UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Urmas Rosenberg

Using Graphic Processing Unit in Block Cipher Calculations

Master's Thesis

Supervisor: Meelis Roos

Tartu 2007

Table of Contents

1 Introduction

1.1 Prolog

Development of Central Processing Unit (CPU) is approaching its physical limits, struggling at the same time with problems like high temperature and increasing need of extra power. There are mainly two alternatives to relieve those problems – Grid computing and Multi Core processors. Both of those approaches are able to give us more computing power than Single Core processors, though it means new algorithms and programs must be developed to be able to use all possibilities of new systems.

Although Grid seems to be a very sophisticated system which seems to need a room full of personal computers and Multi Core processors are expensive – all those options are available almost in every modern machine in the form of Graphic Processing Unit (GPU). Generally there are many processors in GPU and they can act like Grid – run in parallel way or same program on different data, or by Flynn's Taxonomy [1] it is Single Instruction, Multiple Data (SIMD) stream.

The rest of this paper introduces technologies used in GPU programming, algorithm of AES block cipher, implementation of this algorithm on two different graphics cards, problems that rose during creation of working code and the results.

Author wishes to thank his supervisor Meelis Roos for helping to complete this work, for giving support on programming, and also helping to get NVIDIA 8800GTS graphics card for testing (many thanks also to Andrus Laansalu and computer reseller “Ordi”); and biggest thanks to workmate Henri Kuuste who helped to create GPU programs and optimize them. Author is also thankful to his wife for understanding, supporting and helping with translation.

1.2 Overview of the Thesis

Goal of this thesis is to study possibilities of using GPU in non-graphics calculations, like cryptography. Author tries to figure out problems rising when moving arithmetic calculations from CPU to GPU and to determine when this move is reasonable. The goal is to add GPU-using functionality to an existing cryptographic framework and then test it in real-life situation. As GPUs are more powerful with bigger amounts of data, the author has chosen file encryption as the main target, leaving aside other possibilities like stream encryption. For file encryption, block ciphers are usually used as they are faster than asymmetric cryptosystems. AES block cipher was chosen as one of the most popular block ciphers today. Focus was set to testing in real-life situations rather than conducting microbenchmarks to take into account the impact to other parts of the system. Only encryption was implemented because there is no difference at programming whether to study encryption or decryption. The motivation for offloading cryptographic calculations to a general-purpose GPU is the possibility of using GPUs as cryptographic coprocessors, leaving CPU mostly free for other tasks, or replacing specialized cryptographic accelerator cards with cheaper mass-produced GPUs.

The three following sections (2-4) will give an overview of used graphics hardware and software – descriptions of graphics cards hardware, software and drivers used to utilize those cards. Some of this software has been publicly available only for a month, therefore this work could not possibly have been written earlier and on the other hand there may be some changes to them by the time this paper becomes public.

Sections describing technologies are followed by two sections that discuss about implementing the algorithm, problems that rose and the results of testing. Paper ends with conclusions. Detailed results of testing can be found in appendix A.

This work is written using open source and free software. The first reason for this choice is that author has only Linux at his computer and he is also more experienced with this operating system. Linux is well suited for lowlevel programming and gives good access to hardware. OpenSSL and other tools used in this work are also better integrated into Linux than into Windows. Second reason is that Linux is faster as confirmed by first tests where simple calculations were done on GPU, so the results should be more exact because of lower overhead.

1.3 Related work

There are three papers written which are related to the current paper:

- "Secret Key Cryptography Using Graphics Cards" [2] where authors used OpenGL and three different GPUs, getting results which were 40..100 times slower compared to CPU. This work is comparable to current paper's NVIDIA 6800 implementation.
- "AES encryption Implementation and Analysis on Commodity Graphics Processing Units" [3] where author used NVIDIA 6600GT and 7900GT. On 6600 he got approximately 7 times slower result compared to CPU (when 8 bit XOR-table was used).
- "Remotely Keyed Cryptographics. Secure Remote Display Access Using (Mostly) Untrusted Hardware" [4] where authors tested Trusted Computing on GPU.

