UNIVERSITY OF OSLO
Department of Informatics

Offloading an encrypted user space file system on Graphical Processing Units

Master thesis

Magne Eimot -
magneei@ifi.uio.no

Contents

List of Figures

List of Tables

Abstract

Modern computers often have powerful graphics processing units, either on dedicated graphic cards or integrated on the motherboard. These units can used by applications for demanding computation. An important use of this technology is to assist the main CPU in the system, offloading some of its work to the GPU. Offloading can give increased performance, as well as decreased load on the main CPU. Decreasing the load frees resources for other applications.

Keeping documents, images and other potentially sensitive files private is important for many users. One way to do this is to use an encrypted file system, which can prevent others from gaining unauthorized access. However, such a file system occupies resources in the computer system. In this thesis, we evaluate how GPUs can be used for assisting the computationally expensive encryption part of an encrypted file system.

Programming GPUs, is challenging because of the GPUs massively parallel nature and their many memory types. We will look into different architectures, but focus mainly on NVIDIAs architecture and programming framework in our work on evaluating the effects of using graphic processing units for offloading an encrypted file system. In this thesis, we see that offloading parts of this file system is beneficial, giving better performance and reduced CPU load.

Acknowledgments

I would like to thank my advisors, Håkon Kvale Stensland, Pål Halvorsen and Carsten Griwodz. Their feedback and assistance have been inspiring, and this thesis would not have been possible without them. I would also like to thank Håvard Espeland and Paul Beskow for valuable feedback.

I would also like to thank Alexander Ottesen, which I have worked with for some of this thesis. His help has been invaluable and his hard-working attitude has been inspiring.

Writing this thesis at Simula Research Laboratory has been great, I would like to thank the whole lab for a great social and academic environment.

Also, I would like to thank my friends and family, which have been of great support through this thesis.

Oslo, April 30, 2009

Magne Eimot

Chapter 1

Introduction

1.1 Background and Motivation

Modern GPUs from NVIDIA, Intel and AMD have over the years evolved into highly programmable processing units, that can be used for other purposes in addition to rendering graphics. We will investigate some of the possibilities these architectures give us, with an emphasis on the NVIDIA architecture and programming method. An example of an NVIDIA Graphics Card is seen in figure 1.1.

Figure 1.1: A NVIDIA GTX 280 Card (From NVIDIA [1])

Security is an important aspect in todays society, and protecting ones data is important. For a computer to be protected against someone gaining unauthorized access, an important attack vector is the local storage of a computer. The most common way to do this is to use data encryption to render the content of the storage medium useless, unless the correct authentication token can be provided. This token is commonly a pass phrase, which can be combined with a physical token like a smart card. Having a encrypted local storage, requires a encrypted file system. In such a file system, the data encryption is compute intensive. In addition to require good performance from the encrypted storage, it is desirable not to occupy too much of the CPU time of the system, because the computer will generally be used for other tasks in addition to the encryption required to secure data.

Utilizing the GPU, which is present in many modern computer systems for the encryption and decryption, process is interesting to investigate; both in terms of performance and to make CPU resources available for other applications in the system. When we discuss offloading in this thesis, both of these aspects will be considered.

In the CPU market, there is no longer a race for higher clock frequencies of the processors. Generally, increasing the clock frequency also increases the power consumption, and therefore heat in the processors. For example a Pentium IV processor has the power density ($Watts/cm^2$) of a hot plate. To increase performance, the new generation of processors now have multiple cores, which means that we need to focus on parallel programming. This phenomena is often called the *Power wall*, because there is a limit to the clock frequency a processor can have. For the computation power to increase, more cores have to be added. GPUs on the other hand have lower clock frequencies, and a much higher number of cores, and we want to investigate how and if these devices can be used on tasks previously done by CPUs.

Using GPU which is often available in modern computer systems to assist in the encryption process of a file system is therefore interesting, considering the GPUs good performance. Previous research have been successful in offloading tasks to a GPU, e.g. the Reed Solomon encoding discussed in chapter 2, where a RAID solution was offloaded using a GPU. This might give some insight in how to successfully offload a part of a file system.

1.2 Problem Definition

GPGPU is an area of research where there are unknown factors regarding their usage. Parallelizing applications for running on GPU is challenging, as there are limited debugging available, and the massively parallel architecture of the GPUs.

Adapting existing applications to run on GPU is one way to use the GPUs, as an alternative to writing the applications from scratch to suit the GPUs. To adapt the applications, requires that one finds the computationally demanding parts of an application. The problem must then be investigated too see if is possible to map the problem to suit a GPU. Most application was never intended to run on GPU, and porting them to GPU is therefore challenging.

One field where little work has been done, is file systems. Some of these have advanced features like data encryption, or built in redundancy. They are traditionally written to run on ordinary CPUs, which could possibly be better used for other applications.

In this thesis we want to evaluate the benefits of offloading the encryption part of a file system to the GPU.

1.3 Main Contributions

This thesis evaluates GPU offloading of the encrypted file system EncFS. We divide the job of offloading an application into several parts, looking at the GPU kernel, the whole application, and optimization of the application. We investigate and run experiments on each part to get a better understanding of running an application on the GPU.

We will look into how the popular *Advanced Encryption Standard (AES)* algorithm work. Following this, we create GPU implementations of AES, based on different CPU implementations, and evaluate their properties. These implementations will form a basis for offloading the encryption part in a user space file system, where we will integrate these implementations, and investigate their effects.

In this thesis, we evaluate the performance of some different AES encryption implementations. We look into different optimization techniques, and show how memory accesses can be done more optimal. We show that an lookup based encryption implementation can run even faster on GPU, and that a naive encryption implementation can have a even higher relative speedup.

This is followed by an investigation how the CUDA framework can be used for porting an encryption application to the GPU, what type of applications are best suited, and what happens to the execution time of two different implementations of an encryption algorithm. We also look into the encrypted user space file system, and evaluate what offloading effects we can achieve if we use the GPU to run the encryption algorithm, instead of running it on the CPU.

Finally, we investigate a method for increasing application performance by allowing overlap between GPU computation and memory copying, called CUDA streams. We show how using CUDA streams can give reduction in execution time, and also test the maximum speedup for applications.

1.4 Outline

In chapter 2, we look into different possibilities for programming modern GPUs. This is followed by introduction to the different architectures of modern GPUs in chapter 3. We further investigate one of the frameworks mentioned, namely the CUDA framework in chapter 4, and see what possibilities this gives us. In chapter 5, we investigate how a encryption application can be ported and ran on the GPU, how we best use the GPU, and compare it to a CPU based implementation. This is followed by chapter 6 where we investigate the offloading of encryption in a user space file system. We will then look into CUDA streams in chapter 7, followed by a discussion in chapter 8, and a conclusion of our work in chapter 9.

Chapter 2

Graphic Processing Units Programming Examples

2.1 Introduction

General-purpose computing on graphics processing units (GPGPU) is the technique of using a GPU not for graphics but for solving general computer problems with the hardware available on the GPU. In this chapter, we look into some example of what GPUs are used for, like accelerating RAID solutions, physics simulations, Folding@Home and video coding.

The purpose of this chapter is to get a introduction into what kind of tasks can benefit from GPU assistance, before we continue exploring one framework for programming a GPU and look closer into one use case for GPUs.

2.2 Accelerating Reed-Solomon Coding

Reed-Solomon is an error-correcting code, which is used in many applications, like CDs and DVDs, data transmission and for providing data redundancy. Redundant Array Of Inexpensive Disks 6 (RAID6) is a data redundancy technique which gives *n-2* hard disks of capacity, given *n* hard disks. The structure of a RAID 6 is seen in figure 2.1, where the numbered blocks are data chunks, and the blocks called *p* and *q* are parity blocks. It can sustain data integrity when up to two of the hard drives fail. The Reed-Solomon coding required to store this redundancy is compute intensive, and we therefore look into a GPU assisted implementation of RAID6.

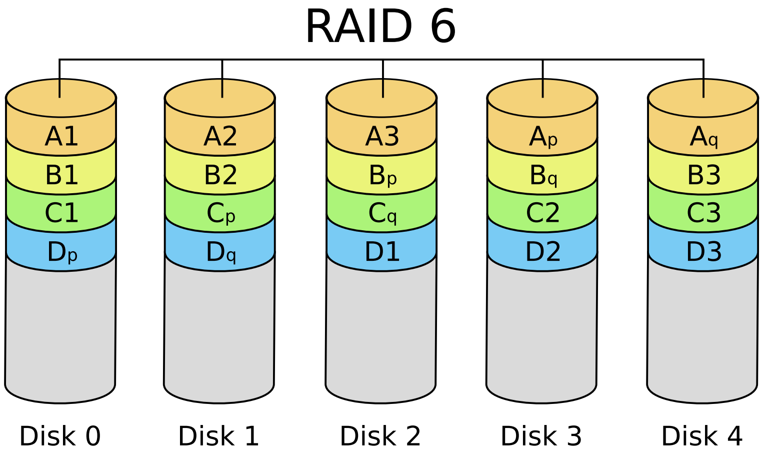


Figure 2.1: Overview of RAID 6 (From Wikipedia [2])

Curry et al. have investigated how a GPU can be used for computationally intensive RAID solutions, where you have 2 or more parity disks [16]. They found that a GPU would handle much larger data throughput, in some cases 5 times the throughput of a Intel Core 2 Quad 6600. They also found that because of all the bus transfer of data, the solution would be most effective if incoming data were written to a buffer, and that the write operation would seem for the application to be completed, then done in the background by the GPU. If it would be checksummed while the write-operation had not yet returned, it would cause a greater latency than when used with the CPU.

2.3 PhysX Physics API

Another application is the PhysX Physics API [17] for games. PhysX was a solution created by Ageia which created a hardware-accelerated solution for games, where physics heavy computing could be offloaded. This computation allowed more advanced and realistic game performance. NVIDIA acquired Ageia, and they have ported PhysX to work on their CUDA enabled GPUs. PhysX is designed for vector calculations and trajectories of large quantities of physical objects, that this is a type of calculation well suited for a GPU.

2.4 Seti@Home Distributed Computing

Seti@Home [18] is a distributed computing project, where the goal is to detect intelligent life outside Earth. The project analyzes large data sets from radio receivers, and

OS Type	Current TFLOPS[a]	Active CPUs	Total CPUs
Windows	200	210098	2104468
Mac OS X/PowerPC	7	8200	117721
Mac OS X/Intel	28	8928	55312
Linux	70	40917	320755
GPU	921	8370	20669
PLAYSTATION®3	1347	47757	568013
Total	2573	324270	3186938

Table 2.1: Current performance numbers, taken from Stanford University [10]

[a] TFLOPS are actual performance numbers, not theoretic peak

have ported their client to support GPUs from NVIDIA.

2.5 Folding@Home Distributed Computing

Folding@Home [19] is another distributed computing project, performing CPU-intensive simulations on protein folding. The project allows users to run a client on their computer, which will download jobs from a central server, do simulations on the data, and upload the data back to the server. The purpose of this research is to better understand diseases, like Alzheimer's disease, Parkinson's etc. The distributed computing allows researchers to get access to more processing power than they could get by running it on their own hardware.

In October 2006, a Folding@Home client for the Microsoft Windows XP operating system was released, which included support for running the simulations on the ATI R520 family of GPUs. After 9 days, the project had received 31 teraFLOPS of computational performance, which averaged 70 times the performance of CPU based simulations on the project [20], the current performance numbers shown in table 2.1. Recently a client that supports NVIDIAs GPUs has been released, which have further increased the computational power available from GPUs. Now the GPUs have 4 times the computing power of CPUs, even though there are ten times as many CPU-oriented clients.

The project also supports PlayStation 3 (PS3), which are based on the Cell Broadband Engine Architecture (CBEA) [21].

The current performance numbers of the various clients in the Folding@Home project are shown in table 2.1.

2.6 Video decoding, encoding and transcoding

Another application for GPUs can be to offload some of the calculations needed for decoding video frames. Wesley et al. have investigated the performance gains available when offloading some of the calculations to the GPU. They have implemented a pixel-shader based approach to decoding video frames and have shown that this was about twice as fast on a GPU than on a CPU [22]. This was with respectively a non CUDA-enabled NVIDIA GeForce 6800 GPU and a AMD Athlon XP 2800+ CPU.

Elemental Technologies, Inc. has an application called Badaboom Media Converter [23], which is an application for transcoding and encoding video and audio files, that offloads some of the process to the GPU.

2.7 Mars MapReduce GPU implementation

MapReduce [24] is a software framework designed to support distributed computing on large data sets. It was inspired by the map and reduce functions used in functional programming, and requires that your problem can be written in this form.

- The Map step takes input, divides it into smaller problems, and distributes it to worker nodes (which the worker node might do again).
- The worker nodes then solve the problem, and returns the answer back.
- The Reduce step assembles the answers in the correct order, to yield the correct result.

He et All found that implementing MapReduce on a GPU in what they called the Mars framework could be up to 16 times faster than its CPU based counter part for some common web applications [25].

2.8 Summary

In this chapter we have investigated some of the possible applications for using a GPU. We have seen that many different compute intensive applications have benefited from using a GPU. Therefore we want to look further into how a GPU is designed, by looking into the hardware and structure of some different GPU architectures from NVIDIA, Intel and AMD, where we get a understanding of how these architectures can provide a platform for high performance applications.

Chapter 3

Graphic Processing Units Architectures

3.1 Introduction

NVIDIA, Intel and AMD all make or develop programmable GPUs. To understand the differences between programming them, we first look at their architectures to provide background on how they work and what sets them apart.

3.2 NVIDIA G80 Graphics Processing Unit

On November 8, 2006 NVIDIA released a new family of graphics cards based on the *G80* architecture [26]. In the NVIDIA G80 architecture shown in figure 3.3 we find a Stream Processor Array (SPA). There are multiple SPAs on each chip. Each SPA consists of Texture Processor Clusters (TPC). A TPC consists of two Streaming Multiprocessors (SM), a group of processors which can run threads. The TPC also contains one texturing unit which is not as relevant in the context of GPU programming, because we mostly use the SMs.

The SMs can access four different types of memory. The memory types accessible are one set of 32-bit registers, a shared memory, a read-only constant cache and a read-only texture cache. There are also uncached local and global memory spaces, that are implemented as read-write regions of the memory of each device. Each SM appears as eight streaming processors (SP) which are scalar ALUs able to run a single CUDA thread. The SM runs several threads in parallel with low overhead. The low thread scheduling overhead means that one can use more threads than one uses in traditional CPU programming. To get optimal performance from the GPU, more threads than ALUs are required, the reason for this will be explained further in section 4.3.2 A stream processor is somewhat similar to a thread processor, being able to work on multiple data in parallel. It can, however, reduce bandwidth demands on the memory system and thus can support more ALUs [27].

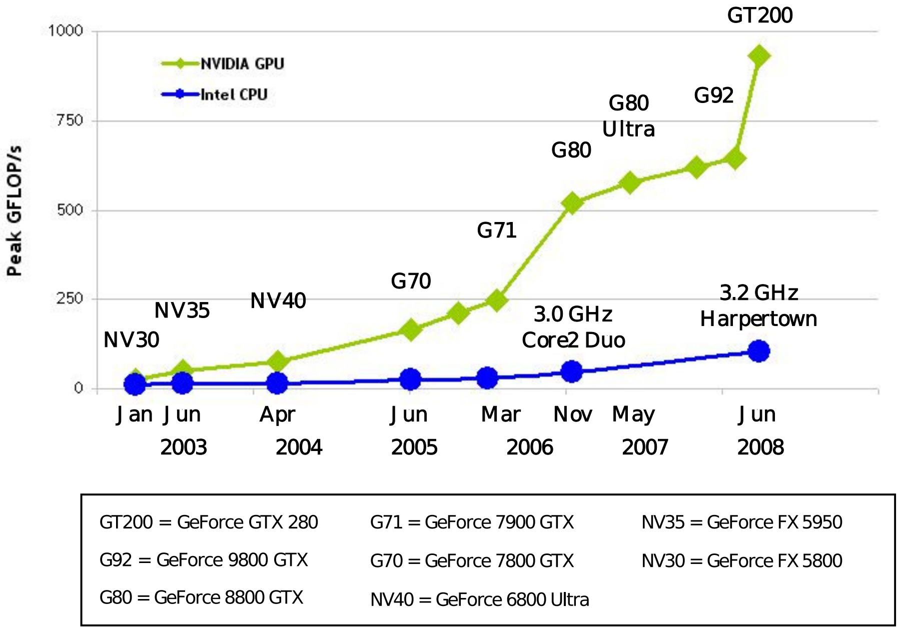


Figure 3.1: Development of GPU and CPU measured in Gigaflops (From NVIDIA [3])

Some of the different architectures from NVIDIAs, has their performance in GFlops (billion floating operations per second) displayed in figure 3.1. Performance measured in GFlops have not increased fast on modern Intel CPUs in the years from 2003 to 2008. On the GPUs, however, we can see a much faster increase. This gives some indication of the potential of a modern GPU compared to traditional CPU. It is important to notice that these performance numbers concerns only floating point operations, which is not the most relevant for all types of applications. However, floating point operations are common in scientific calculations that require a lot of computing power.

The memory bandwidth of NVIDIA GPUs has increased much faster than the memory bandwidth has increased for modern x86 CPUs, seen in figure 3.2. In 2003, the NV30 chipset from NVIDIA did not have that much higher bandwidth than the Intel "Northwood" processor. However, we see that the bandwidth has not increased much in the years up to 2007 for the x86 CPUs, compared to the very high increase for the NVIDIA chipsets, where the G80 Ultra has a memory bandwidth of up to 100GB/s.

These two performance metrics give some indication of the potential of the GPU to fulfil certain tasks. Especially, data-intensive floating point operations benefit from

Chip	TPCs	SMs per TPC	SPs per SM	Total SPs
GeForce 8 and 9 Series	8	2	8	128
GeForce GTX 200	10	3	8	240

Table 3.1: Number of GPU Processing Cores (From NVIDIA [11])

using the GPU.

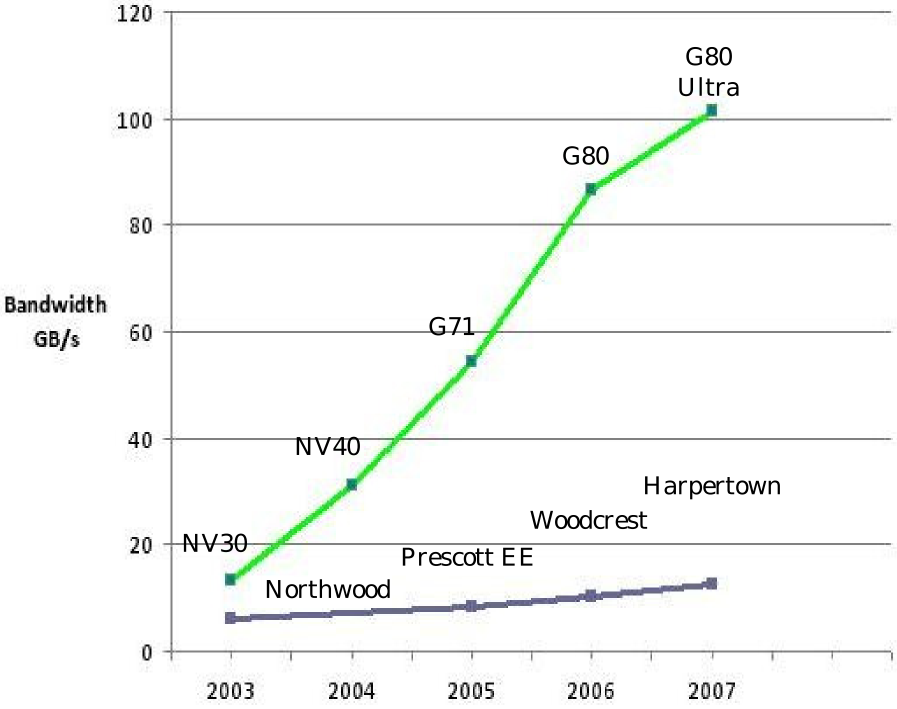


Figure 3.2: Development of GPU and CPU memory bandwidth (From NVIDIA [3])

The GeForce 8800 GTX has eight TPCs, with a total of 128 SPs, each TPC with 1024 32-bit registers. Figure 3.3. shows one of SPAs, which consists of 8 TPCs, each TPC has 16KB of shared cache between the SPs. This is shown in figure 3.3, a NVIDIA GeForce 8 GPU. For optimal performance, programs should follow the same code path in each SM, otherwise the programs will be run in sequence, instead of in parallel. Stream processors are somewhat similar to vector processors, but can reduce bandwidth demands on the memory system, and can thus support an order of magnitude more ALUs with the same memory bandwidth [28]. Table 3.1 shows some of the specifications of the newest NVIDIA GPUs.

The NVIDIA GeForce architecture is called Single Instruction Multiple Thread (SIMT), meaning that one instruction gets executed by multiple threads, in the TPC. Threads are organized in *warps*, which are groups of 32 threads. The SIMT processor manages the threads in a warp, and warps are batch scheduled. In a warp, all threads have the same starting point in the code, but are free to branch as they please. The fastest execution is achieved when all threads in the warp agree on a code path. This means branching can have a high cost, unless you get all threads in a warp to take the same branch.

The SM will, given a set of threads to run, divide these into warps, and schedule them for execution. For each instruction, the SM will choose a warp that is ready.

