
```

Compilation on Windows

Build dependencies

Microsoft Windows unfortunately lacks a well supported package manager, all dependencies have to be downloaded manually.

- *Microsoft Visual Studio* — version 2013 is recommended
- *cmake 2.6+* and *cmake-gui*
- *Boost* libraries precompiled for *Windows*
- *NVIDIA CUDA Toolkit 5.0+*

Configure step

Run *cmake-gui*, select the repository as the source directory and select directory of your choice as the build directory. Selecting source directory as the build directory is discouraged and unsupported, do an out-of-source build instead. In *cmake-gui*, select *Configure*, walk through the wizard dialogs and then click *Generate*. *Visual Studio* solution and project files will be generated inside the build folder.

Build step

Open the *Visual Studio* solution, select a desirable configuration — *Debug* or *Release* — and click *Build*.

Install step

It is not customary to run the install step on *Windows* system. Instead, it is recommended to bundle the built *DLL* files with your application.

Running the unit tests

Double-clicking *oclcrypto-tests* will work but the output will be lost after tests finish. We therefore recommend running the tests from a terminal emulator of your choice. *cmd.exe* will also work.

```
cd build
oclcrypto-tests.exe
```

Using the API

Browse the *tests/* folder for sample usage of the API.

B Minimal Example Program

```
#include <oclcrypto/System.h>
#include <oclcrypto/Device.h>
#include <oclcrypto/AES_CTR.h>

int main(int argc, char** argv)
{
  oclcrypto::System system;
  oclcrypto::Device& device = system.getBestDevice();

  oclcrypto::AES_CTR_Encrypt context(system, device);
  context.setKey("helohelohelohelo");
  context.setInitialCounter("1234567890abcdef");
  context.setPlainText("testingplaintext");
  context.execute();

  {
    auto data = context.getCipherText()->lockRead<unsigned char>();
    // now we can use data[index] to access the resulting ciphertext
  }

  return 0;
}
```