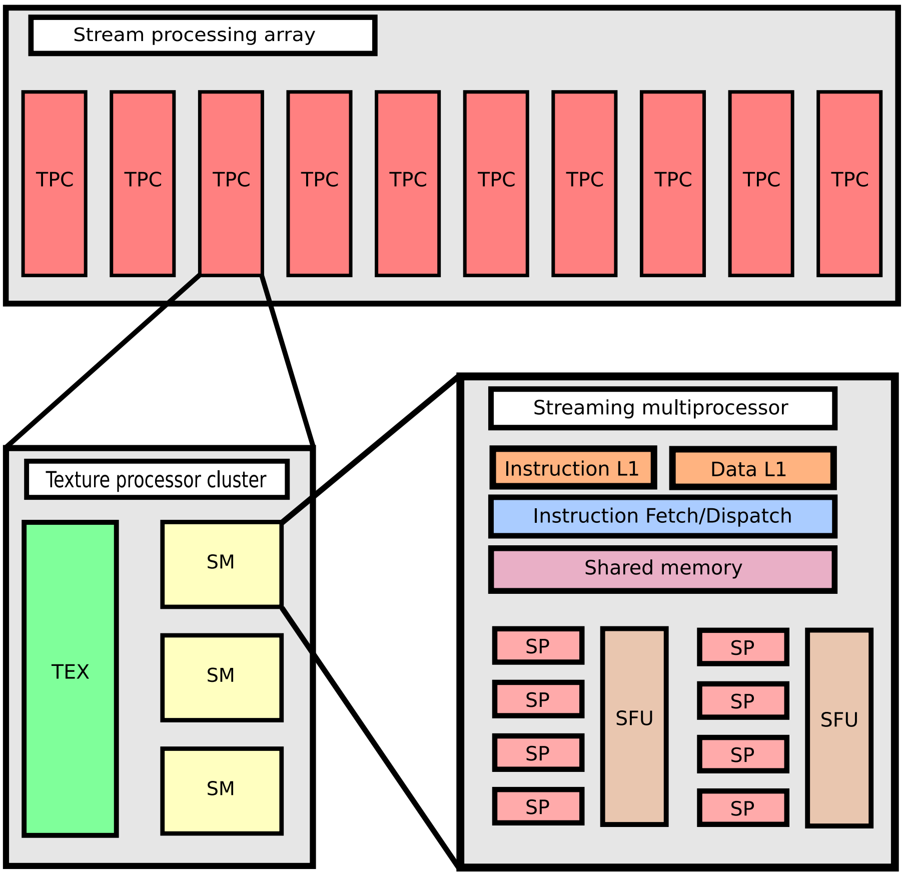


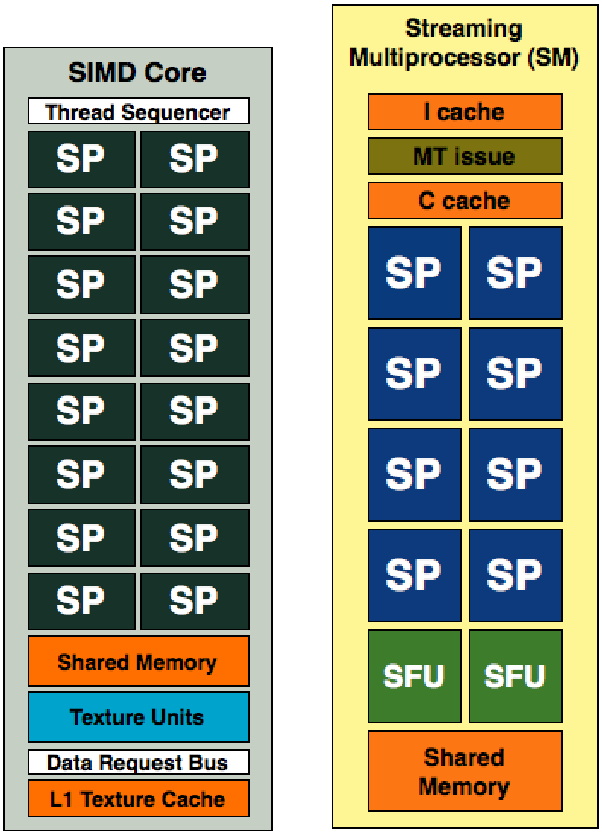
Figure 3.3: Architecture of the GeForce 8 GPU

3.3 AMD

AMD produces graphics chipsets under the ATI brand. Radeon R770 (code named RV770) [29] is AMD's name for the latest generation of graphics chipset which is used in modern ATI products, like the ATI Radeon HD 4800. This chipset contains of 800 streaming processors, which is an increase from the previous 320 cores found in the ATI HD 3800 card [30], that was based on the RV600 architecture. These units are divided into groups of 10 SIMD cores.

The RV770 architecture features a 256-bit memory controller, and was the first GPU to support GDDR5 memory, running at 900 MHz. This memory achieves a bandwidth of up to 115 GB/s. ATI previously used a ring bus like the Intel Larrabee architecture, that has been replaced for a combination of a crossbar and an internal hub in the RV770 architecture, see chapter 3.4.

AMD's Streaming Processor, seen in figure 3.6, consists of Stream Processing Units (SPU), that resemble NVIDIAs SP. The SPU consists of five units, called *x,y,z,w and t*. x,y,z,w are ordinary ALUs, and t is a transcendental unit, which can do all the operations of x,y,z,w and also transcendental operations. This means that unlike the SPs on NVIDIA cores, the AMD core can run five instructions simultaneously, giving the possibility for Instruction Level Parallelism (ILP), not only Thread Level Parallelism (TLP) as NVIDIA supports, because each of the NVIDIA cores can do only one instruction at the time.



(a) AMDs SIMD core (b) NVIDIAs SM

Figure 3.4: AMD and NVIDIAs groups of SPs (from Anandtech [4])

AMD group their SPs into a *SIMD core*, which is somewhat similar to the SMs in the NVIDIA architecture, shown in figure 3.4. There are, however, some differences worth noting. There are more SPs in AMDs SIMD Core, two times as many, and these SPs can do ILP. A difference between the AMD and NVIDIA cores is that there is no constant and instruction cache located in AMDs SIMD core, it is shared between SIMD cores. NVIDIA has no texture cache in its SM, which AMD has on the SIMD core.

3.4 Intel Larrabee

Intel is developing their own hybrid GPU technology, called Larrabee [31]. This design differs from the NVIDIA and ATI GPUs in that it will support the x86 instruction set, including EMT64.

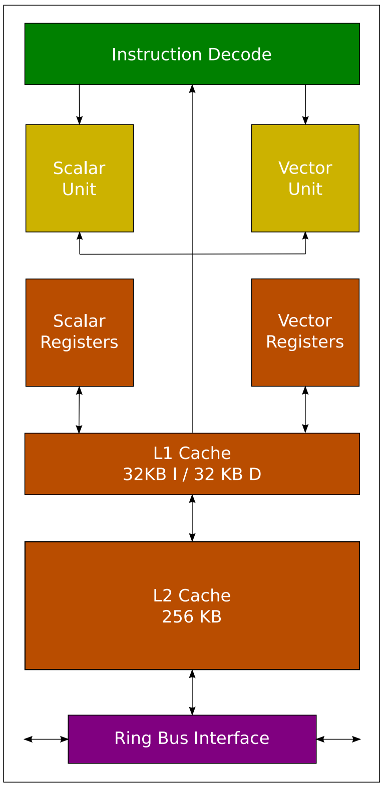


Figure 3.5: Architecture of the Intel Larrabee Core

Intel's Larrabee architecture is based on the Intel Pentium core. In order to increase the number of possible cores on a single die, the Larrabee core is based on a 45nm process, instead of the original 600 nm (0.60μm) which was used for the Pentium. The Larrabee cores can run 4 threads of execution, where each thread has its own set of registers, unlike the original Pentium, which did not support Simultaneously Multithreading (SMT). It can execute 2 instructions per cycle, and has a pipeline depth of 5 stages. The Larrabee core also features a larger cache than the original Pentium, a total of 64KB

L1 cache, split equally between Instruction and Data (32KB for each), where the Pentium had 8KB, and 256KB L2 cache, whereas the Pentium relied on a optional external cache on the motherboard. An overview of the Larrabee Core is shown in figure 3.5. The biggest change from the original Pentium core is that the new Larrabee core has a SIMD unit. A Larrabee core vector ALU is 16 data elements wide, whereas the original Pentium architecture did not have any SIMD unit, except for the later Pentium revisions which supported MMX instruction set [32].

Larrabee does not have as many cores as the NVIDIA and AMD Streaming Processors architectures, so it relies on its wide vector ALU and high number of registers to yield high performance. Initial versions of Larrabee are estimated to have between 8 and 16 cores [33].

Larrabee will not be a GPU in the traditional sense, but rather a set of x86 compatible cores that can also be used for general purpose programming. However, Intel will make DirectX and OpenGL front-ends for Larrabee, to enable it to work like a GPU.

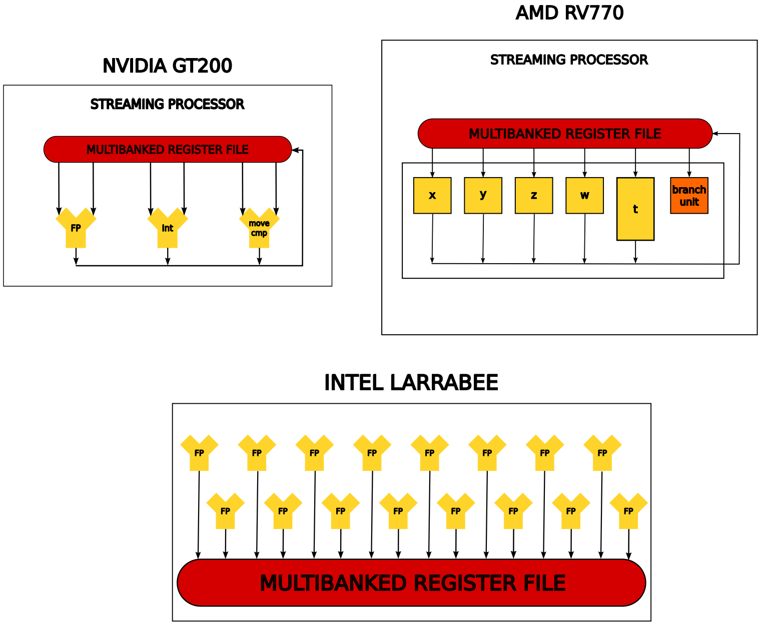


Figure 3.6: Overview of the Intel, AMD and NVIDIA cores

An overview of the Intel, AMD and NVIDIA SIMD architectures is given in figure 3.6, showing the different architectures. The AMD and NVIDIA cores look more similar the the Larrabee. However, they will differ widely in instruction set and other properties. The NVIDIA SP can process a single operation at the time. AMDs SIMD core can process five. Intels core can process sixteen. AMD already have challenges scheduling five operations at the time, a lot of compiler help is needed to use the core efficiently, so we might see some challenges for Intel utilizing the vector ALU to its full capacity.

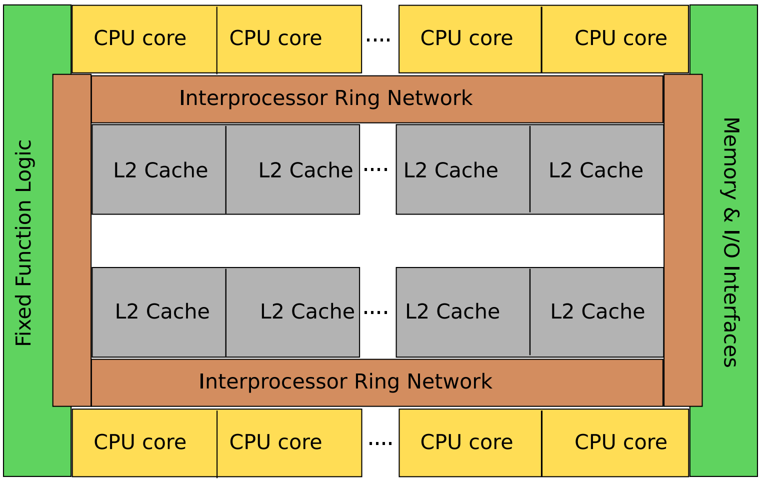


Figure 3.7: Overview of the Larrabee architecture

Intel's Larrabee Cores will be connected by a ring bus architecture. The quite large amount of cache available on the Larrabee cores is expected to be one of Larrabees strengths, as each Larrabee core has four times the L1 cache of the original Pentium. The reason for this is that the core can work on four times the number of threads as the Pentium. The Larrabee cores will each have 256KB L2 cache. These caches on the Larrabee are fully coherent. Communication between the L2 caches is handled by the ring bus architecture seen in figure 3.7. If a core cannot find the desired data in its L2 cache, a request will be placed on the ring bus, and if one of the other cores L2 cache has it, it will be transferred back to the correct cache.

Even though the Larrabee architecture support the x86 instruction set, several features from the Intel Core 2 architecture are not supported. The Larrabee architecture does not support out-of-order execution and branch prediction. It does, however, have a larger number of cores, which will mean that a different programming model must be used to yield good performance. This might result in some of the same challenges that we encounter while programming the NVIDIA and ATI GPUs, explained further in chapter 4.

3.5 Summary

In this chapter, we have looked into some of the different GPU architectures that are or soon will be available. We have seen that the GPUs hardware will give great possibilities for programming and porting applications to the GPUs to give increased performance than running on the CPU. One of the most important differences between the architectures is the vectorization required to efficiently use the hardware. We will look into how GPU is programmed, by looking at one of the frameworks available for programming GPUs, NVIDIAs CUDA framework in the next chapter, in order to give us the possibility of investigating further how an application is ported to run on the GPU. We will also briefly discuss alternatives to NVIDIAs CUDA framework.

Chapter 4

Compute Unified Device Architecture

4.1 Introduction

We have chosen to further investigate the NVIDIA GPUs, because it has shown good potential, and we have seen many example applications that use NVIDIAs GPUs. In this chapter we investigate which possibilities the NVIDIA Compute Unified Device Architecture (CUDA) [34] gives us.

Of the three architectures we looked into in chapter 3 we choose to investigate the NVIDIA GPUs and therefore the CUDA framework, because of the easy access to documentation, and a programming model which was not to complicated to comprehend. We felt that other architectures were not as well documented, nor as widely used as the NVIDIA architecture. This chapter will focus on the NVIDIA programming model, however, we will also look into some other programming models for GPUs. The NVIDIA programming model was the only mature GPGPU framework available for our platform when we started this thesis.

An overview of CUDA, which gives a introduction to how CUDA is structured is shown in figure 4.1. The CUDA architecture consists of several components, one through four, which we will discuss in this chapter.

1. Parallel Compute engines in the GPU
2. Kernel level support
3. User mode driver
4. PTX instruction set architecture

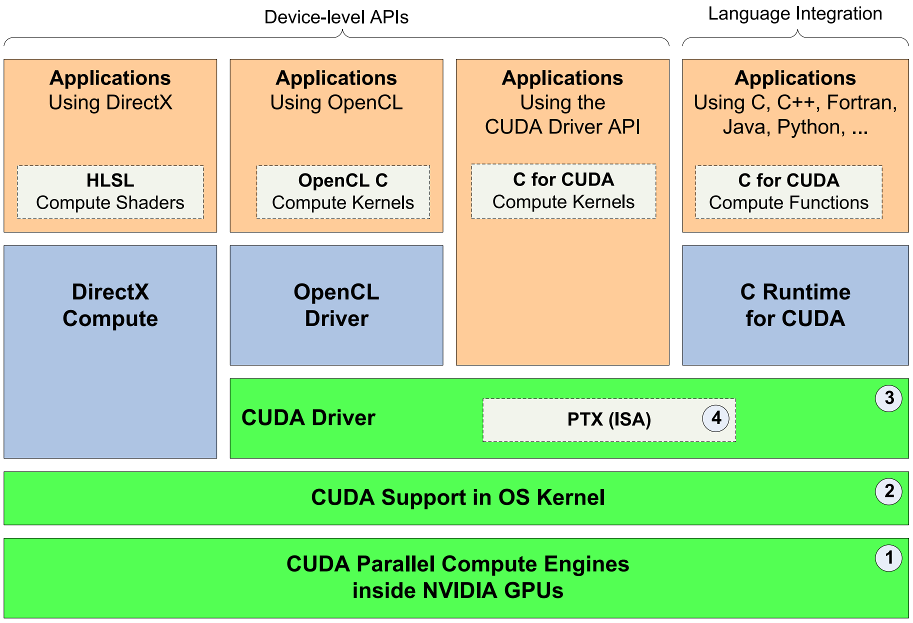

Device-level APIs
Language Integration
Applications Using DirectX
HLSL Compute Shaders
Applications Using OpenCL
OpenCL C Compute Kernels
Applications Using the CUDA Driver API
C for CUDA Compute Kernels
Applications Using C, C++, Fortran, Java, Python, ...
C for CUDA Compute Functions
DirectX Compute
OpenCL Driver
C Runtime for CUDA
CUDA Driver
PTX (ISA)
4
3
CUDA Support in OS Kernel
2
CUDA Parallel Compute Engines inside NVIDIA GPUs
1

Figure 4.1: CUDA architecture overview (From NVIDIA [5])

4.2 Programming CUDA

The NVIDIA CUDA technology consists of an extension to the C programming language, and a compiler that understands the extended programming language. The language C is extended in such a way as to enable programming on GPUs without changing the way of writing computer programs. It also contains libraries for Fast Fourier Transform and Basic Linear Algebra Subroutines. A key part of CUDA is the driver, which enables computing on the GPU.

CUDA gives a number of advantages by allowing to program the GPU with the C programming language directly, instead of using a Graphics API. CUDA also supports scattered writes are supported, meaning you can write to arbitrary GPU memory addresses. The framework also gives full integer support.

The CUDA solution is proprietary, and runs on Windows (32/64bit) and Linux (32/64bit) as well as Mac OS X. The CUDA SDK was initially released to the public in February 2007. CUDA requires a modern NVIDIA graphics card with a unified shader architecture. The first card who had this capability was the GeForce 8.

CUDA enables the programmer to write programs almost as standard programs, except that some syntax is different, and that you program a lot more threads have to be considered, for optimal performance.

Which parts of the CUDA library are supported on a GPU is determined from what

Compute capability the GPU has. For example, a G80-based GeForce 8800 GTX has *Compute capability 1.0*, while a G92-based GeForce 8800 GT has *Compute capability 1.1*, meaning that more of the CUDA library is supported. Even though the 8800GTX card is better in some ways, by having more stream processors, it lacks some of the features that the CUDA API offers, like for instance atomic memory operations. The latest version as of now is CUDA version 2.0, where some of the newest features requires *Compute capability 1.2*.

CUDA abstracts away some of the low level programming, where programmers need to use a assembly language for programming. This means that NVIDIA can make changes to the architecture and instruction set without having to rewrite applications. This is different to how the Intel instruction set is carved in stone. It cannot be changed because of the many applications that are written partially in Intel assembly code, meaning they could potentially stop working. It can, however, be extended with new features (like MMX [35] and SSE [36]).

The program that executes on the GPU is called a *kernel*. This kernel is invoked from CPU code. The kernel can call other functions that are declared as *__device__* functions, but not functions running on the CPU. The kernel does not return any value. When invoking the kernel, the number of *blocks*, and threads in each block must be declared. A block is a group of threads that are executed together, and have synchronization possibilities. A *grid* is a number of thread blocks which must be completed before the next grid can start.

Only limited synchronization is available, which can be a problem when writing applications. Threads can synchronize internally in a block, between blocks however, no synchronization is available. The CPU code can wait until all threads are synchronized, so in a way it is possible to synchronize between the blocks, but only with help from CPU code. The synchronization is done with the *__syncthreads()* function. This is a barrier function, meaning that all threads in a block must reach this point, before they can continue.

The main system memory is not available from the GPU threads. Therefore, data must be copied to the GPU for processing, and back again when the data is processed. This is one of the challenges for GPU programming. Even though the memory bandwidth and speed of the GPU is high, transferring enough data over the bus can be a challenge. The bus between the GPU and CPU is PCI Express. For example, the PCI Express version 1.1 has a capacity of 250MB/s in each lane. A graphics card normally has 16 lanes (called PCI Express 16x), which gives a data rate of 4GB/s. The latest version of the PCI Express standard, version 2.0 doubles this data rate.

.cu	CUDA source file
.cup	Preprocessed CUDA source file
.c	C source file
.cc, .cxx, .cpp	C++ source file
.gpu	Gpu intermediate file
.ptx	Ptx intermediate assembly file
.o, .obj	Object file
.a, .lib	Library file
.res	Resource file
.so	Shared object file

Table 4.1: nvcc supported files (From NVIDIA [12])

4.3 The compiler

The CUDA compiler is called *nvcc*, and is based on Open64, which is an open source compiler originally designed for the IA-64 Intel architecture. NVIDIA chose Open64 because of the strength of its optimizations [37], although customizing the gnu compiler collection (gcc) was also an option. It uses gcc for compiling the programs. The nvcc compiler hides some of the details of compilation. We will, however, discuss some of the details here.

The *nvcc* compiler can compile regular .c files, which it just passes directly to gcc for compilation, as these files contain only standard C. nvcc also deals with .cu files, that are made in the extended C language which is used in CUDA. .cu and .cup files, that include both host code and device functions. It is therefore possible to use pure C code directly in CUDA, without having to change anything. This code will then run as normal on the CPU and not on the GPU. In case of running nvcc on a Windows platform, code compilation is not always forwarded to gcc, but for example to the Microsoft Visual Studio C compiler.

The nvcc compiler also supports generating an emulation version of the code. This means that the code will not actually run on the GPU. This emulation code can be useful for testing, as it is easier to debug, for example with a debugger like gdb [38] or with printing data to the screen, neither is possible when running on the GPU. The emulation mode is, however, limited because it does not run the threads in parallel, but in sequence. This implies that a lot of errors can go past without being noticed in emulation mode. It also means that if one thread depends on the outcome of another, it might stall, because the other thread will not run until the first has completed its execution. Emulation mode is often slower, so testing code in emulation mode might take more time.

Cubin are a file format the compiler generates as intermediate files, and it can be instructed to keep these files for inspection. In these files we are able to see some information about the code. Especially interesting are the shared resources required by each thread, like shared memory used for the parameters, and the number of registers needed per thread. These numbers can be interesting in explaining performance numbers.

4.3.1 Compilation stages

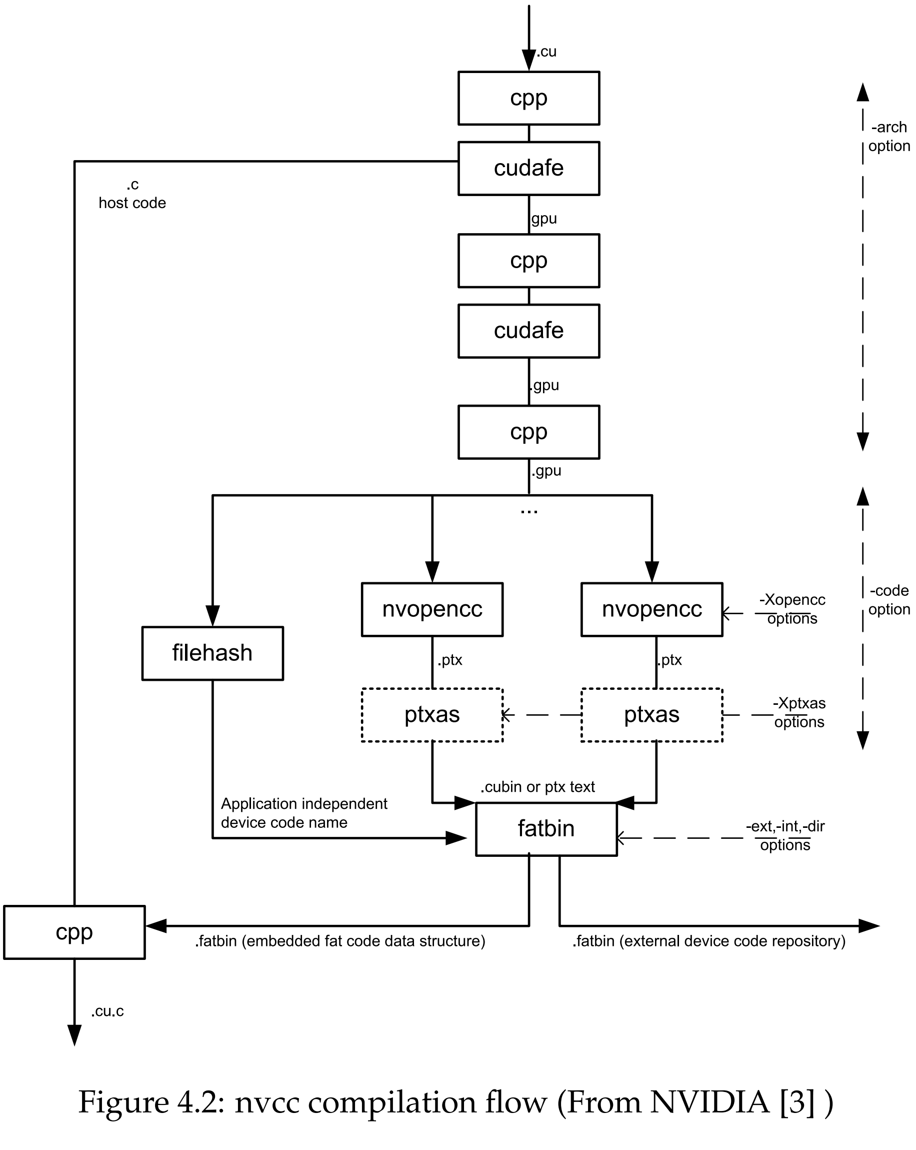


Figure 4.2: nvcc compilation flow (From NVIDIA [3])

The stages of the nvcc compilation is shown in figure 4.2. The input program is separated by nvcc into a gpu part, and the part that is running on the host code. The part of nvcc responsible for this is called *cudafe*. This code is then processed further and generates cubin and ptx (device only) code files, while the host-only code is forwarded directly to gcc. Cubin are CUDA binary files, while ptx are intermediate files, seen from table 4.1. The code is assembled and a *fatbin* is generated which is merged with the previously separated host code. A fatbin is a binary with all code that needs to be run on the GPU embedded.

Preprocessing is done multiple times, first the .cu files, then later for expanding CUDA macros. The last preprocessing step merges the files back together. Cudafe can be used for preprocessing, which hides some output.

4.3.2 Runtime

The runtime is the part of CUDA that executes programs on the GPU. This has support in the driver, which is mandatory for CUDA. The runtime does memory management, and shuffles data between the main and GPU memory in the system.

4.4 Performance

CUDA performance is often measured in how well the GPU is utilized. NVIDIA offers a spreadsheet for calculating how well the GPU is utilized, giving the developer a metric for calculating efficiency. NVIDIA has also provided a profiler, which gives the developers a possibility to optimize their applications.

When optimizing CUDA applications, there are certain things one should consider. A parallelizable algorithm is beneficial where many threads have to be able to work on the program without exchanging data with each other (thus avoiding too much waiting). One also has to think about memory accesses. When many threads try to access the same memory the memory can be locked, and the threads forced to wait, thus degrading the performance. To get high utilization on the GPU, many threads should be used. Instead of having few threads doing a lot of work, one should have many threads doing less work. Loops should be unrolled if they do not have too many iterations. This can be done either manually, or with the pragma statement for the CUDA preprocessor, which can unroll loops for you. The programmer can write *#pragma unroll 5*, will unroll the loop to five iterations. This means the loop must be done in a multiple of 5 times, to yield correct results. Divergence in a single warp should be avoided, branching should preferably be done for the whole warp.

One of the most important places where we can optimize is how we use the memory available on the GPU. We have seen in chapter 3 that we have several types of memory available: shared, registers, constant, texture and uncached local and global memory. The shared memory can be viewed as a sort of used managed cache, and Selberstein et al. have investigated this idea [39]. This memory is shared between all threads in a block, and is as fast as the registers when the access pattern is correct. The access pattern should avoid multiple threads trying to access the same *bank*. A bank is a 32-bit part of the memory. Shared memory should be preferred, because using to many local variables in the code means that some of these variables can be placed in the uncached global memory, and this can severely impair the efficiency of the program. To see if variables are placed in global memory one can inspect the assembly code from the compiled. Texture and constant memory are read-only, but can be very efficient for lookup tables and other memory that should not be edited.

The NVIDIA CUDA Visual Profiler, is a tool from NVIDIA which can profile applications executing on the GPU. It can plot and write out tables on the performance of the applications running, listing number of calls, GPU time, CPU time, and other information. It can also show the occupancy of a program running on the GPU, meaning how much of the time the multiprocessors are executing code, and not waiting for memory accesses.

Ryoo et al. have investigated optimization principles for CUDA programming [40]. They found that optimizing various applications gave between a 1.16X and 431X total application speedup, showing some of the potential of optimizing programs running on the GPU.

4.5 Debugging

NVIDIA has released a debugger called CUDA-GDB [41]. It is based on GNU GDB [38]. This debugger as of writing only supports Linux, and is therefore has a more limited user base. It's goals are to make debugging CUDA applications easier than it was earlier, and to present developers with a well known debugger. It requires a compiler flag for compiling and will only work when code is compiled without optimization. It gives the possibility to pause CUDA execution at any function symbol or line of code, like normal GDB. Because all device functions are inlined, it is not possible to step over a function.

4.6 Other technologies

There are several other technologies for programming GPUs. We look into Shading languages, Stream SDK, BrookGPU and OpenCL.

4.6.1 Shading language

The first method of programming GPUs was basically just to use the shading language available in OpenGL [42] and DirectX [43]. This requires knowledge of OpenGL/DirectX and meant you had to use a lot of time implementing, not being able to focus on the algorithms [44]. Available shading languages include Cg, HLSL and the OpenGL shading language.

4.6.2 Stream SDK

Close to Metal (CTM) was ATI's (and now AMDs) alternative to CUDA [45]. It is now called Stream SDK [46]. Stream SDK is available for Windows XP (32/64bit) and Linux(32/64bit).

CTM was originally intended to be more low level than CUDA, but was changed when the name changed to a more high level platform. Stream SDK still gives access to a low level programming model, through the Compute Abstraction Layer (CAL). Libraries included in Stream SDK are the AMD Core Math library (ACML), which provides access to mathematical functions. The AMD Performance Library (APL) is also included and a video transcoding library called COBRA. The latter assists in video transcoding, which is a computationally intensive task, and has been successfully offloaded to the GPU.

4.6.3 BrookGPU

BrookGPU [47] is developed at Stanford University. It is a variation of the Brook programming language, which is an extension to the C programming language. Brook is one of few GPGPU libraries that is licensed under a free license, namely the BSD license. BrookGPU works on Intel, NVIDIA and ATI GPUs. Brook is currently in version 0.5 beta.

4.6.4 OpenCL

Open Computing Language (OpenCL) is Apple's alternative for programming GPUs [48]. This is based on the C standard C99. OpenCL is an attempt to create an industry standard for programming GPUs, and is therefore named in the same way as OpenGL and OpenAL, and also submitted to Khronos Group, which has standardized OpenGL. OpenCL is set to be included in Mac OS X version 10.6 (Snow Leopard). OpenCL is now standardized by the Khronos Group, and has a syntax quite similar to CUDA.

4.7 GPU multitasking

Using the CUDA framework there are some challenges with running multiple applications at once. They share the GPU, but there is currently no trivial way to instruct the GPU on what tasks to run when, and GPU memory is not pageable, so the GPU can easily run out of memory running many applications.

4.8 Summary

We have in this chapter discussed how the CUDA framework works, and briefly discussed other alternatives to the CUDA framework. We discussed NVIDIAs CUDA, which we believe is one of the most mature frameworks for GPU programming, and we therefore wish to further use this framework, when we in our next chapter investigate programming an actual application, to investigate various parts of optimization.

Chapter 5

AES Encryption on CUDA-enabled GPUs

5.1 Introduction

In this chapter, we will investigate the Advanced Encryption Standard (AES) encryption, how it works, and how an ordinary CPU implementation of a AES encryption program can be converted to run on the GPU.

AES is an important part of the encrypted file system we will investigate in chapter 6. Before we can investigate offloading the CPU intensive encryption in this file system, we first look into some different implementations for encryption on the GPU, to see what kind of performance we get from a GPU implementation, compared to a CPU implementation.

AES is also an algorithm which is massively parallelizable, which we will see in this chapter. In will addition to be an important part of the file system we will investigate, serve as an example of an application that is ported from running on the CPU to GPU, using the CUDA framework. We will also look into a variety of optimization techniques, where AES also is a good example, but these techniques can be applied in a variety of applications.

5.2 Advanced Encryption Standard

AES [49] is a block cipher encryption standard, designed as a successor to the Data Encryption Standard (DES). It was designed to be easy to implement in hardware and software, and requires little memory. The AES algorithm was designed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and submitted to the AES selection process under the name of Rijndael.

AES encryption consists of several steps, mentioned here, with the individual sub steps described later. Decryption of data is achieved by doing all the stages in reverse order. Some of the items in the list are repeated several times, in what is called rounds. Note that there is no MixColumns in the last round. The number of rounds in the encryption process is based on the key size seen from table 5.1.

- KeyExpansion
- AddRoundKey (Initial round)
- SubBytes (Rounds 1 to N-1)
- ShiftRows (Rounds 1 to N-1)
- MixColumns (Rounds 1 to N-1)
- AddRoundKey (Rounds 1 to N-1)
- SubBytes (Last round)
- ShiftRows (Last round)
- AddRoundKey (Last round)

The definition of a block cipher is that it operates on blocks of data, which in this case are 16 bytes large. The block is called a state during the encryption phase, and can be viewed as a 4x4 matrix. The block cipher takes either a plain text input that yields a encrypted output called a cipher text, which is the same size as the plain text, or a cipher text input that yields a plain text. Without knowing the key the cipher text should be impossible to decode correctly. It can be required to know a initialization vector, as we see in paragraph 5.2.5.

One important thing to be aware is that the AES algorithm is big-endian, so in case we are on a little-endian architecture (like normal Intel-compatible architectures used in desktop-computers and many servers) data must be converted to big-endian before

Key size	Number of Rounds
128	10
192	12
256	14

Table 5.1: Number of rounds given key size

the algorithm is run, and must be converted back to little-endian after the algorithm has been run.

5.2.1 KeyExpansion

This stage converts the encryption key into a set of round keys. This operation is known as the Rijndael Key Schedule. The round keys are used in the later stages to encrypt the data.

5.2.2 SubBytes

In the SubBytes stage all the bytes in the cipher block are substituted with a byte from a substitution box table (sbox). The original value is used as an index in the sbox. The purpose of this is to avoid linearity in the cipher block. The SubBytes stage is seen in figure 5.1.

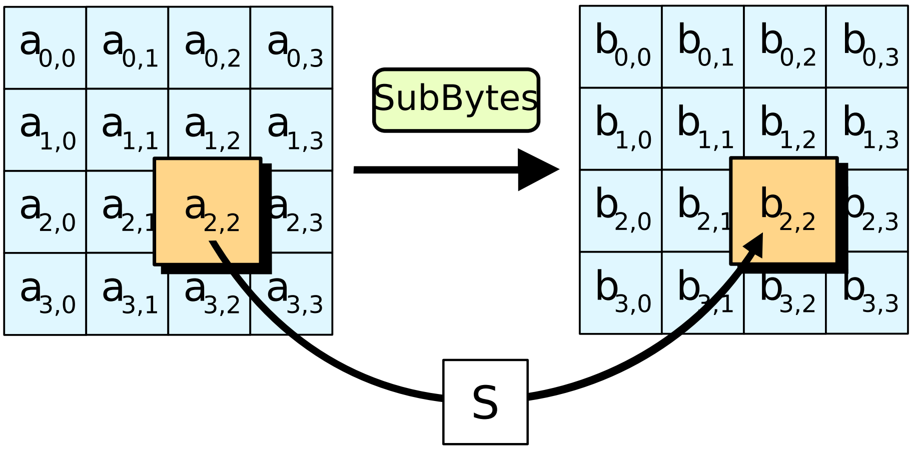

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
SubBytes
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
S

Figure 5.1: The SubBytes step (From Wikipedia [6])

5.2.3 ShiftRows

The ShiftRows stage operates on rows in the cipher block. For 128 and 192 bits keys, it leaves the first row intact, rotates the second row one position to the left, and the third and fourth row two and three positions respectively. For 256 bits keys the first row is left intact, while the second third and fourth are shifted one three and four bytes respectively. The ShiftRows stage is shown in figure 5.2.

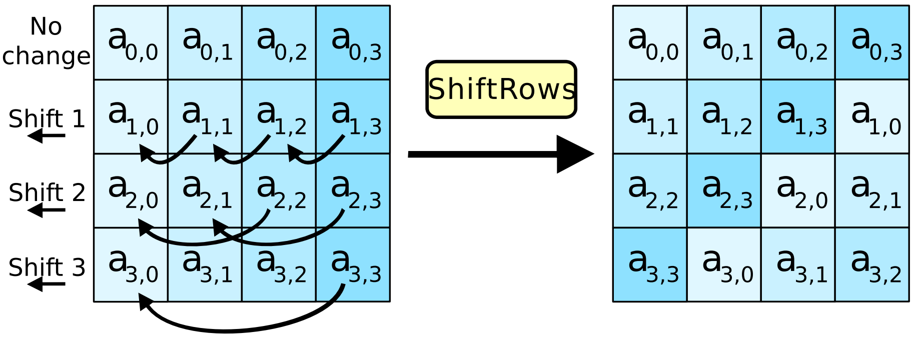

No change
Shift 1
Shift 2
Shift 3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

Figure 5.2: The ShiftRows step (From Wikipedia [6])

5.2.4 MixColumns

The MixColumns stage operates on columns in the state. It multiplies each column of the cipher block with a fixed polynomial c(x) as seen in figure 5.3. The purpose of this stage is to provide diffusion in the cipher block, which means that redundancy in the plain text is dissipated in the statistics of the cipher text.

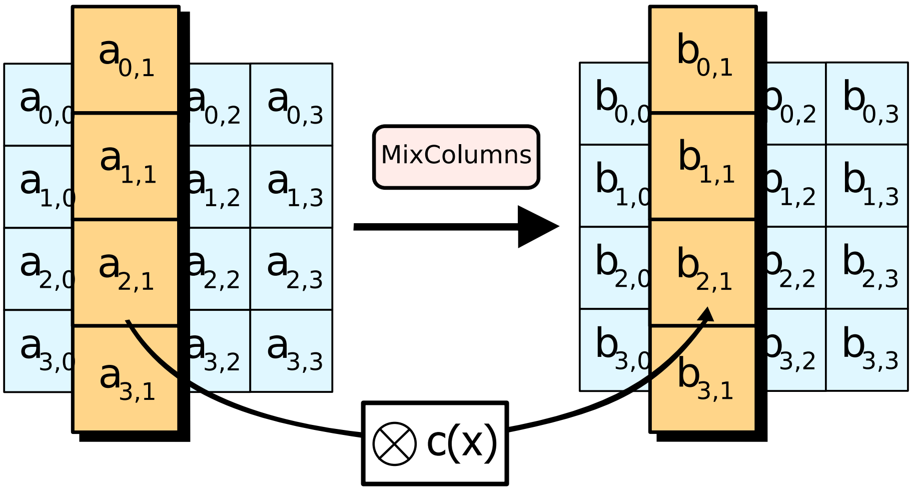

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
MixColumns
$\otimes$ c(x)
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Figure 5.3: The MixColumns step (From Wikipedia [6])

5.2.5 AddRoundKey

The AddRoundKey stage is where the state is combined with a subkey of the round key, to introduce the key in the encryption. The subkey is added by XORing each byte of the state with the corresponding subkey, yielding a new byte for the state.

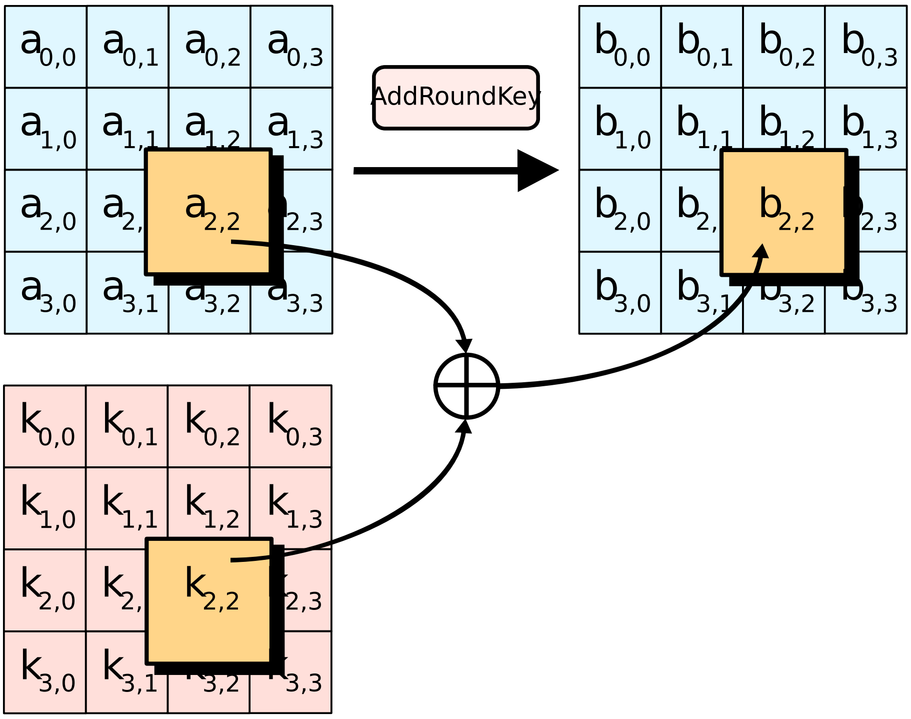

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
AddRoundKey
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$
$k_{0,0}$ $k_{0,1}$ $k_{0,2}$ $k_{0,3}$
$k_{1,0}$ $k_{1,1}$ $k_{1,2}$ $k_{1,3}$
$k_{2,0}$ $k_{2,1}$ $k_{2,2}$ $k_{2,3}$
$k_{3,0}$ $k_{3,1}$ $k_{3,2}$ $k_{3,3}$
$\oplus$

Figure 5.4: The AddRoundKey step (From Wikipedia [6])

5.2.6 Lookup table

It is possible to speed up the execution of AES on systems with 32 bits or wider data types, by combining the SubBytes, ShiftRows and MixColumns into lookup tables. These tables require four 256-entry 32-bit tables, needing a total of four kilobytes. Each round now needs 16 table lookups and 12 XOR operations. It is also possible to use just one single 256-entry 32-bit table by using circular rotates.

5.2.7 Modes of operation

A *Mode of operation* describes how a stream of data is encrypted using block ciphers, because block cipher algorithms themselves only describe what is done with a block of a fixed length. Data lengths are arbitrary, and to have secure encryption one has encrypt blocks differently, even though they have the same plain text input, thus avoiding dictionary and replay attacks. The simplest mode of encryption does not protect against these threats. This mode is called Electronic Cookbook (ECB) and is regarded as insecure. This is the only mode of operation which does not require an initialization vector. None of the modes of operation described in this section provides integrity protection, when they are used for encryption, meaning a change in the encrypted data either by error or an attack can go undetected. They can, however, provide integrity protection instead of encryption.

Initialization vector

A initialization vector (IV) is used to start the encryption process for the first block. The IV must be known for decryption, but does not have to be kept secret like the key. However, for encryption, the same IV should not be reused for encrypting different data, as it can be possible to detect patterns in the cipher text. The length of the IV must also be large enough, to avoid IV collisions, which was a problem for security the Wired Equivalent Privacy 802.11 standard [50]. This made it possible to brute force attack the encryption of Wireless LANs using this form for encryption.

Electronic Cookbook (ECB)

In this mode of operation, each block is encrypted in the same way. This is the simplest mode of operation to implement, and is easy to do in parallel because there are no inter-block dependencies. The encryption simply runs the cipher block algorithm on each block in the data, see figure 5.2.7 for the encryption process, and figure 5.2.7 for the decryption process. This means that two identical plain texts will be encrypted to identical cipher texts. As mentioned it has a severe problem with security. In figure 5.7 we see a picture of the Linux mascot Tux decrypted, encrypted with ECB (in the middle) and how it would look encrypted with any other mode of operation. We see in figure 5.7(b) that it is possible to distinguish the original picture, and this shows that the encryption with ECB in some cases is not sufficient. One should however, note that even though figure 5.7(c) looks random, it is not a guarantee that the encryption is secure.

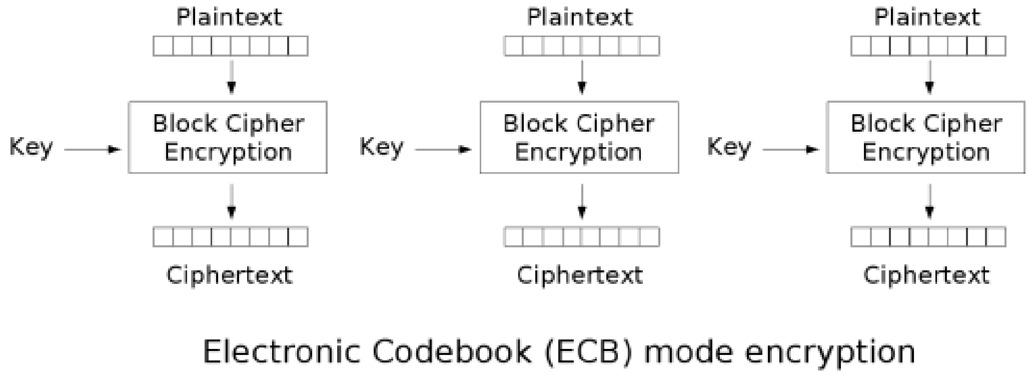


Figure 5.5: ECB Encryption (From Wikipedia [7])

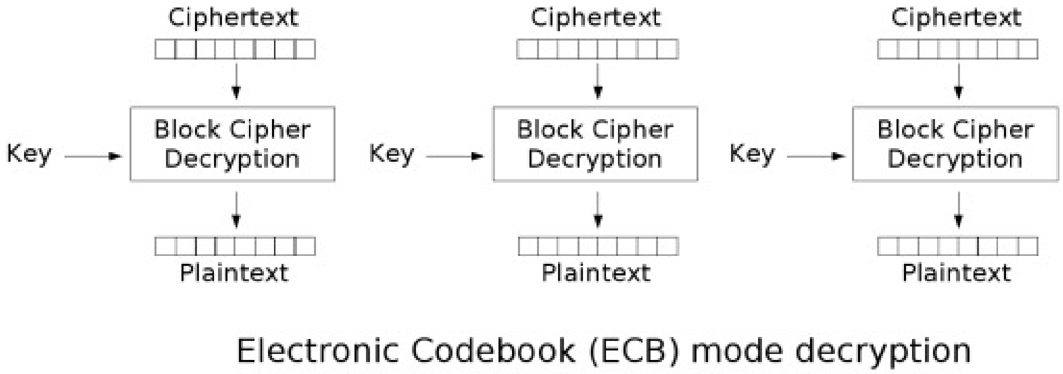
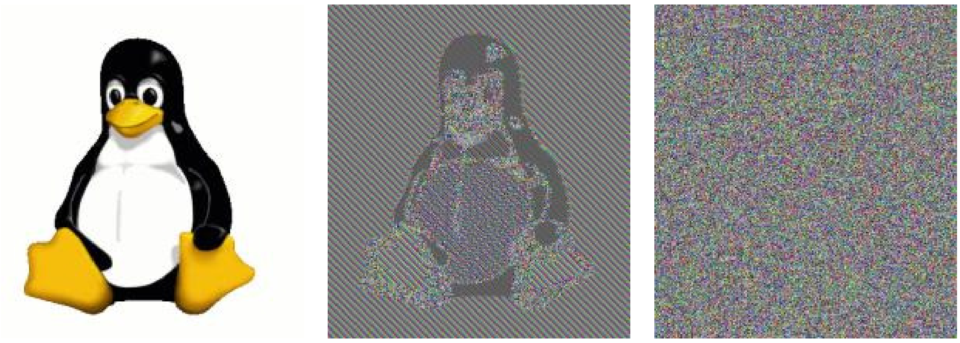


Figure 5.6: ECB Decryption (From Wikipedia [7])

Cipher Block Chaining (CBC)

Cipher Block Chaining (CBC) is a mode of operation developed by IBM in 1976. It is based on XORing each plain text with the previous cipher key before being encrypted. The first plain text is XORed with the IV as seen in figure 5.2.7. This mode of operation is common, but has the disadvantage that its encryption stage cannot be parallelized as efficiently as ECB, because of the inter-block dependencies of the mode of operation. To decrypt the data, each decrypted block must be XORed with the previous cipher key, or the IV for the first block as seen in figure 5.2.7.



(a) Decrypted (b) Encrypted with ECB (c) Encrypted with any other mode

Figure 5.7: Encrypted and decrypted image of Linux mascot Tux (From Wikipedia [7])

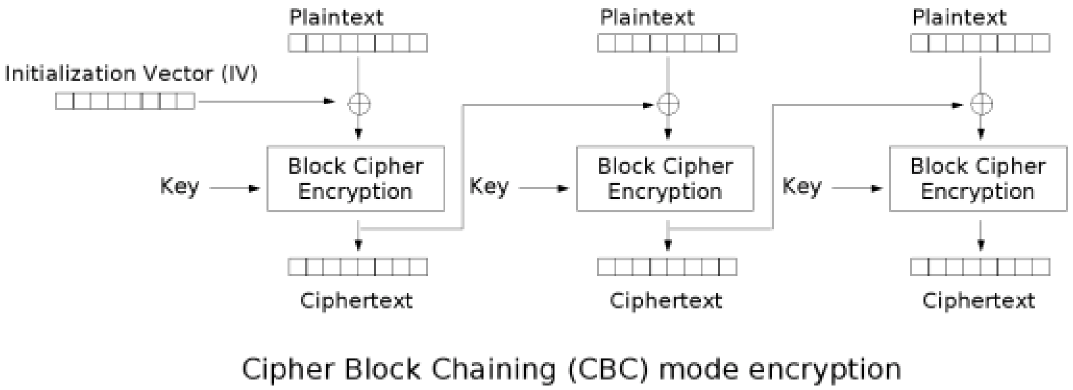


Figure 5.8: CBC Encryption (From Wikipedia [7])

Cipher Feedback

Cipher Feedback (CFB) is somewhat similar to CBC. It first encrypts the IV, and XORs this encrypted data with the plain text to yield the cipher text. It then takes this cipher text, encrypts it and XORs it with the following plain text. CFB have the advantages over CBC that the block cipher is only used in the encryption stage, and that the message does not need to be padded to a multiple of the cipher block size.

Output Feedback

Output Feedback (OFB) is a mode of operation, where the IV is encrypted, and this is XORed with the plain text to yield the cipher text. The encrypted IV is then encrypted, and is XORed with the next plain text, to yield the next cipher text. This is called a synchronous stream cipher. It has the advantages that it allows many error correcting

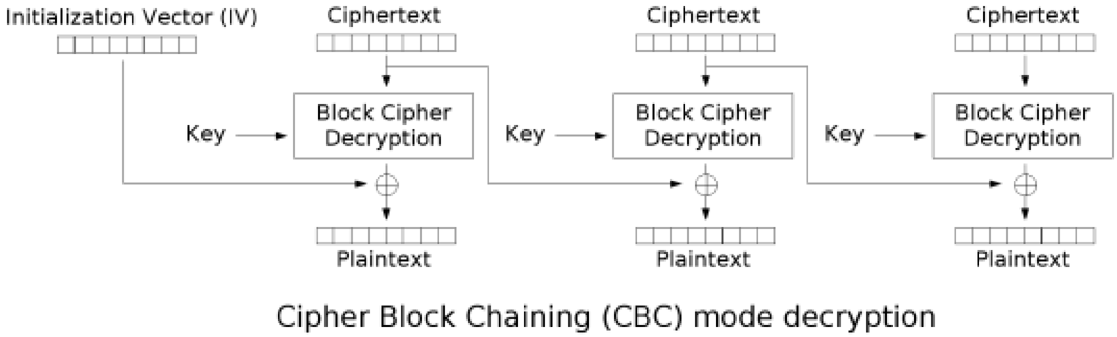


Figure 5.9: CBC Decryption (From Wikipedia [7])

codes to work, because it does not use the cipher text as input for the next encryption.

Counter

Counter (CTR) is also a stream cipher, but works quite different from the other modes of operation. It introduces a new term called *nonce*, that is similar to the IV, but has a changing element called a *counter*, in addition to a static element that remains the same throughout the encryption. CTR encrypts this nonce for every stage, with the counter element increased with one for each cipher block. The encrypted nonce is then XORed with the plain text for each cipher block, and this yields the cipher text as seen in figure 5.2.7. Because of there are no inter-block dependencies, as seen in the figure CTR is fully parallelizable. In figure 5.2.7 we see how a block is decrypted, by decrypting the nonce and XORing it with the cipher text.

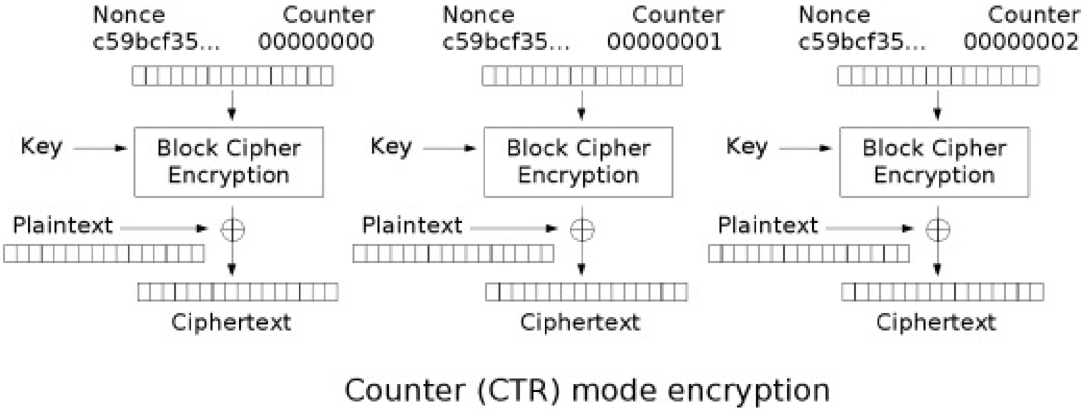


Figure 5.10: CTR Encryption (From Wikipedia [7])

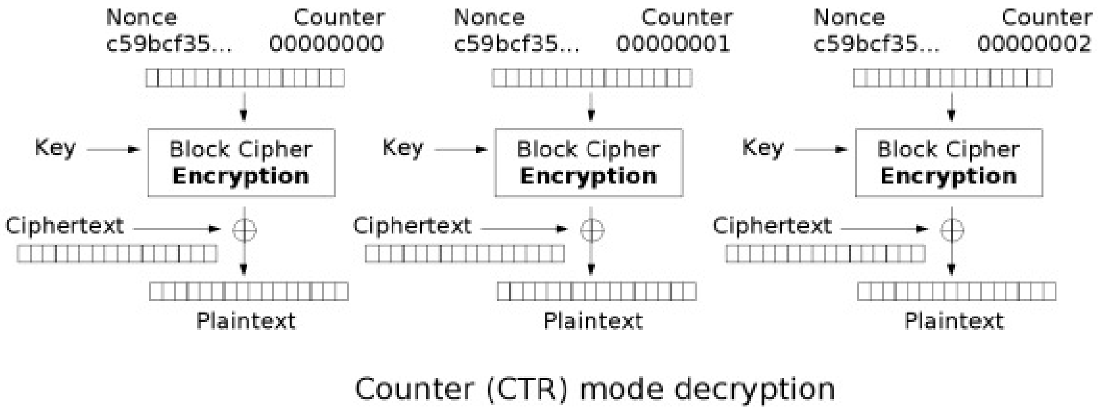


Figure 5.11: CTR Decryption (From Wikipedia [7])

Error propagation

Error propagation is something that happens with the sequential modes of operation, where a bit error in one cipher text will follow into the next cipher text decryption.

5.3 Implementation

We have investigated two different approaches to the AES algorithm on a GPU using the CUDA framework. Both are based on the CTR mode of operation, because of both is security and its possibility to be parallelized. Both implementations are based on an implementation written for a conventional CPU. We also want to show that it is possible to get higher efficiency porting these applications to a GPU.

We also give a background on programming CUDA, and will look into some examples of code to see how it differs from traditional programming.

5.3.1 Standard Implementation

Our standard implementation is a implementation, written by directly following the AES encryption standard [51], without focus on optimization. It is based on a implementation aimed for beginners wanting to understand how the AES algorithm works [52]. This implementation is well documented, and does not focus on speed. We decided to use this implementation to give an introduction to CUDA programming, and also show how we can get performance benefits from porting this application written for a traditional CPU to a GPU-assisted program for encryption/decryption of data.

We first re-wrote the application to use a CTR mode of operation, for high security combined with the possibility of parallel execution of cipher block encryption. We also wrote a simple threaded implementation of this code for use on CPU, so we could compare the GPUs with traditional multi-core CPU architectures. The multithreading for CPU was implemented with the POSIX threads (pthreads) library. It ran one thread per core, and divided the encryption job into equally sized parts, after reading the data for encryption into memory, it then wrote the result back to file. This is not an optimal way of multithreading of this application, because it performs both reads and writes to/from the file single threaded, but we wanted to make it more similar to how the GPU implementation was done.

A lot of the work with this implementation was making it thread safe, because it used several global variables, which is not well suited for parallel execution. When this work was completed, it needed only minor adjustments to run on the GPU. By itself, this did not give the speedup we wanted, so we did further work to increase the efficiency. We stored the round key and nonce in texture memory, which is cached. We reasoned that this would be beneficial because of frequent accesses to the round key. The nonce, however, is only accessed once, so it might not be as important to the performance. We also used constant tables for the sbox, which are not changed, but accessed frequently. This memory is also cached. Last we used shared memory for the state matrix. Because the state matrix is not shared between threads, we allocated shared memory for one state per thread, and accessed the shared memory based on the thread index.

5.3.2 Lookup Table Implementation

We also had an AES implementation which uses lookup tables, this implementation reduces the calculations needed to perform AES encryption, because it uses lookup tables with precalculated values for the calculations as described in section 5.2.6. For use on a CPU, this is much faster than the standard implementation, as we will see in the results section. We wanted to see how much of a speedup we could get on a already optimized program, compared to the standard implementation which was initially not optimized at all. This code was originally based on an implementation made by Philip J. Erdelsky [53], modified to support counter mode of operation.

GPU	8800 GTX	8800GT	8800GT-OC	8800GT	GTX 280
Architecture	G80	G84	G92	G92	GT200
Stream processors	128	32	112	112	240
Core Clock (MHz)	575	540	600	660	602
Shader Clock (MHz)	1350	1180	1500	1674	1296
Memory Clock (MHz)	900	700	900	950	1107
Memory Amount	768MB	256MB	256MB	512MB	1024MB
Memory Type	GDDR3	GDDR3	GDDR3	GDDR3	GDDR3
Memory Interface	384-bit	128-bit	256-bit	256-bit	512-bit
Memory Bandwidth (GB/sec)	86.4	22.4	57.6	60.8	141.7
Compute Capability	1.0	1.1	1.1	1.1	1.3

Table 5.2: Hardware specifications for the GPUs we have tested on [13] [1] [14] [15]

We modified this implementation to use different types of memory for better performance. Texture memory was used for the nonce and texture, and shared memory for the cipher block internal state. The lookup tables were stored in constant memory, and because the cache is 8kB per multiprocessor we reasoned that the whole lookup table could be stored in the cache.

For both the standard and the lookup table implementation we implemented support for a arbitrary number of threads. This was done in order to be able to test which effect the amount of threads had to the performance. We read the entire job into memory and divide it into chunks based on the amount of threads we were testing with. The chunks were sized in such a way as to have sufficient blocks for encryption. Because it was hard to calculate the number of blocks and threads correctly to be able to encrypt the data, we had to pad data so that all launches were the same size. We modified the benchmarks in such a way that it accounted for this padding.

5.4 Results

For testing we tested the execution time of the GPU kernel, and compared different GPUs, the two implementations and also compared this to the original CPU implementations run on a few different computers. We also tested how many threads we could run on each implementation and what kind of differences in performance this yielded. We tested the code on two different machines, one with a E6750 2.66 GHz Intel Core 2 Duo processor (2 cores in total), the other with 4 AMD Opteron 8218 2.6 GHz processors (8 cores in total). We also tested the code on the GPUs listed in table 5.4.

5.4.1 Overview

Figure 5.12 show an overview of the different architectures performance, running the lookup table implementation. The X-axis show the number of threads per block, and the Y-axis shows the encryption performance in Mbit/s. Note that the X-axis is not linear, the increments are not always equal. This is partly because we see an interesting dip in the performance when running with 496 threads, which will be discussed later in this section. We see that all but one of the graphic cards are faster then the Intel Core 2 Duo processor, while only three of the cards outperform the 8-core AMD Opteron machine. We can also observe that the overclocked 8800GT card outperforms the high-end GTX 280 card running with 320 GPU threads per block. We believe this is because the lookup table implementation is memory bound, and therefore the higher core clock on the overclocked 8800GT card allows the lookups in the tables to go faster, with this amount of threads per block.

We can also see that for all implementations the performance increases until a peak. We believe this is because this is due to the shared resources available for each multiprocessor, and how it is used optimally. Increasing the thread count means that it is easier for the scheduler to hide away memory latency, because it can easier find a new warp which is ready for execution. However, when increasing the thread count beyond this peek, it seems that there are too few shared resources available for each thread, and some local variables need to be placed in global memory, impeding performance. The sudden peak for the 280GTX card is also interesting, which might be because this new card hits a point in the scheduling where latency for memory accesses to a large degree hidden, because there are always available new threads for execution when a warp has to wait for memory accesses. The later dip in performance might be related to the fact that threads are scheduled in warps of 32 threads, and running with 496, the last warp will only have 16 threads. We see the increased performance later for the 280GTX, because it has more shared resources available on each multiprocessor, than the other cards. The other cards are not able to run more than 384 threads per block, at that point there is not available resources in the multiprocessors to give each thread gets what it needs. It is also interesting to note that the three cards with compute capability 1.1 (8600GT, 8800GT and 8800GT-OC) all have a similar curves, even though the performance is different. This is probably because their architectures are similar, but the number of multiprocessors are different.

In figure 5.13, we see the same measurements done with the standard implementation. These numbers are different, which is due to the fact that this implementation is computationally bound. We see that the line for all GPUs are quite flat, and that we do

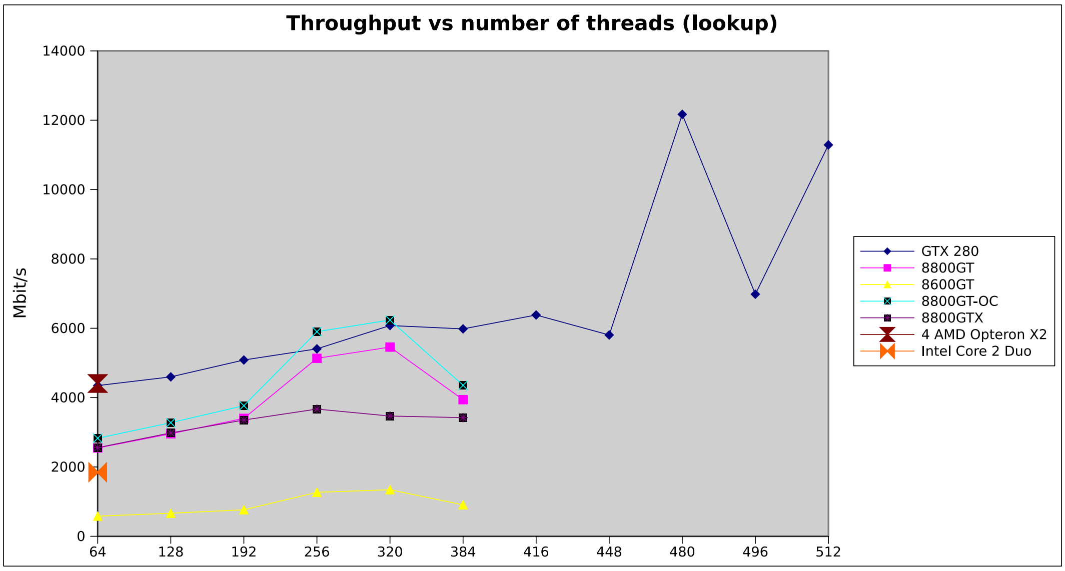


Figure 5.12: Throughput vs number of threads (Lookup implementation)

not get an increase in the performance increasing the number of threads. Because the implementation is computationally, there are always threads available for execution, since we test with minimum 64 threads, and each warp are 32 threads. There should be no memory accesses that needs to be hidden away with increasing the number of threads like in the memory bound lookup table access, because all operations in this kernel works on on chip shared memory.

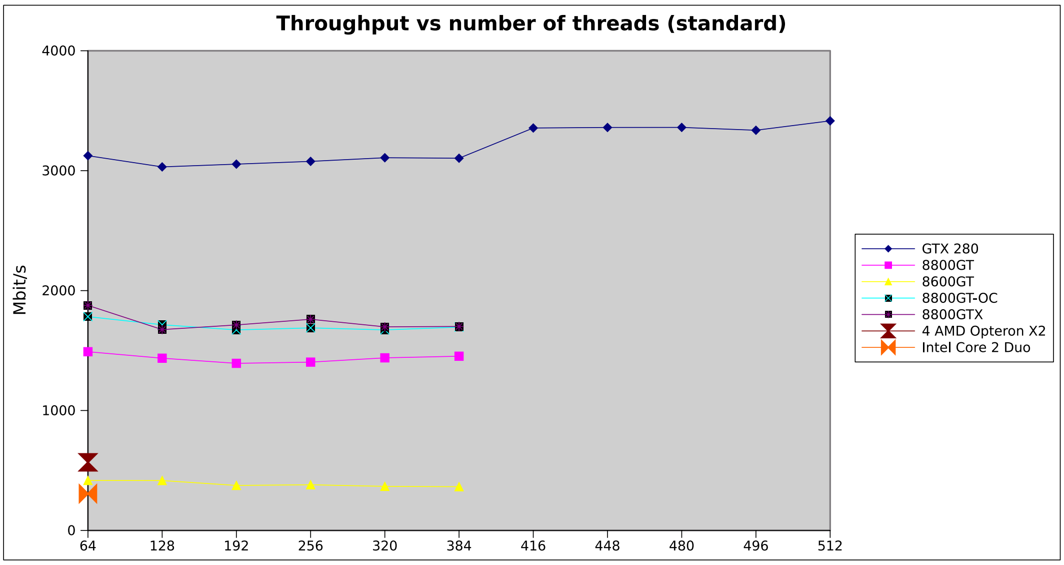


Figure 5.13: Throughput vs number of threads (Standard implementation)

5.4.2 Optimization

We have investigated various optimization schemes to see how we can change memory access patterns to increase the performance. Figure 5.14 shows the performance of the standard implementation with the various optimizations turned on and off, while the figure 5.15 shows the same for the lookup table implementation.

Without optimization means that no optimization is turned on. Without constant memory means that the lookup tables (for lookup implementation) and sbox tables are not stored in constant memory. The reason for using constant memory is because it is cached. There is a 8kB cache for each multiprocessor, which allows all of the tables to be stored in cache. Texture memory is used for storing the round key and nonce used for the encryption. Each element in the round key and texture memory are only accessed once, but the memory is cached, and the prefetching done by the cache will make the memory accesses faster.

We can observe that the shared memory optimization gives a great performance boost for the standard implementation, but not for the lookup table. We believe this speedup in the standard implementation comes from the great number of accesses we have to the state array, that is stored in shared memory instead of local variables in this optimization. Without this optimization the compiler stores the state array in the uncached global memory, because there are not enough free registers in the multiprocessor to store the state array. The standard implementation uses a lot of intermediate calculations, and therefore occupy more registers. This is not the case for the lookup table based implementation, which has fewer members of the state array, 4 instead of 16 because lookup operates on 32bit words and standard operates in 8bit words, and therefore has the ability to place these variables in registers.

5.4.3 Speedup

We have seen that it is possible to optimize AES encryption on the GPU. Figure 5.16 and 5.17 shows the increase in performance we get, running respectively the standard and lookup table implementation on the GPU compared to running the standard and lookup table implementation with 2 threads on a Core 2 Duo test machine.

In the standard implementation it is interesting to note that the relative speedup is much higher than we had with lookup table implementation, which was already optimized for CPU usage. The standard implementation was initially not optimized for CPU, and testing it better shows the possible relative speedup from optimizing, we

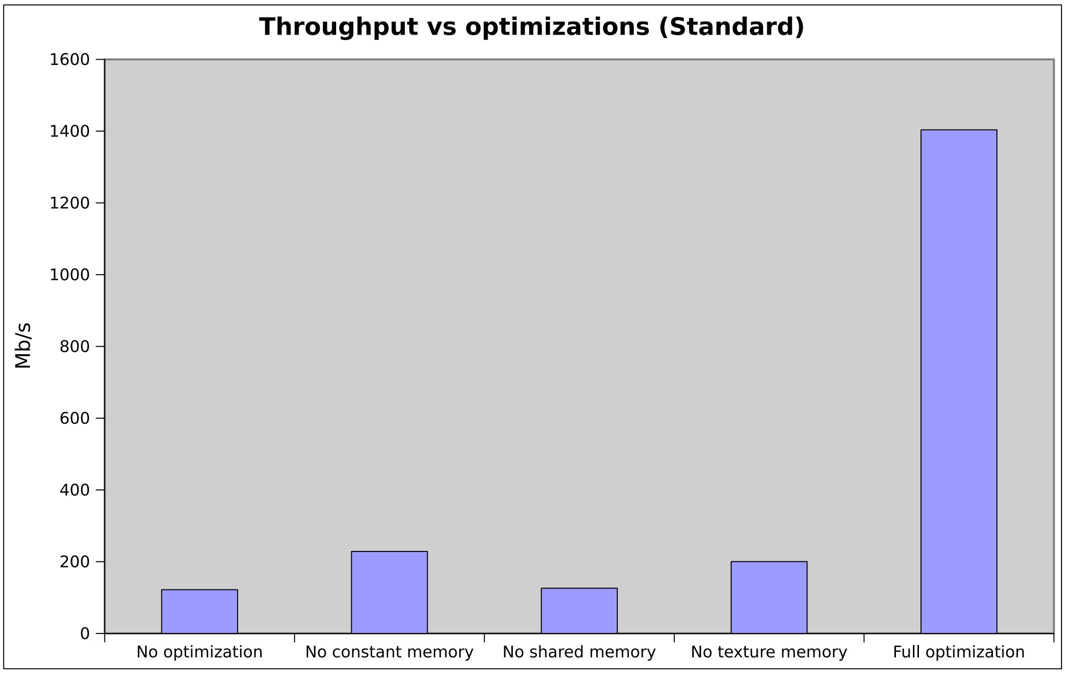


Figure 5.14: Throughput vs optimization (Standard implementation)

have almost a ten times increase in performance. This shows that a computationally bound implementation is easier to speed up on the GPU than a memory bound implementation. Even though we get a better result from the lookup table implementation, the relative speedup is lower. We also can observe that it is possible to increase the performance further on a already optimized implementation because the GPU has different types of memory which we must utilize in order to increase performance.

5.5 Discussion and lessons learned

In this chapter we have tested the core computing time on the GPU kernel, and the part of the CPU code which encrypts the data, not included copying from file to GPU/CPU memory. This is because we have not investigated and implemented an efficient method for asynchronous data transfer to the GPU.

We have learned that rewriting applications to run on the GPU is not the most challenging part, but that the simplest implementation often does not have the desired performance, we learned that one of the most important things on the GPU is to use the memory correctly, finding the correct memory for different parts of the application. We learned that the number of threads per block is a parameter that affects performance.

We have learned that debugging on the GPU is quite challenging, with few debugging

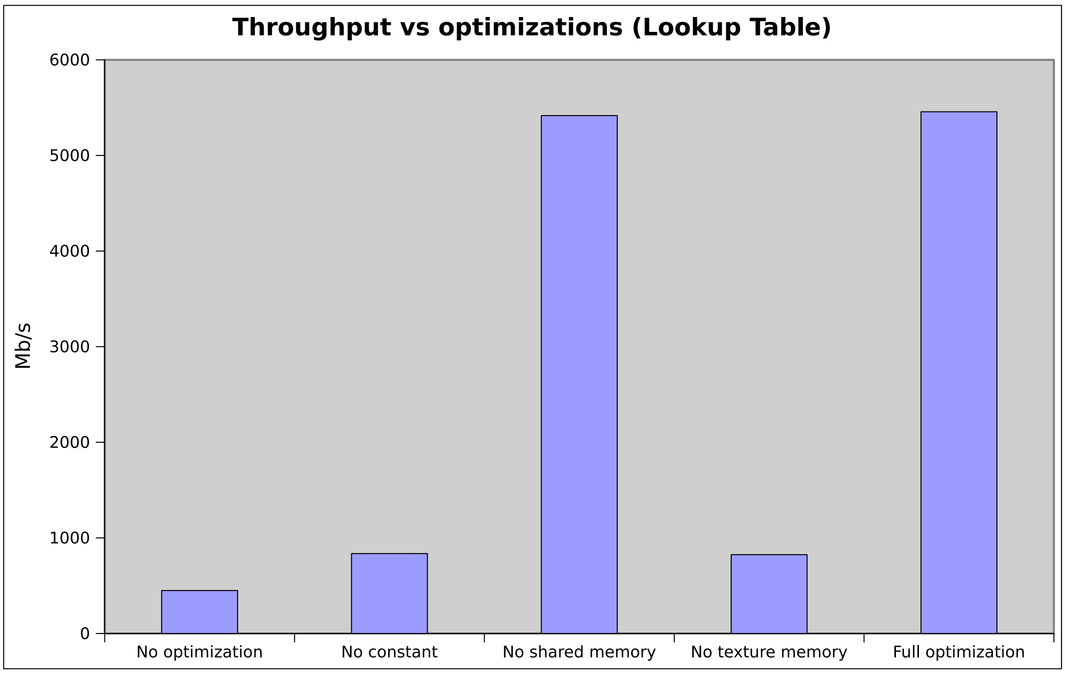


Figure 5.15: Throughput vs optimization (Lookup implementation)

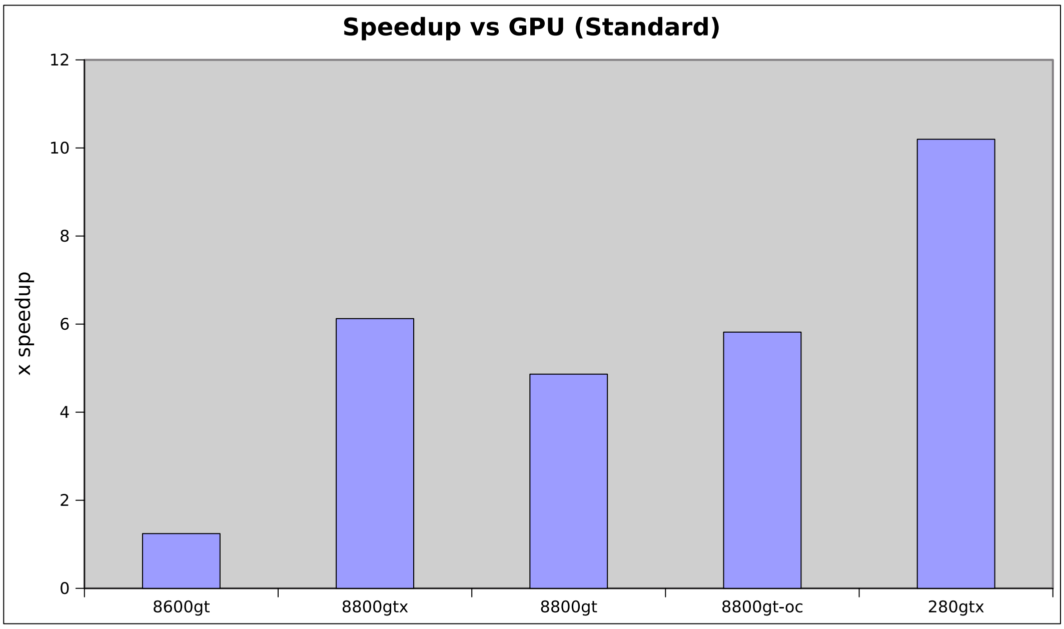


Figure 5.16: Speedup vs GPU (Standard implementation)

methods available. Even though NVIDIA eventually released a version of GDB for the GPU it was not trivial to debug our applications.

Performance gains can be quite large as we have learned when the application is written correctly, with the correct memory types and memory patterns. We see that the access patterns are interesting and it would be interesting to investigate this further.

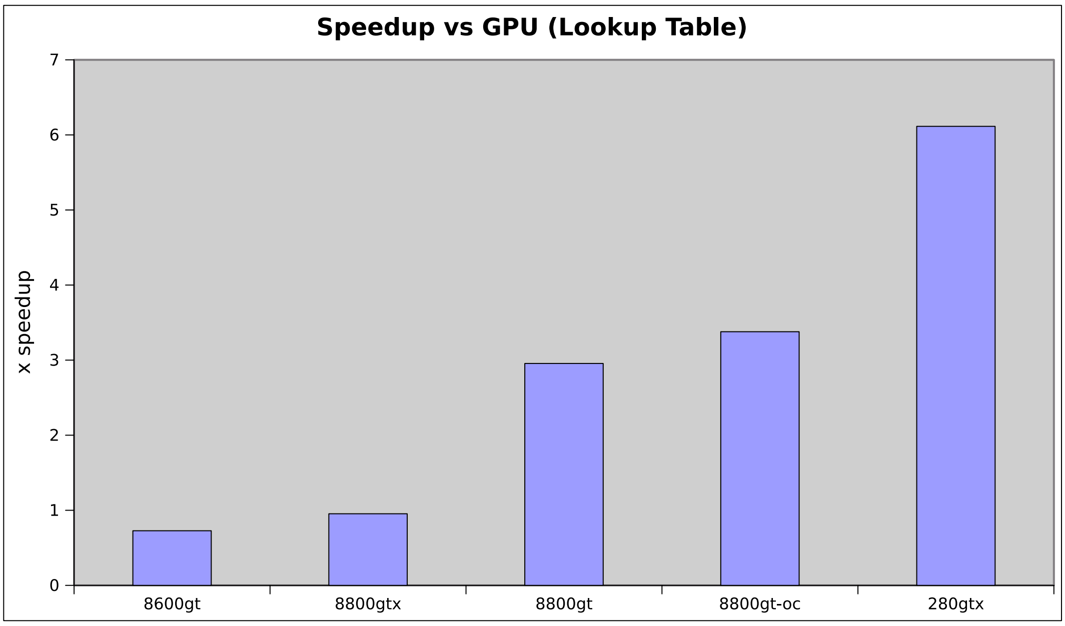


Figure 5.17: Speedup vs GPU (Lookup implementation)

5.6 Future work

For future work we would like to implement double buffering, and have host thread reading from file, one writing from file, and one launching kernels. This would give us an idea of how good the overall throughput of the application is compared to CPU, not only the core computing time.

It could also be interesting to measure what in parts of the kernel the most time is spent. The CUDA Visual Profiler can measure where the most time is spent, however as it default inlines all functions it is not trivial to see where the kernel spends its time. The CUDA Visual Profiler could also give us a occupancy number, which says how high the usage of the multiprocessors are.

We have not currently measured the overhead of launching kernels, which is interesting to know, because it can tell us how much data we should have for each launch of the kernel.

5.7 Summary

In this chapter we have seen examples of two applications parallelized to run on the GPU. We have seen that performance can increase by offloading parts of an application to the GPU. We also have learned that not everything can be ported to GPU, in that only some of the modes of operation used in AES was trivial to run on the GPU. The experiments in this chapter and the implementations can now be used in the next chapter, where we integrate these implementations in a user space file system.

Chapter 6

Offloading of encryption in a user space file system

6.1 Introduction

As security is an important part of modern computing, protecting data against unauthorized access can be useful. One way to do this is to encrypt data, to prevent others from gaining unauthorized access to it. Especially for portable computer, but also other computers that are vulnerable to theft can benefit from encrypting their hard drives data. In this chapter we will investigate an encrypted file system called *EncFS* which is implemented in user space, and investigate which benefits come from offloading some of the encryption process to the GPU.

EncFS is an example of a cryptographic file system. These are specialized file systems designed for security, and encrypt all data including metadata, and are typically layered on top of existing file systems. Other examples of cryptographic file systems are TrueCrypt [54] and CryptoFS [55].

Because the encryption part of the file system is CPU intensive, we will investigate what parts of the EncFS program needs to be modified in order to utilize the GPU efficiently to offload the program in encrypting data.

The EncFS file system is implemented in the *"Filesystem in User space"* (FUSE) system available for Unix-like operating systems. In this chapter we will also go deeper into how FUSE works. We have chosen to base the implementation of the encryption in this file system on the work done in chapter 5, where we implemented the AES encryption algorithm.

We have chosen to base our work on the encrypted file system EncFS, as it is a user space file system. A user space implementation was chosen because it is not supported at the time of writing to run CUDA code from the Linux kernel. We could not find any examples of this, nor anything about it in the documentation. We also felt that we could show most of the same concepts with a user space implementation. A user space implementation is also an advantage, because it can show what benefits other user space programs might get from GPU offloading.

To get some background on EncFS, we will discuss how file systems in user space are implemented, and later some more details about the EncFS file system, before we discuss our implementation.

6.2 Filesystem in User space

User space file systems allows for easier development of file systems, easier testing, and higher system stability. We have chosen to implement the encrypted file system in user space for these reasons.

A common interface for implementing a file system in user space in Unix, is using the FUSE library [56]. FUSE is a system for Unix-like operating systems which allows for file systems to be written in user space, often living on top of other file system, but other abstractions are also possible. Non-privileged users can write their own file systems without touching the kernel code. FUSE provides a bridge between the operating system kernel and user space, and the actual file system runs in user space. The basic structure of FUSE is shown in figure 6.2. Other alternatives to user space file systems are GnomeVFS [57] or KIO [58].

One advantage of FUSE compared to GnomeVFS and KIO is that the file systems with FUSE uses the single-root file system, meaning they are mounted in a folder in the filesystem. This means that applications can be written without being FUSE-aware. A consequence of this is that application can operate on files using the standard file system calls available without knowing the file system is a user space implementation, possibly operating on files stored on an other computer. However, applications might have unexpected failures because they have assumptions about the files they operate on, like always being possible to write, but a network failure might prevent this. As networked file systems like NFS have existed for a long time, this should not be a big problem, many applications have mechanisms to handle these problems.

Another advantage of using FUSE is that file systems can be developed quickly, because it is easy to test new functions, and that they are easy to debug. Choosing a user space implementation does, come at a cost, giving possibly worse performance, because more context switching between kernel and user space can occur. Developing a kernel file system encryption on GPU, would if possible take more time to implement, and we can with the user space implementation still investigate the effects of offloading. Having a user space based file system also means that we get easier debugging. A crash does not mean kernel panic, merely a process that needs to be killed and restarted.

We see that the advantages to using FUSE are many, and that FUSE is well-tested. We have therefore decided to evaluate how a FUSE based encrypted file system will perform, especially in terms of offloading.

FUSE file systems are often virtual file systems existing on top of another file system. The EncFS file system is an example of this, where the encrypted files are stored in an already existing file system. Other types of FUSE file systems are for example SSHFS [59], which uses the SSH protocol to get a remote computers file to appear to be on the local Unix filesystem, GmailFS [60] (Google Mail used as a FUSE file system), and NTFS-3G [61] which allows read/write-access to the Microsoft proprietary NTFS format.

It is possible implement a FUSE filesystem in two different ways, as there are two different interface levels. One is a binary protocol using the kernel module *fuse.ko*. This is a inode based API. It uses C structures for sending commands. There are 14 structure types that can be sent to the kernel module, and 7 structures received. Commands can be for example of type LOOKUP, OPENDIR, RELEASEDIR or similar. These commands must be implemented by the file system. We will however not go into depth of this binary protocol, as the filesystem we work with is implemented with *libfuse*.

Libfuse is a path based API. In contrast to the binary protocol, a simple filesystem can be implemented with a dozen lines of code, where the fuse.ko alternative needs several hundred lines of code. This interface uses C callback functions and has automatic support for threading. The binary protocol using the kernel module is more suited when more control over the implementation is required, it gives greater control, but requires more code to be written to accomplish the same basic functionality. It also has a potential of giving better performance, since it operates on a lower level and has less overhead, as libfuse is built on top of fuse.ko. Using libfuse, the filesystem must implement the different file system calls like *open*, *close*, *read* and *write*. We see how a libfuse call is traced in figure 6.2. The user requests a directory index in the folder

```
int my_readdir( const char *path, void *buf, fuse_fill_dir_t filler, ...)
{
    filler(buf, ".", NULL, 0);
    filler(buf, "..", NULL, 0);
    filler(buf, "example", NULL, 0);
    return 0;
}
```

Figure 6.1: Implementation of readdir function

/tmp/fuse, which is a FUSE file system, and this goes through the Linux Virtual File System (VFS), which recognizes this as a FUSE request, and gives it to the FUSE kernel module, fuse.ko. The kernel module knows that this is a libfuse-based file system, and gives the request to libfuse, which returns the file *hello*.

Example of a C callback function is seen in the *my_readdir* function in figure 6.1, which is an implementation of the system call *readdir*. This function lists all files in a directory. It uses the *filler* function to store the files in the buffer *buf* which is then presented to the application calling readdir on a directory.

The *my_readdir* function will be called for example when the user gives the command ls in a directory (actually when the *readdir* system call is issued). It will show three files in the directory, the usual current directory *"."* and parent directory *".."* and a file called *"example"*.

6.3 EncFS

EncFS [9] is an encrypted file system implemented in user space. It is a virtual file system which runs as a user space process. This means we does not have to trust the storage medium, for anything else than to be reliable. We do not have to depend on it to keep our data private, as this is done by the virtual EncFS file system. This separation of trust is shown in figure 6.3. The storage medium can be basically anything, like an NFS mount, a local disk or a USB flash drive. EncFS is a large file system project, measured in lines of source code, it is as big as the ext3 file system, with about 10 000 lines of source code.

EncFS relies on the well-known OpenSSL library [62] for encryption of the data. This

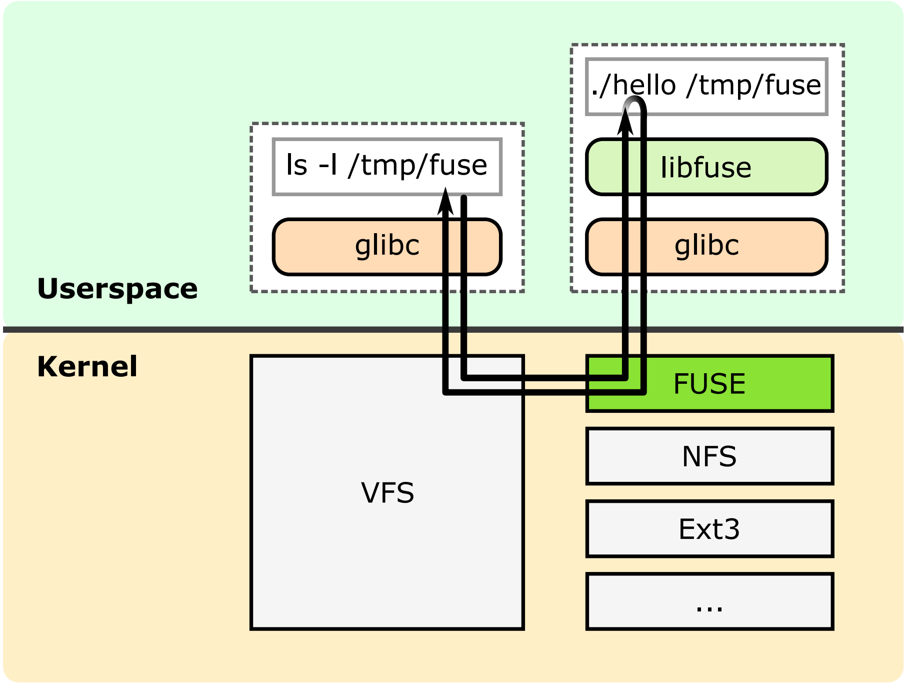


Figure 6.2: FUSE structure image (From Wikipedia [8])

library provides amongst other AES and Blowfish encryption, which EncFS offers to the user as available encryption algorithms.

EncFS works by storing all its encrypted files in a directory used as the file system, and when mounted by the user, it will, given the correct pass phrase, create a virtual directory and decrypt and encrypt data on demand, reading and storing in its data in the file system. The file system also contains some metadata, such as the algorithm, key size and block size, which is stored in a XML file.

Figure 6.4 shows the structure of the EncFS program, and a bit more how fuse is structured, which we will go briefly into here.

The callback layer is called by libfuse. It has implemented the FUSE specific functions required for a FUSE filesystem, like open, close, read and write, and it passes the calls to these functions on to specific classes in libencfs depending on what type of operation it is.

EncFS uses a randomly generated key, which is called the volume key. The volume key is encrypted using a password given by the user. The advantage of this approach is that we can easily change the password used to access the file system.

Originally, libencfs uses OpenSSL to encrypt data, and our modification will first have to be to replace the usage of the OpenSSL library with our own AES GPU implemen-

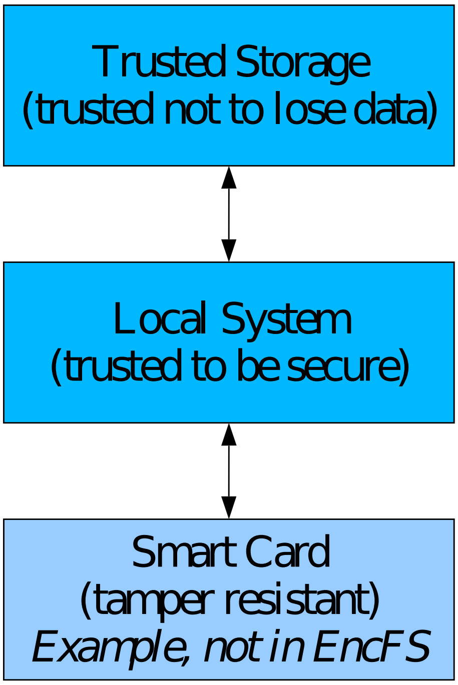


Figure 6.3: Separation of trust with encrypted file systems

tations, as shown in figure 6.4.

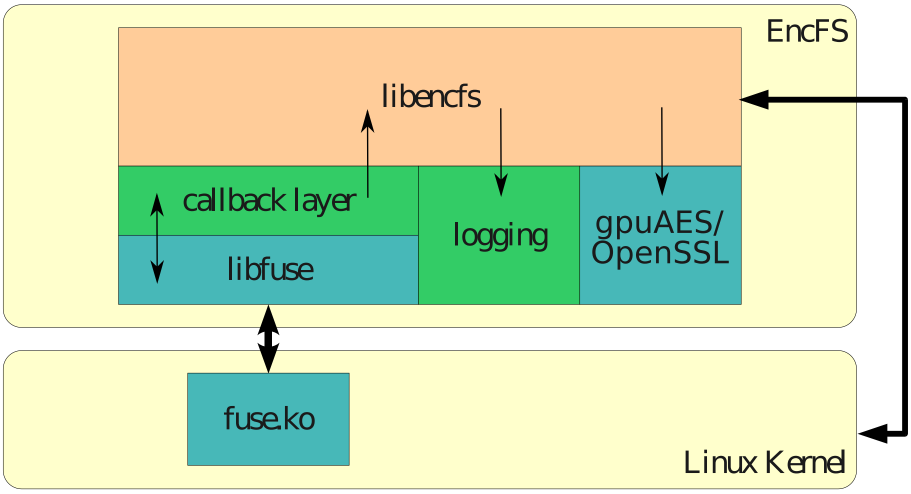


Figure 6.4: EncFS structure image (From EncFS [9])

6.4 Implementation

As the EncFS system is designed so that other encryption back-ends can be easily implemented with other encryption, we began our work with implementing our encryption module from scratch. We could not base our work on anything other then a dummy module which did not do encryption. We therefore implemented a module which did GPU encryption with CUDA instead of using the OpenSSL library to encrypt data. Because we did not always get a fixed size of data to encrypt, we chose to use the host CPU for encryption when the data size was smaller than a EncFS file system block. We have therefore extended the EncFS application with a new module which does encryption on the GPU instead of using the OpenSSL library as is normal. We implemented encryption with two different schemes, based on the the lookup-table and the standard implementation which we both have from chapter 5.

The new module was implemented with host, lookup-table and standard implementation, and it was possible to switch between these with a simple recompile.

We found that the EncFS library has separated the encryption from the main program, as can be seen in the structure figure 6.4. This meant that we in the encryption class had very little influence over what to do with the data, when it was done encrypting. We had no means of buffering it into a larger job, or deferring it, because libencfs expects to get the data encrypted back as soon as possible. This meant that we were not able, as we hoped, to be able to defer the writes to the underlying file system in order to improve the performance. The layered structure of EncFS means that in order to change this behavior, to one better suited for GPU, encfs would need to be rewritten almost from scratch. Everything is constructed in classes built on top of each other, almost like a protocol stack, which makes it hard to do things across classes. Having a more flat design, would make it easier to change the functionality to one better suited for GPU.

6.5 Results

In this section we will describe the results from the different tests we run. The tests consists of benchmarking the throughput and time of our GPU offloaded implementations. Also tested is the host based implementations for comparison.

Different EncFS block sizes has been used for the tests, and in these the implementations have been tested on a single GPU. Since we in chapter 5 had different host based implementation, we compare to these instead of comparing to OpenSSL as the implementations are equal, and thus we can easier see what actual effects we get from porting some part of the encryption to the GPU. The implementations are the one we call standard, and the lookup-table based implementation.

For the throughput performance benchmarks the program *iozone* [63] is used. We benchmark for average throughput in KB/s, as well as testing the CPU time used for both CPU and GPU implementations, with the Unix *time* tool. The standard automatic test in iozone is used, which runs a various set of tests for record sizes of 4k to 16M for files sizes of 64k to 512M. We calculate the average throughput of reading and writing the different record sizes, and graph this average. Our implementations are tested on the GPU listed in table 6.2. The different tests are *Writer*, *Reader*, *Random Reader*, *Random Writer*, *Fwrite* and *Fread*:

- Writer is a test which creates a new file with content.
- Reader is a test which reads an existing file.
- Random Read is a test to read from random locations within a file.
- Random Write is a test to write to random locations within a file.
- Fwrite is a test which writes a new file using the library function fwrite.
- Fread is a test which reads from an existing file using the library function fread.

The tests are run with some different EncFS block sizes, 16384, 32678, 65536 and 81920 bytes. The reason for the relative large block sizes is to avoid being too penalized for the latency to transfer the job to the GPU, and we believe these block sizes can still be useful for various file operations, like operations on larger media files. Transferring very small jobs, is generally faster to run on the CPU, except for very compute intensive tasks. In other words, we are optimizing our file system for large files. The EncFS block size does not correspond with the underlying file systems block size.

6.5.1 Standard implementation

We see in figure 6.5 that for the standard implementation the throughput with a EncFS block size of 16384 bytes is basically the same for the host and gpu based implementation. We believe the reason that we do not get a increase in throughput or is due to the fact that the jobs scheduled for the GPU are to small, so the increased performance in the encryption job we get from the GPU is lost by the latency we experience transferring the data to the GPU.

To investigate this suspicion, we run the tests with larger EncFS block sizes for this implementation in figures 6.6 and 6.7. Here we see that with larger block sizes, both the GPU and HOST performance increases for read operations, while it decreases for

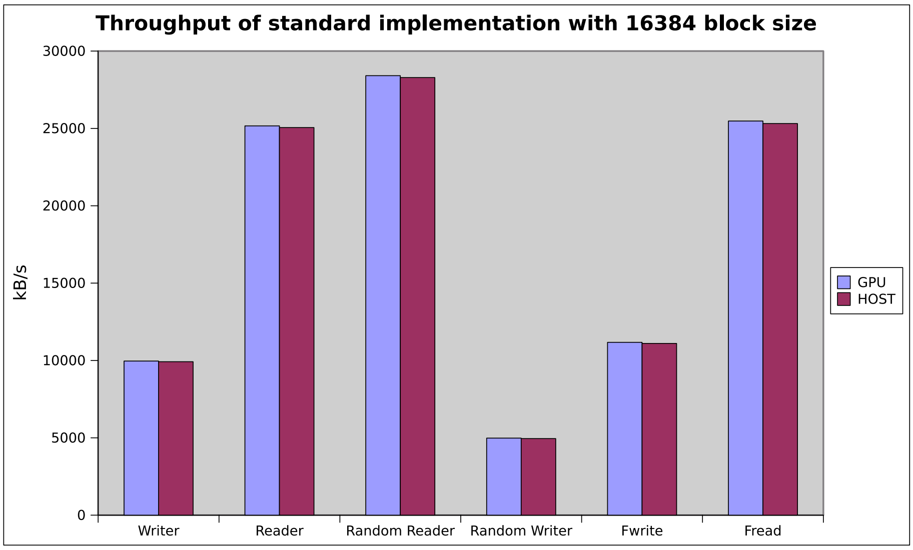


Figure 6.5: EncFS standard implementation throughput (16384)

write operations. The larger block size is beneficial when reading because of caching, especially for the GPU, because EncFS has a caching system which can cache the last block written. The writes are however, penalized for writing such large blocks, because a large block must be encrypted and written, even if the data chunk is small. We see that fwrite performs better than write, which can be because it is a library function which has its own caching, which helps with the large block sizes.

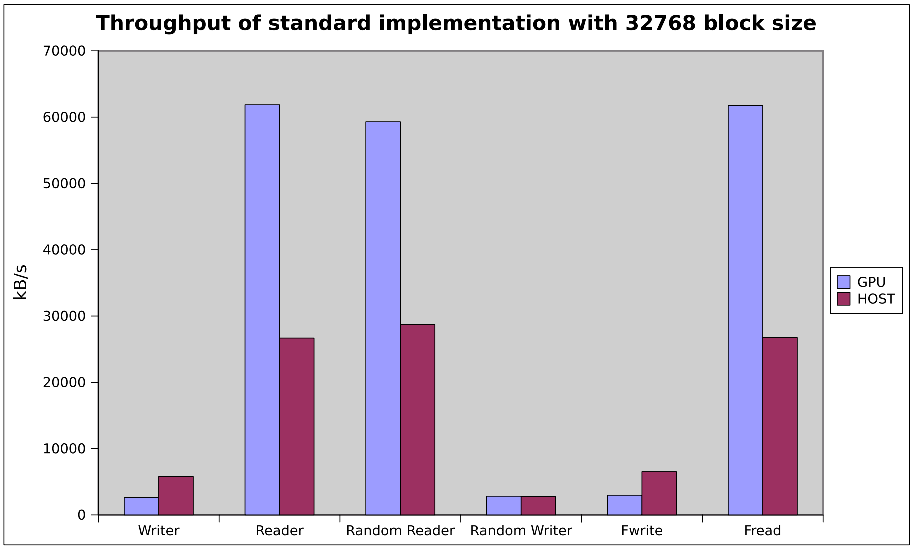


Figure 6.6: EncFS standard implementation throughput (32768)

Running the same experiment with an even larger block size, 81920 is seen in figure 6.8. We see the same trends here, but with writes suffer more from the overhead. We also here believe we see the effects of the EncFS single block cache.

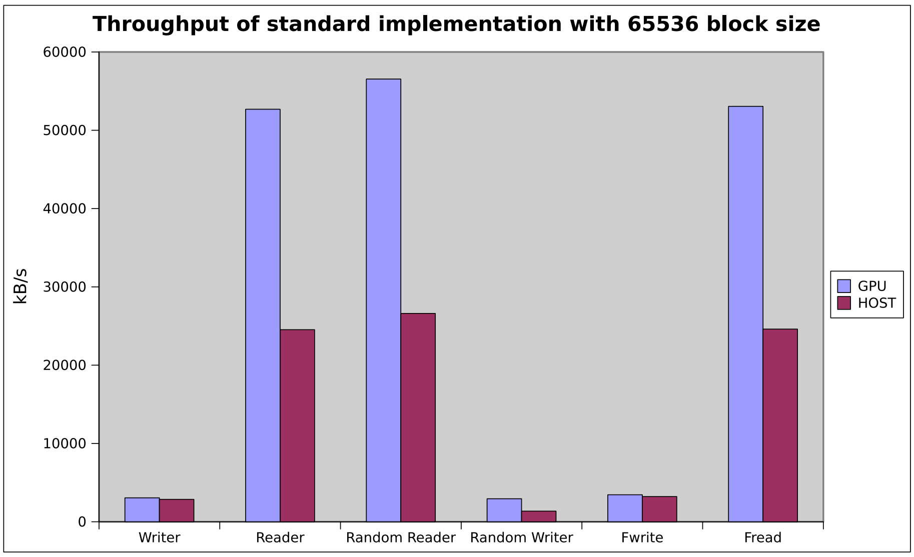


Figure 6.7: EncFS standard implementation throughput (65536)

In figure 6.9, we see the read speeds with different block sizes. We see that it does not increase beyond 32768, and this is therefore an ideal block size to use for further testing.

We have therefore looked into how the standard implementation would behave with both certain file sizes, and certain write and read chunk sizes. We have taken a typically small file size, 32KB which could be a log file, a document or similar, and the performance is seen in figure 6.10. This figure shows that performance is somewhat worse on the GPU for writes, except for random write where it is the same. The performance is better for reads, except random reads, where it is somewhat worse. It seems that for these kind of files, the offloaded file system will be suited.

We have also looked at the performance for larger files. These file are 1024MB, where file chunks are 16KB. For these large chunks and file sizes, we see that the read performance is better, with writes still having decreased performance compared to the host implementation.

Because we are not only interested in the performance, but also the effect offloading has on CPU usage, we have run tests on the CPU time while running an iozone test. These are seen in figure 6.12, and we see that CPU usage in the offloaded file system has been reduced. The individual User and System times are seen in table 6.1, where we see that the User time has decreased, and the system time has increased a lot. The total times generally decrease, however for the block size 32768 byte, we saw a large decrease in CPU load for the host implementation, so the host implementation has a slightly lower load, which might be related to CPU cache effects. The CPU load while

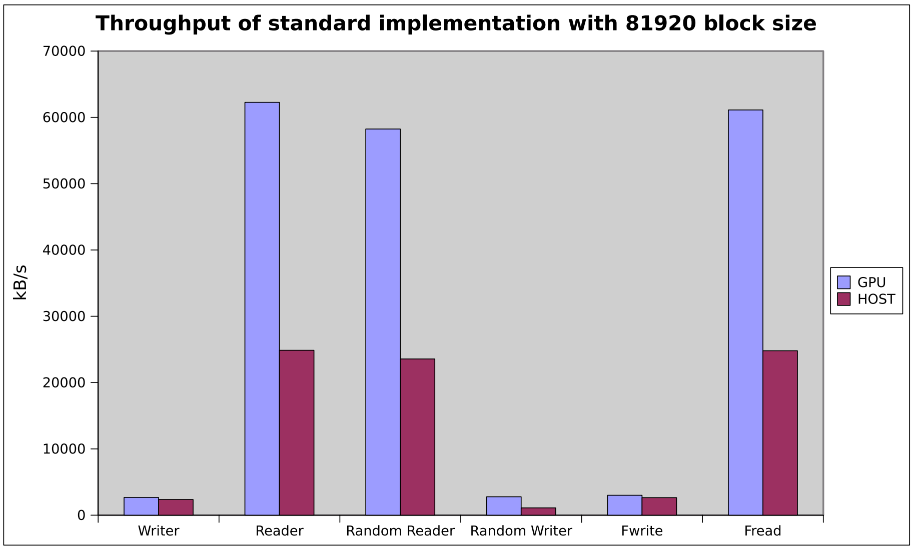


Figure 6.8: EncFS standard implementation throughput (81920)

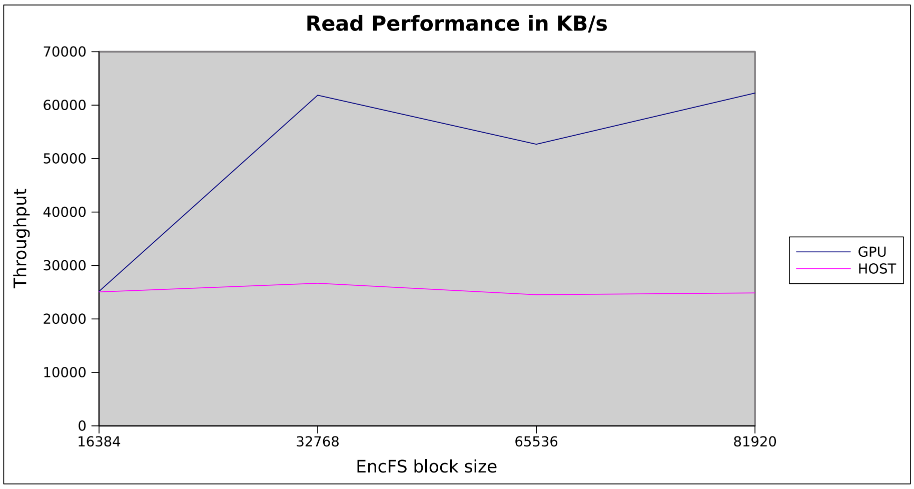


Figure 6.9: EncFS GPU throughput

running the GPU implementation stays constant for the different block sizes. We can therefore conclude that CPU load generally decreases, even though performance does not always increase.

6.5.2 Lookup-table based implementation

We also run the some of same tests on the more efficient lookup table based implementation. With this implementation we clearly see that the latency is too large because of the small job sizes, so it is more efficient to run the jobs on the host CPU, with the lookup table implementation.

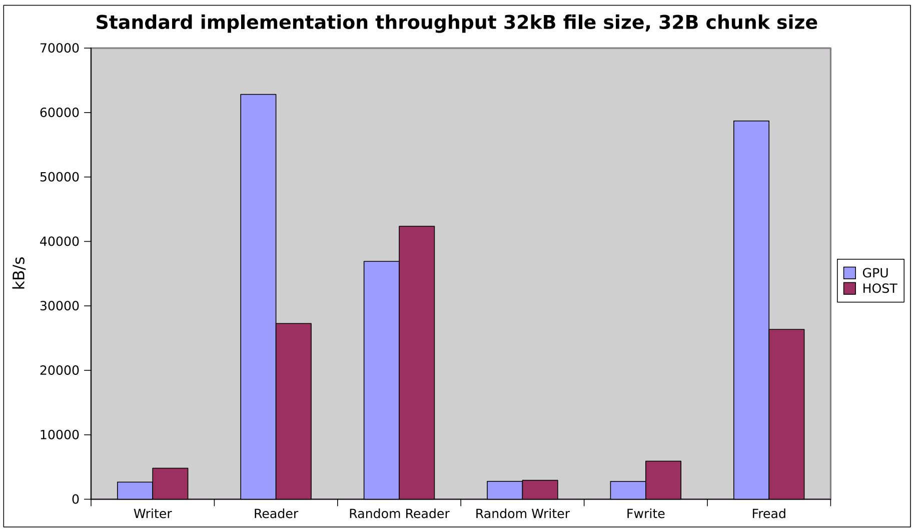


Figure 6.10: EncFS standard implementation throughput for small files with 32768B EncFS block size

Implementation	16384	32678	65536
gpu-user	207	207	191
host-user	707	306	575
gpu-sys	134	132	136
host-sys	10	7	9

Table 6.1: User and System time of EncFS standard implementation

In figure 6.13, we see that the host CPU has a much higher throughput. If we increase the block size to 81920 bytes as shown in figure 6.14 the GPU performance is equal to the CPU, however we do not get an actual throughput increase compared to the same implementation with a smaller EncFS block size. This is because the tests iozone uses operate on smaller chunks in average, although we can adjust this to larger ones, it would not be practical for a file system.

GPU	**GTX 280**
Architecture	GT200
Stream processors	240
Core Clock (MHz)	602
Shader Clock (MHz)	1296
Memory Clock (MHz)	1107
Memory Amount	1024MB
Memory Type	GDDR3
Memory Interface	512-bit
Memory Bandwidth (GB/sec)	141.7
Compute Capability	1.3

Table 6.2: Hardware specifications for the GPU we have tested on [1]

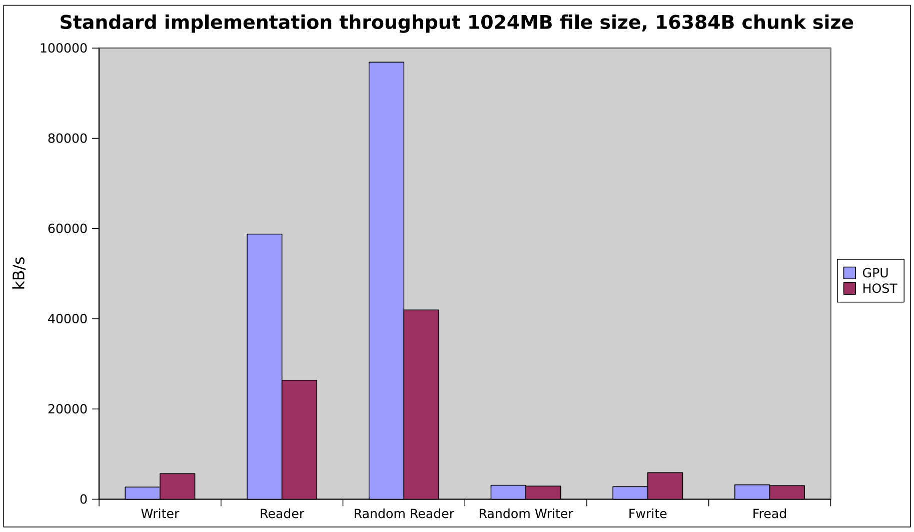


Figure 6.11: EncFS standard implementation throughput for large files with 32768B EncFS block size

6.6 Discussion and lessons learned

We have shown that the EncFS library is not optimal for GPU offloading. We believe that the way it is designed severely limits the possibility for getting a performance increase from offloading. The main reason is that EncFS does not have a block cache, and this is not easy to add because of the way a libfuse file system works. libfuse works on files, not on blocks in a file system, and therefore, it would be hard to create an efficient cache, which we think is necessary for efficient GPU usage in EncFS. With a block cache or a system for deferring writes to the underlying file system, it could have been possible to get an increased performance. Buffering per file could be a possibility, but is not feasible with the current structure of EncFS.

It would possibly be more ideal to implement the encrypted file system in kernel space, but since the CUDA framework is made for user space, we might need to run the encryption in user space, for example by running a daemon in user space, which the kernel could communicate with. However, this might have more overhead both because of the context switches between kernel and user space, but there is also more overhead because requests must go through the FUSE kernel module, as well as libfuse. These issues could mean that performance might not be better than running the encryption using only the CPU in kernel space.

It is possible that file systems are not well suited for offloading, because they operate generally on blocks that are too small to be practical for GPU usage that makes no sense to give as a job to the GPU. We have shown that if the job size increases, the

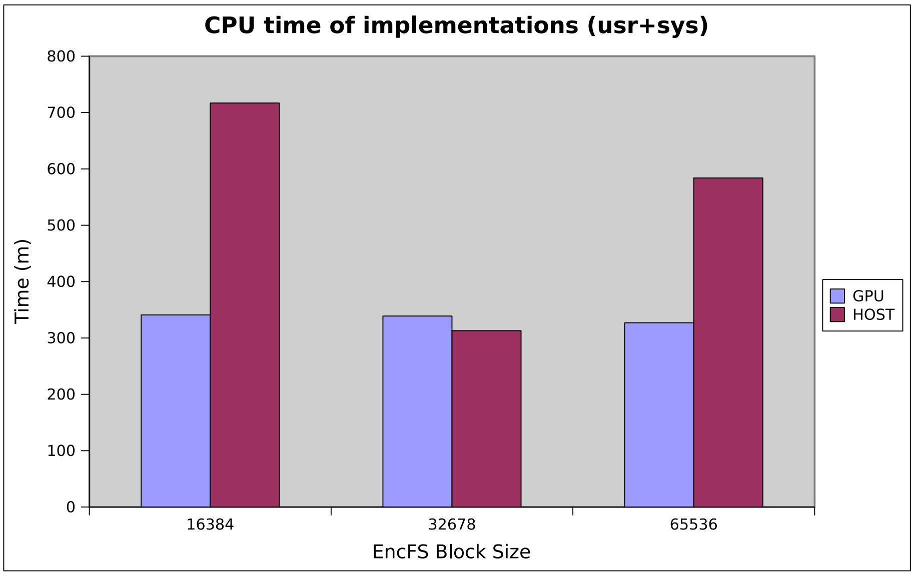


Figure 6.12: EncFS standard implementation CPU time

performance can also increase if the implementation is efficient.

The binary protocol using the FUSE kernel module is inode based, and might have been possible to use in a file system like EncFS to give better possibilities of implementing a defer mechanism or an efficient cache. The FUSE kernel module could allow us to implement a block based ordinary file systems, and have a buffer from which we would do deferred writes, and also work as a cache for read and write request, until they are actually written to disk.

One thing we could have done more tests on was when it was beneficial to do a job on CPU and when it was beneficial to do it on the GPU. We did as mentioned do jobs smaller then the EncFS block size on the CPU, and it would be interesting to see what size the ideal job on the GPU would have been.

We have however found that CPU load can decrease by offloading parts of an application even if performance does not increase, and this is clearly beneficial for many applications, because it allows other applications running on the same machine that runs the offloaded application to get access to more CPU time.

We learned that the structure of EncFS and FUSE made it hard to implement an efficient deferred writing method. At most we could have buffered parts of a file, but since a real filesystem will often work on multiple files we do not believe this had a potential for a real performance benefit.

Since we knew from chapter 5 that the speedup from running the AES algorithm on the GPU was not orders of magnitude faster, we could reason that the performance

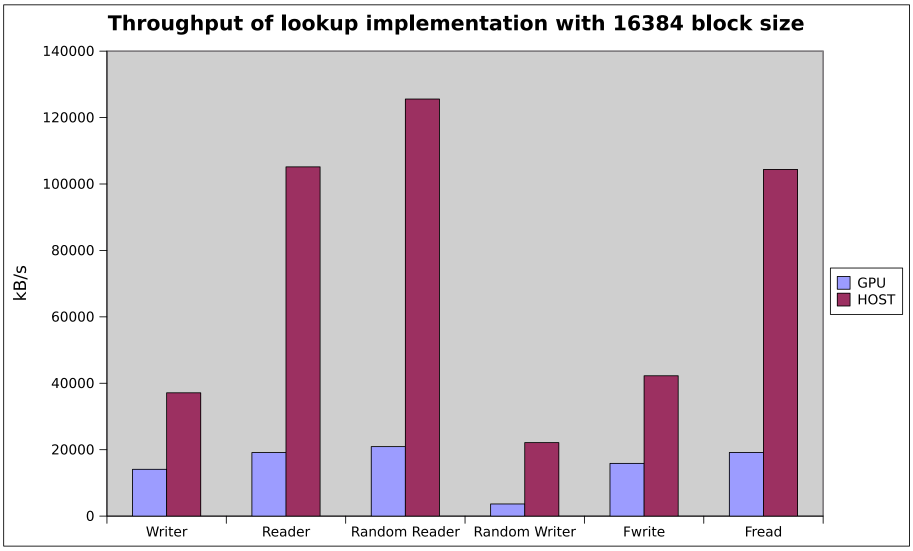


Figure 6.13: EncFS lookup implementation throughput (16384)

potential of offloading the encryption part of EncFS to the GPU was possibly not great. Other tasks which are more compute intensive and are orders of magnitude faster to run on GPU compared to CPU are probably more beneficial to offload, because the data copying will not be that much of a hinder, if the GPU kernel runs much faster than the CPU counterpart. It was however, interesting too see the reductions we could achieve in CPU load, which we regard as a good result.

6.7 Future work

For future work it would be interesting to measure the latency of file operations, not only the throughput. We have not given this any consideration, but for many purposes it would be important with low latency file operations, instead of, or in addition to high throughput file operations

For future work it would be ideal to write the encrypted file system from scratch, instead of basing it on EncFS. This could be done with either the FUSE kernel module, or from the kernel communicating with a user space daemon.

6.8 Summary

Our tests show that it is possible to modify an existing user space file system to use a GPU, but that GPUs might not be suited for offloading file systems unless they have

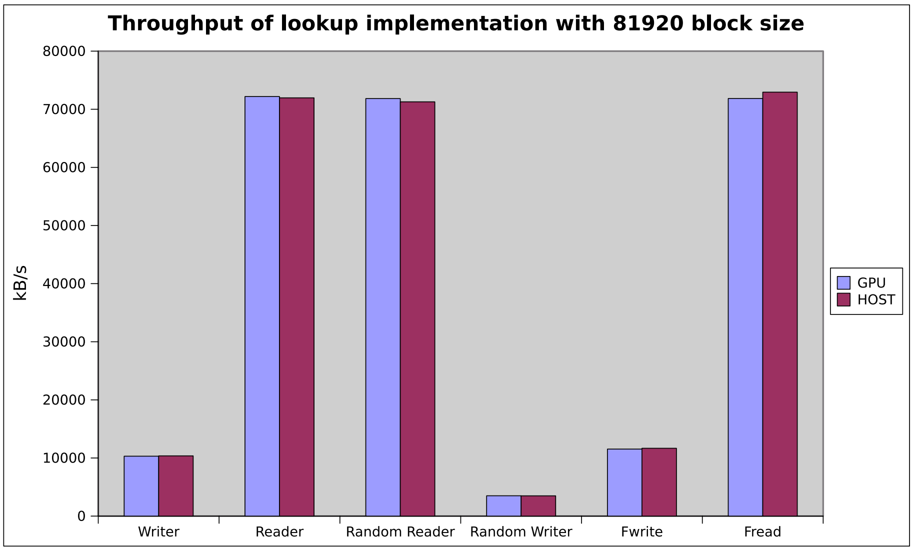


Figure 6.14: EncFS lookup implementation throughput (81920)

some sort of buffer or deferred writes. This is also seen in the accelerated RAID file system we discussed in chapter 2 where Curry et al. found that you need deferred writes to get good performance [16]. We have observed that it must be much faster to run the code on GPU in order for a offloading to give a performance increase, however, there are benefits in that the CPU loads are reduced. Obviously, as GPUs and CPUs evolve differently, running these same tests in few years might yield different results.

We have seen that for implementations that are significantly faster running on the GPU then the CPU, we can benefit from offloading the application to GPU, and get both better or the same throughput and lower CPU load while running the application, which we believe is important for many other applications.

An open question is whether there are more efficient ways of using the GPU for offloading than how we have used it so far. In the next chapter, we therefore investigate if a technology available in CUDA, called CUDA streams has potential for further performance gains. If this technology is promising, it could be used in a file system that wrote larger chunks, with the deferred writes and buffering we have discussed.

Chapter 7

Optimizing AES with CUDA streams

7.1 Introduction

In this chapter, we return to our AES implementation from chapter 5, to investigate the potential for optimizing not only the speed of the kernel, but the total execution time using a technology available in CUDA called *CUDA Streams*. We will first discuss streams, and how they are programmed, followed by a discussion on our AES implementation using streams. We will also discuss what kind of applications gets the most benefit from using streams.

7.2 CUDA Streams

Streams are a technology, available in CUDA that allows for memory copying to overlap with kernel execution, which gives the potential for better performance.

In this section, we introduce a small example on how CUDA streams are programmed, with code from NVIDIA [34]. Creating two streams are done by calling the *cudaStreamCreate* function seen in listing 7.1. Here we create two streams, which might allow for some overlap in memory copying and execution. Note that this is not the same as double buffering, it could be any number of streams, and we do not have full control over their execution.

In listing 7.2, we see the actual kernel execution, as well as the the calls to *cudaMemcpyAsync* which does asynchronous memory copy operations, meaning we start off all memory copy operations in lines two through three, and associate each of them with a

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
```

Listing 7.1: Creating two streams with CUDA

```
1
2  for (int i = 0; i < 2; ++i)
3      cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
4                      size, cudaMemcpyHostToDevice, stream[i]);
5  for (int i = 0; i < 2; ++i)
6      myKernel<<<100, 512, 0, stream[i]>>>
7             (outputDevPtr + i * size, inputDevPtr + i * size, size);
8  for (int i = 0; i < 2; ++i)
9      cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
10                     size, cudaMemcpyDeviceToHost, stream[i]);
11 cudaThreadSynchronize();
```

Listing 7.2: Copying memory and launching a kernel

stream. The purpose of this is to launch a kernel when the memory operation is complete. The launch of *myKernel* in line five, which is associated with a stream, and this kernel will start when the memory operation is complete. The second launch of *cudaMemcpyAsync* is done after the kernel is launched in lines eight to nine, which copies the data back when the kernel for a given stream is complete. The purpose of using streams here is that a small memory copy can be allowed to finish, so that a kernel which operates just on this memory can be launched. When this kernel is complete, the memory is copied back to the host.

The point for using CUDA streams is to get overlap between memory copying and the kernel launches. We are working from a hypothesis that to achieve the most benefit from using streams, having a kernel that takes the same amount of time as the total memory copy time, that is to and from the device is most optimal, and can cause a 50% reduction in execution time. In figure 7.1, we see the total application time with single launch, multi launch and using streams in an example applications where the kernel takes twice the time as the memory copy. We divide the task into two launches using streams and multi launch. Normally, a higher number of launches will be used. From the figure, it can be seen that we cannot start executing before the first memory

```
float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
```

Listing 7.3: Allocating pinned memory on host

copy is complete. With many launches, waiting for the first memory copy to finish will be insignificant for the total execution time. We can assume that for larger memory operations needed for each stream executed, the more computationally intensive the kernel needs to be. If either the kernel or memory operation takes a very short time to execute, the other operation will be the most significant, and therefore, getting an overlap between these two, we will not get a significant increase in speed. Because we do not have information about how streams are scheduled, we will test this hypothesis in this chapter.

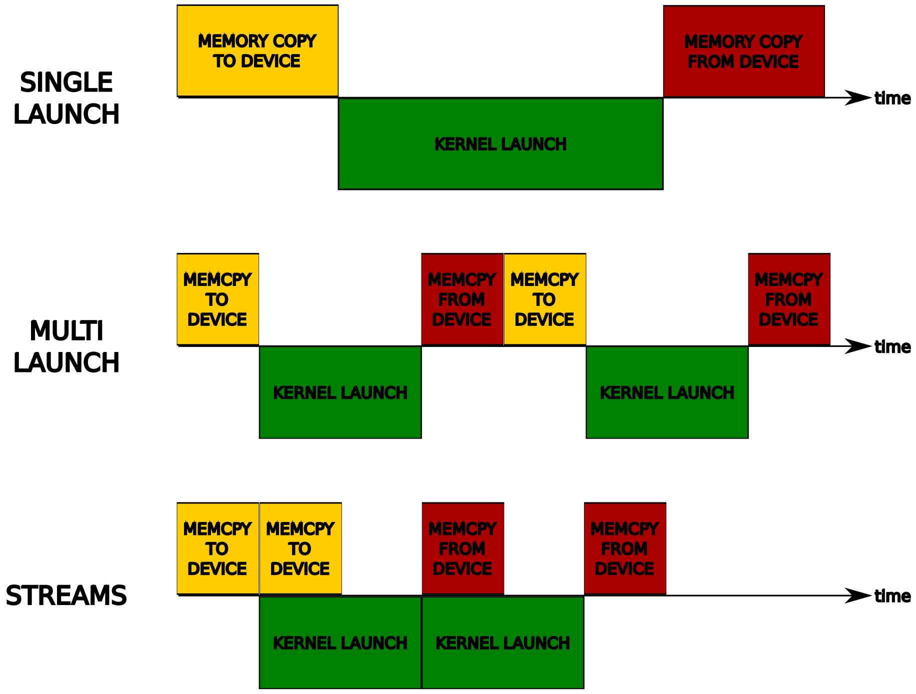


Figure 7.1: Overlapping memory copy and computation, under optimal condition

One disadvantage of using streams is that we need to use pinned (non swappable) memory on the host. However, this can be faster than using swappable memory, but might not be beneficial on all systems, since memory is often limited. Allocating pinned host memory is done with the *cudaMallocHost* function, shown in listing 7.3.

7.3 Implementation

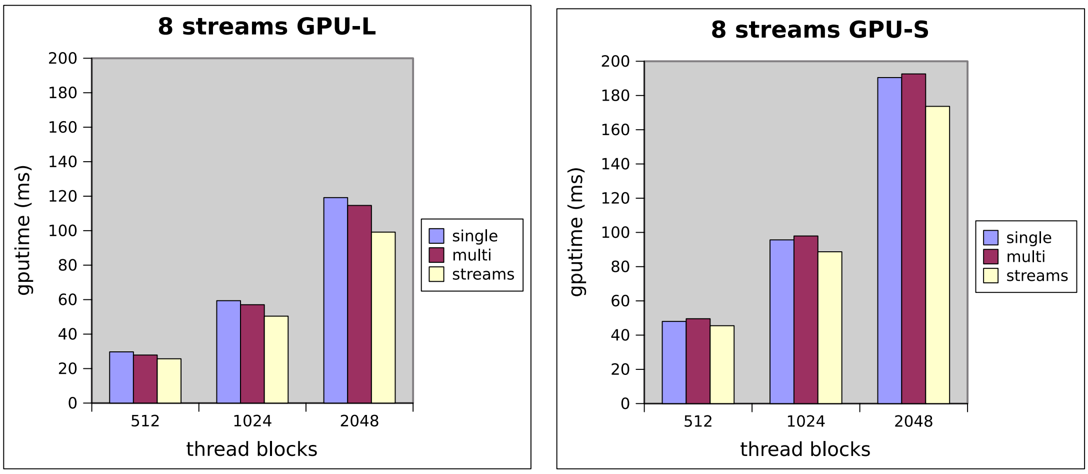
We based the work in this chapter on the AES encryption implementations from chapter 5, and changed them to use streams, to see what changes we would get from this. The implementations have three different modes, which we run tests with. The three modes are shown in figure 7.1, one where we first do all memory copying, then launch a kernel, then copy back, called *one-launch*. Another mode was the *multi-launch* which divides the job into smaller pieces, then do a smaller memory copy to device, a smaller kernel launch and a smaller memory copy to host, until the job is complete. The last mode was using *streams* where we designed the program as to be able to adjust the number of streams arbitrarily, too see what number of streams was most beneficial.

We also implemented a new kernel, to be able to test if there was connection between the ratio between memory copying and kernel execution time. In this kernel, it was possible to adjust the kernel execution time with a parameter, so we could test out this hypothesis.

7.4 Results

We have run test on the lookup table and standard implementation, and the results are shown in figure 7.2. We tested with from 2 to 8 streams, we could not test with a higher number of streams, because it would demand too much pinned memory. From the figure, we can see the performance benefit from using streams. The details of the speed increase can be seen in table 7.1. We see that the number of streams affect the increase in speed, topping out at a 15% speedup for the lookup table implementation. The standard implementation does not result in an equal speedup, which shows that the ratio of memory copying to kernel execution time influences the possible speedup. We do not, however, get a speedup near what we believe is the theoretical limit 50%, so we have chosen to further test with a arbitrary workload that we can adjust, until we get the optimal performance.

We have tested the kernel which takes a a parameter that controls its load, and showed the different speedups compared to the single launch in figure 7.3. Here, we can see that there is a optimal load for the kernel, depending on the memory copying needed. Job size is on the x-axis, and this is iterations in doing a mathematical calculation consisting of some multiplications, additions, subtractions and divisions in global memory, but it could be any sort of workload, hence we call it a *generic workload kernel*. The



(a) Lookup Table implementation

(b) Standard implementation

Figure 7.2: Streams GPU

number of thread blocks vary, and with that the memory copied varies. We see that streams might be slower than running a single launch for kernels which have a low execution time, compared to the time for memory transfers. If the job is too small, or too large compared to the memory copying needed, the speedup of using streams is therefore not as good.

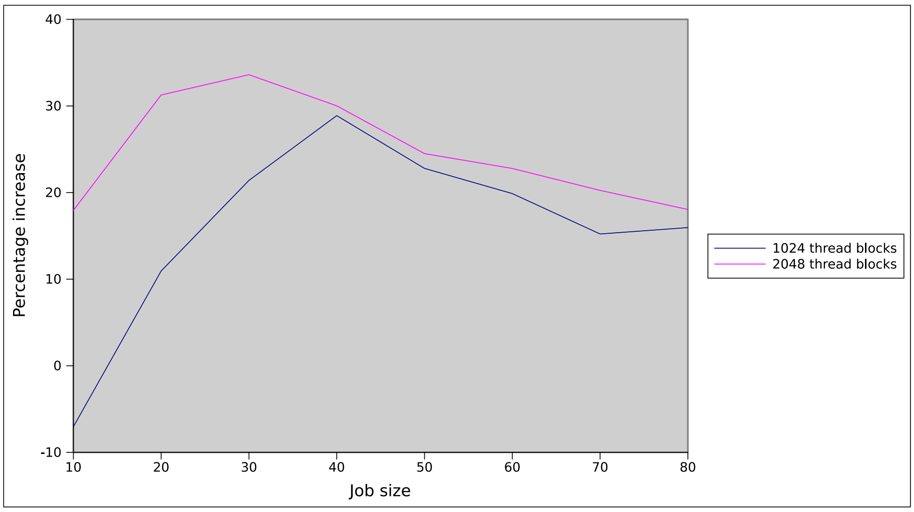


Figure 7.3: Job completion ratio on stream implementation

7.5 Discussion and lessons learned

We have seen that both the standard and lookup implementations are far from the optimal speedup we could achieve with the generic workload kernel. The generic

number of streams	GPU-Lookup % increase	GPU-Standard % increase
2	6.6	2.1
4	13.1	5.3
6	13.8	6.0
7	14.6	6.1
8	15.1	7.1

Table 7.1: The average increase in speedup of the kernels running 512, 1024 and 2056 thread blocks compared with different number of streams compared to a single launch.

workload kernel had over 30% speedup, the lookup table implementation was at 15% speedup, and the standard implementation was at 7%. This shows that the kernels do too much work for the memory copied.

We did not, however, achieve the theoretical speedup that we had a hypothesis about. Getting a kernel which takes exactly the same time as the memory copying is not trivial to implement, and we believe that there will always be some overhead in the multiple kernel launches needed to use streams, so achieving the theoretical maximum speedup 50% is not realistic.

Applications might have another memory copying pattern, some will copy a big dataset to the GPU for calculation, but read a few bytes for the result. Others might use little or no data to generate a big dataset. For these applications the potential speedup using streams are still the same, but since there is only one memory copy, this copy operation would need to take as much time as the kernel execution time.

As we have seen, streams can often increase performance, but as mentioned earlier, it demands host pinned memory. This can prove problematic in some cases and means we must be more careful with how much memory to allocate. In testing, we experienced that our test computer was not particularly responsive when running with a high amount of pinned memory. Running a single launch with non-pinned memory the computer did not have this problem.

Unfortunately streams can not be integrated in EncFS at the time, because we did not find a solution for buffering data until we had larger requests. We do however believe that the streams technology is important, because we saw that it could increase performance of our AES implementation. Combined with buffering, throughput could be increased in a file system, in addition to the reduction in CPU load we have already seen in chapter 6.

7.6 Future work

For future work it would be interesting to test what kind of speedup we could get for kernels that does not have the same amount of memory copied to device, as the amount copied from device.

It would be interesting to evaluate the streams performance using a even more optimized kernel, like the one Manavski have used in their AES GPU implementation [64], which potentially could be more optimal for stream usage. This kernel was more mathematically complex, and used four threads per cipher block, and we have therefore yet to test it.

Designing and writing a encrypted file system from scratch, we believe using CUDA streams could be a interesting technology to use. For future work it would therefore be interesting to investigate the performance of streams in a file system where we had larger data sets to encrypt.

7.7 Summary

In this chapter, we have learned that using CUDA streams can give benefits in performance of CUDA applications. It is not always beneficial to use streams and there needs to be a certain ratio between the time used for kernel execution and memory copying. We have seen that our AES implementation was an application that could benefit from using streams.

Our tests has also shown that it is hard to know in advance if using streams are beneficial. As a result, application developers should try using streams to see if their application benefits from using it. We have seen that streams will rarely have a negative impact on performance, and if they do, it is not significant. However, streams should only be used if it is acceptable to used pinned memory, which might cause problems if the data required by applications uses a high percentage of the available system memory.

We have seen that AES performance increases by using streams, and discussed how this would be positive in a file system. Larger data requests then what is used in EncFS is although needed, and streams are therefore not directly applicable in EncFS.

Chapter 8

Discussion

8.1 Introduction

In this chapter, we will discuss some of our experiences with programming GPUs with the CUDA framework. We will also discuss the future for CUDA, and our thoughts about other coming technologies. We also discuss the different aspects of file systems with GPU assistance, and how AES encryption performs on the GPU. Finally, we discuss our experiences with CUDA streams, and how they could be used in a file system.

8.2 CUDA for development

Parallelization is one of the first challenges a developer using CUDA will face. As we have seen in this thesis, porting an application consist of several steps. The first step is to find out if, and how, the algorithm can run with thousands of threads, which is needed to achieve good performance on a GPU.

Identifying which part of the application is computationally demanding and how to parallelize them, is a challenge. For some tasks the OpenMP library, which can be used on ordinary CPUs, might be a first step to parallelize an application, because it allows some parts of an application to be isolated, and parallelized, without thinking in great details of threads and thread communication. It allows the programmer to easily parallelize loops, by automatically creating threads for each iteration of the loop, which is possible if there are no dependencies between each iteration of the loop. From using OpenMP, the developer can acquire knowledge about which parts of the application that would best be suited to run on GPU.

In this thesis, we evaluated two different AES implementations; one which was not optimized at all, called the *Standard implementation*. This implementation was based directly on the AES standard, and no consideration to optimization had been taken. It worked directly on the state matrix, like the standard describes. Implementing in this way, means that a lot of calculation is done, and so the implementation was compute bound. The other implementation was optimized for ordinary CPUs. This implementation used precalculated lookup tables and is called the *Lookup implementation*. We saw in chapter 5 that the lookup implementation was memory bound. We found that the standard implementation had the highest performance increase, but that it could not outperform the CPU optimized AES version running on the GPU. Finding out how to use the memory correctly on the GPU was one of the most important challenges we faced. We have described our methods for how to access memory for others developing and running applications on the GPU. We have looked into the four different types of memory available on the GPUs, which all have different properties, and how memory is used must be considered for each application. This is in contrast to programming ordinary CPU applications, where there is one type of available memory.

Finding out what parts of an application can be offloaded on the GPU requires the developer to do two things. The first step is to isolate the parts of the application that are computationally expensive. The second step is to investigate if this part can be parallelized into smaller jobs not dependent on each other. If it can be parallelized, the developer can start porting this part of the application to CUDA, to evaluate if there are offloading benefits.

8.3 CUDA Tools

We have some various experiences using the CUDA toolkit, and found that it works quite well, but there are some challenges to using it. Integrating it with *autotools* was challenging, and it is a field where more work should be done. We had to solve these problems with EncFS, where we had to write scripts to integrate the *nvcc* compiler with autotools.

Finding out about the memory accesses, which showed to be crucial for performance, was challenging with the tools available. The CUDA profiler could tell us that we had inefficient memory accessed, but not where in the code they where, so finding out where they are can be a challenge, requiring experimentation to find out more optimal access patterns. The CUDA profiler does, however give useful information, like the occupancy, which previously was calculated in a spreadsheet called the *occupancy calculator*. The occupancy calculator required some parameters like number of threads per block and shared memory in use by the kernel. Some of the data needed for the parameters could be acquired by running the CUDA compiler nvcc in a special mode.

8.4 Debugging

Throughout the work in this thesis, we found ourself debugging often, while learning the CUDA framework, but the tools for debugging were not very good. For most of the time writing this thesis, the CUDA GDB debugger only supported 32-bit Linux. Because GDB is not designed for massively parallel GPU applications, it is challenging to use. It is also quite hard to detect if a kernel crashes, the runtime does not always provide an error code, so we had to see that it crashed from unexpected results. Finding out where it crashed was often challenging. However debugging is always an extra challenge dealing with many thousand threads, because of the challenges that come with parallel programming, like synchronization, making sure threads work on the correct data. There are also challenges dealing with architectures which are not as well tested as ordinary CPUs.

8.5 Encryption on the GPU

We have seen that the GPU can perform AES encryption, and that the implementations on the GPU can outperform the CPU. In chapter 5 we learned about the challenges that needed to be solved in order to efficiently port an application to the GPU. We discussed the various strategies for using memory, and saw that memory must be used correctly to get good performance. This work gave us valuable insight in how a CUDA application should be optimized, and what work we needed to do in the GPU kernel, before we could start the offloading of the EncFS file system.

Initially when the implementations was ported to the GPU, performance was poor. When we learned about the various memory types, we understood that it is often important to use a memory type that is cached. Using global memory, is perhaps the intuitive choice, because it is the memory that is allocated with the *cudaMalloc* function, as well as being the easiest to use. With the global memory, memory accesses are not cached and performance suffers, if there are a high number of memory requests. Also, using shared memory when possible proved to yield good performance. We did not use the shared memory to communicate between threads, but divided the shared memory to one cipher block per thread. This is not the way NVIDIA recommends using shared memory, but it worked good for our purposes. The shared memory should although be automatically be used for variables, if there are not enough registers, so it should not have been necessary to do it this way.

Implementing both a standard and lookup implementation we learned a great deal about the challenges a CUDA developer is faced. Debugging was as mentioned a challenge, and when the application finally works, with the most intuitive implementations, performance was much worse the on CPU. Much work was needed to get the desired performance.

8.6 Offloading a file system

In this thesis, we have seen that it is possible to offload parts of a FUSE file system to the GPU. We saw that the structure of EncFS to some extent limited the possibilities to implement the necessary mechanism to get a performance increase, however we saw a reduction in the CPU load. We think that FUSE might be possible to used for this purpose, but the EncFS project does not focus on performance; they focus on security. Our attempt to increase the performance of EncFS shows this.

The lower level FUSE kernel module *fuse.ko* described in chapter 6 seems to be better suited for offloading the file system, and we think this would be a better starting point for a encrypted user space file system. Had CUDA been supported to being able to access from the kernel, a ordinary kernel file system would also be interesting.

Another approach is a hybrid solution, in where the encrypted file system could reside in kernel space, and communicate with a user space daemon which could do encryption. Although there could be challenges implementing this, it could yield interesting results, as more of the code would reside in kernel space, and we could possibly have based our work on an existing file system. It would, however, be better to be able to use CUDA from the kernel, which we hope will be a reality in the future. Since the GPU is already accessed in kernel space, this should be possible to implement. Integrating the GPU with the kernel could also yield some benefits in other parts of the kernel, where offloading to the GPU could be beneficial. However, integrating CUDA into the Linux kernel is not only a technical challenge, but also a licensing and political challenge.

Latency was an issue we did not address, which could be interesting to run tests on. We did not transfer all jobs to the GPU, small jobs was done on the CPU, and this is an area where it would be interesting to run more tests. Optimizing the threshold for when we encrypt files on GPU could give more optimal latency and higher throughput. Latency would probably not be an issue with a deferred write solution. This solution could have stored write requests in a buffer, and serve read requests from this buffer, thus working as a cache. The buffer would be written after a certain time, or when it was full. This would allow a large job to go to the GPU, which would be more beneficial for performance.

8.7 Streams

We have seen that streams is a promising technology, which enables application to overlap memory operations with computation. A challenge with streams is knowing when to use them. The AES implementations had benefit from using them, and for many applications they will be beneficial. To be most beneficial, applications must have as much compute time as memory copy time. If there is a big difference between these times, streams will be less efficient. Theoretically, if there is full overlap, the execution time can be halved. We experienced reductions up to 30% in our tests, showing that there is some overhead in using streams. The main disadvantage with using streams are that they require the used of pinned memory on the host. Pinned memory can yield potentially better performance, but it is not beneficial for other applications running on the same system, as memory is limited.

Using streams would be ideal for a file system where we could have buffered larger amount of data than a file system block. Having a large buffer, and doing a deferred write, means that streams could be beneficial for the file system. We could not, however, implement this kind of buffering in EncFS, and could therefore not benefit from using streams. Even so, streams is a good way to increase performance, and would thus be important when creating a offloaded file system.

Chapter 9

Conclusion

9.1 Summary and contributions

In this thesis, we have implemented AES encryption on the GPU. We have investigated two different implementation, one memory bound and one CPU bound, and investigated the possibilities of optimizing them. Also, we have seen that the performance of GPU programs can increase in orders of magnitude by using the correct optimization techniques in GPU programming.

This knowledge about AES encryption has been applied to a encrypted user space file system, where we changed the data encryption part to run on GPU, instead of the CPU-based OpenSSL implementation it previously used. We have seen that performance can increase in some cases, and that the CPU loads can be reduced when we offload parts of the application to the GPU. This is to our knowledge both the first FUSE program, and the first file system to have offloaded processing to the GPU.

The thesis then investigates a technology called CUDA streams, which allows for doing overlap between memory copy and kernel execution, which can increase performance. We have looked into how much this helps with our AES implementations, and also discussed and investigated in what situations streams are most beneficial.

In this thesis we have shown that AES encryption is beneficial to run on GPU, using the *counter* mode of operation. It is shown that different implementations can have various effects from offloading to the GPU. We have shown how applications can be ported to run efficiently on the GPU. The thesis explains how memory accesses are an important part of programming with the CUDA framework, and how one should go about to access memory on the GPU.

It is shown that applications which are not previously optimized for CPU usage has a greater potential for GPU optimizing, than applications optimized for CPU with using the correct memory access patterns. We show that a user space file system can benefit from offloading a CPU intensive part to the GPU, in terms of performance and CPU load.

Our experiments have shown that using CUDA streams for applications can give reductions in execution time, with a theoretical maximum of half the execution time. Our experiments shown up to a 30% execution time reduction best case, which means we have shown that there is some overhead using CUDA streams.

9.2 Critical Assessments

The user space file system EncFS was shown not to be ideal for GPU offloading, because it was not trivial to implement the desired features. Although the code was well documented, its structure made it difficult for us to implement a buffer solution. Alternatively, we could have chosen to develop the file system from scratch using the low level fuse library, and focus on one AES implementation instead of two. With an implementation such as this, it would might have been possible to implement the deferred writes we have discussed in chapter 6, which we have reasoned for why would have much better performance.

A file system might not be the optimal application for GPU offloading. Unless we have a deferred write solution, data will often not be coming continuously in a stream, unlike other applications, for example video decoding or encoding, which could have been a more optimal application to show the effects of the work we did in 5.

We chose to work with the CUDA framework, although we knew it was closed source. This meant that we did not have full insight into how the framework was constructed. We have seen many several times that it is hard to know exactly why a optimization attempt works, or do not work, although we have been able to make educated guesses.

9.3 Future Work

For future work it would be interesting to look into the new features coming in later versions of CUDA, and test them against our implementations.

It would also be interesting to look into OpenCL and Larrabee, and evaluate how running the AES encryption, and perhaps creating a file system would behave with these frameworks. This would enable us to do a comparison of different frameworks.

Writing an alternative to the EncFS file system from scratch would be interesting, to implement all the desired features, and to optimize for GPU, implementing the deferred writing which we have discussed.

Appendix

Attached is a CD-ROM containing all source code of the AES, EncFS and CUDA streams implementations. The CD-ROM also contains all test results from this thesis.

The contents of the CD-ROM can also be found online at the following address:

`http://www.ping.uio.no/~magne/master/`

Bibliography

[1] NVIDIA. GeForce GTX 280. *http://www.nvidia.com/object/geforce_gtx_280.html*, 2008. [Online; accessed 20-November-2008].

[2] Wikipedia. Standard RAID levels — Wikipedia, The Free Encyclopedia. *http://en.wikipedia.org/w/index.php?title=Standard_RAID_levels&oldid=283482170*, 2009. [Online; accessed 13-April-2009].

[3] NVIDIA. Programming guide 2.0 beta2. *http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf*, 2008. [Online; accessed 14-October-2008].

[4] Anandtech. The Radeon HD 4850 & 4870: AMD Wins at 199*and*299. *http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=3341*, 2008.

[5] NVIDIA. CUDA Architecture Overview. *http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf*, 2009. [Online; accessed 29-March-2009].

[6] Wikipedia. Advanced Encryption Standard — Wikipedia, The Free Encyclopedia. *http://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=243022814*, 2008. [Online; accessed 6-October-2008].

[7] Wikipedia. Block cipher modes of operation — Wikipedia, The Free Encyclopedia. *http://en.wikipedia.org/w/index.php?title=Block_cipher_modes_of_operation&oldid=227002102*, 2008. [Online; accessed 9-October-2008].

[8] Wikipedia. Filesystem in Userspace — Wikipedia, The Free Encyclopedia. *http://en.wikipedia.org/w/index.php?title=Filesystem_in_Userspace&oldid=273845271*, 2009. [Online; accessed 2-March-2009].

[9] EncFS. EncFS. *http://www.arg0.net/encfs*, 2009. [Online; accessed 2-March-2009].

[10] Stanford. Performance measures for Folding@Home. *http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats*, 2008. [Online; accessed 14-October-2008].

[11] NVIDIA. Nvidia geforce gtx 200 architectural overview. *http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf*, 2008.

[12] NVIDIA. Cuda compiler driver nvcc 2.0. *http://www.nvidia.com/object/io_1213955090354.html*, 2008.

[13] NVIDIA. GeForce 8800GT. *http://www.nvidia.com/object/geforce_8800gt.html*, 2008. [Online; accessed 20-November-2008].

[14] NVIDIA. GeForce 8800. *http://www.nvidia.com/page/geforce_8800.html*, 2008. [Online; accessed 20-November-2008].

[15] NVIDIA. GeForce 8 Series. *http://www.nvidia.com/page/geforce8.html*, 2008. [Online; accessed 20-November-2008].

[16] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Accelerating reed-solomon coding in raid systems with gpus. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–6, April 2008.

[17] NVIDIA. NVIDIA PhysX. *http://www.nvidia.com/object/nvidia_physx.html*, 2009. [Online; accessed 16-April-2009].

[18] Berkeley. The science of SETI@home. *http://setiathome.berkeley.edu/sah_about.php*, 2009. [Online; accessed 29-March-2009].

[19] Stanford University. Folding@Home. *http://folding.stanford.edu/*, 2008. [Online; accessed 10-October-2008].

[20] Stanford University. Folding@Home FAQ-ATI2. *http://folding.stanford.edu/English/FAQ-ATI2#ntoc5*, 2008. [Online; accessed 14-October-2008].

[21] Stanford University. Folding@Home Clients. *http://folding.stanford.edu/English/Download*, 2008. [Online; accessed 10-October-2008].

[22] Wesley De Neve, Dieter Van Rijsselbergen, Charles Hollemeersch, Jan De Cock, Stijn Notebaert, and Rik Van de Walle. Gpu-assisted decoding of video samples

represented in the ycocg-r color space. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 447–450, New York, NY, USA, 2005. ACM.

[23] Inc. Elemental Technologies. Badaboom media converter. `http://www.badaboomit.com/`, 2008.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[25] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[26] NVIDIA. New NVIDIA Products Transform the PC Into the Definitive Gaming Platform. `http://www.nvidia.com/object/IO_37234.html`, 2006. [Online; accessed 10-October-2008].

[27] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[28] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[29] ATI. ATI Radeon™ HD 4800 Series - GPU Specifications. `http://ati.amd.com/products/Radeonhd4800/specs.html`, 2008. [Online; accessed 10-October-2008].

[30] ATI. ATI Radeon™ HD 3800 Series - GPU Specifications. `http://ati.amd.com/products/Radeonhd3800/specs.html`, 2008. [Online; accessed 10-October-2008].

[31] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[32] Intel. Intel MMX Pentium. `http://www.bdti.com/procsum/mmx_pent.htm`, 1997.

[33] Anandtech. Intel's Larrabee Architecture Disclosure: A Calculated First Move. http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=3367, 2008.

[34] NVIDIA. CUDA ZONE – The resource for CUDA developers. http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf, 2008. [Online; accessed 14-October-2008].

[35] Intel. IA Software Developer's Manual, Vol 1. ftp://download.intel.com/design/PentiumII/manuals/24319002.PDF, 1999.

[36] Intel. Intel® Streaming SIMD Extensions 4 (SSE4) Instruction Set. http://download.intel.com/technology/architecture/new-instructions-paper.pdf, 2006.

[37] Mike Murphy. NVIDIA's Experience with Open64. http://www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc, 2008.

[38] GNU. GDB: The GNU Project Debugger. http://sourceware.org/gdb/, 2008.

[39] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318, New York, NY, USA, 2008. ACM.

[40] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[41] NVIDIA. NVIDIA CUDA-GDB. http://developer.download.nvidia.com/compute/cuda/2.1-Beta/cudagdb/CUDA_GDB_User_Manual.pdf, 2008. [Online; accessed 24-November-2008].

[42] OpenGL. The Industry's Foundation for High Performance Graphics. http://www.opengl.org/, 2008. [Online; accessed 14-October-2008].

[43] Microsoft. DirectX Developer Center. http://msdn.microsoft.com/en-us/directx/default.aspx, 2008. [Online; accessed 14-October-2008].

[44] Mike Houston. High level languages for gpus overview. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 5, New York, NY, USA, 2007. ACM.

[45] AMD. AMD's Close-to-the-Metal. `http://sourceforge.net/projects/amdctm/`, 2008. [Online; accessed 14-October-2008].

[46] AMD. AMD Stream Computing. `http://ati.amd.com/technology/streamcomputing/sdkdwnld.html`, 2008.

[47] Stanford. BrookGPU. `http://graphics.stanford.edu/projects/brookgpu/`, 2008. [Online; accessed 14-October-2008].

[48] Apple. Apple Previews Mac OS X Snow Leopard to Developers. `http://www.apple.com/pr/library/2008/06/09snowleopard.html`, 2008.

[49] Federal Information. ADVANCED ENCRYPTION STANDARD (AES). `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2009. [Online; accessed 17-April-2009].

[50] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: the insecurity of 802.11. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 180–189, New York, NY, USA, 2001. ACM.

[51] Federal Information Processing Standards Publication. Advanced Encryption Standard (AES). `http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001. [Online; accessed 13-November-2008].

[52] Niyaz PK. Advanced Encryption Standard (AES) Implementation in C/C++ with comments. `http://www.hoozi.com/Articles/AESEncryption.htm`, 2008. [Online; accessed 13-November-2008].

[53] Philip J. Erdelsky. Rijndael Encryption Algorithm. `http://www.efgh.com/software/rijndael.htm`, 2002. [Online; accessed 20-November-2008].

[54] TrueCrypt. Free open-source disk encryption software for Windows Vista/XP, Mac OS X, and Linux. `http://www.truecrypt.org/`, 2009. [Online; accessed 22-April-2009].

[55] Christoph Hohmann. CryptoFS. `http://reboot78.re.funpic.de/cryptofs/`, 2009. [Online; accessed 22-April-2009].

[56] FUSE. Filesystem in Userspace. `http://fuse.sourceforge.net/`, 2009. [Online; accessed 18-April-2009].

[57] Gnome. GnomeVFS - Filesystem Abstraction library. *http://library.gnome.org/devel/gnome-vfs-2.0/unstable/*, 2009. [Online; accessed 28-March-2009].

[58] KDE. KIO: Network-enabled File Management. *http://api.kde.org/4.1-api/kdelibs-apidocs/kio/html/index.html*, 2009. [Online; accessed 28-March-2009].

[59] SSHFS. SSH Filesystem. *http://fuse.sourceforge.net/sshfs.html*, 2009. [Online; accessed 2-March-2009].

[60] Richard Jones. http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html. *http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html*, 2009. [Online; accessed 19-April-2009].

[61] NTFS-3G. NTFS-3G Stable Read/Write Driver. *http://www.ntfs-3g.com/*, 2009. [Online; accessed 28-March-2009].

[62] OpenSSL. OpenSSL. *http://www.openssl.org*, 2009. [Online; accessed 2-March-2009].

[63] iozone. IOzone Filesystem Benchmark. *http://www.iozone.org*, 2009. [Online; accessed 26-March-2009].

[64] S.A. Manavski. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. pages 65–68, Nov. 2007.