To summary, it can be said all efforts that have been made earlier on older chips are slower than CPU. Also they have been made for microbenchmarking leaving aside other system components like HDD which is bottleneck when talking about CPU implementations.

2 Graphic Processing Unit

2.1 History

Ancestors of modern GPUs are native from late 1970s and 1980s when they had comparatively small functionality comparing to modern GPUs – they had several operations for graphics commands, some of them could combine bitmap patterns in a limited way and they could use direct memory access to reduce the load on the host processor. During that time GPUs were used even for faster printing [5].

At the beginning of 1990s, when Microsoft Windows was released, the need for faster 2D graphics rose. In 1991, S3 Graphics introduced first single-chip 2D accelerator and by 1995 already all major graphic chip makers had added this support to their chips. In mid-1990s, when CPU-assisted 3D became common in computer games it lead to an increasing public demand for hardware accelerated 3D graphics. First chips that arrived were not pure 3D accelerators but 2D chips which included also some 3D features. DirectX became one of the leading 3D graphics programming interfaces and chips got 3D rendering pipeline. For some cards the graphic acceleration was not the only thing to do – for example NVIDIAs NV1 could also manage sound and video and act as a modem [6][7].

In 2000 and later, programmable shading got added to GPUs' capabilities which meant that every pixel could be separately processed by a short program. NVIDIA was first on the market with programmable shading and in 2002 graphic chips were already able to make loops and lengthy floating point calculations – they quickly became nearly as flexible as CPUs [5].

2.2 Architecture

Today, parallel GPUs are offering generally quite good computation power compared to latest CPUs (as seen on figure 2.2.1) and have found their way to several different application types [5].

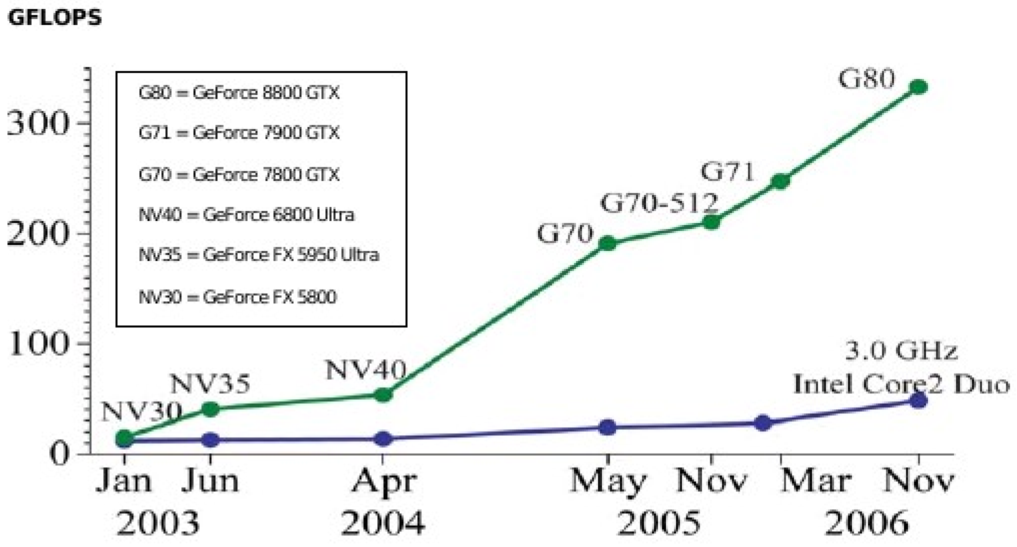


Figure 2.2.1: Floating-Point Operations per Second for the CPU and GPU [8]. As seen – GPUs can give remarkably more computing power and the progression of GPUs is also much faster.

The main reason behind such an evolution is that GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore is designed so that more transistors are devoted to data processing rather than data caching and flow control [8][9]. Also, as seen on figure 2.2.2, compared to the CPU GPU devotes more transistors to data processing meaning that work can be done faster.

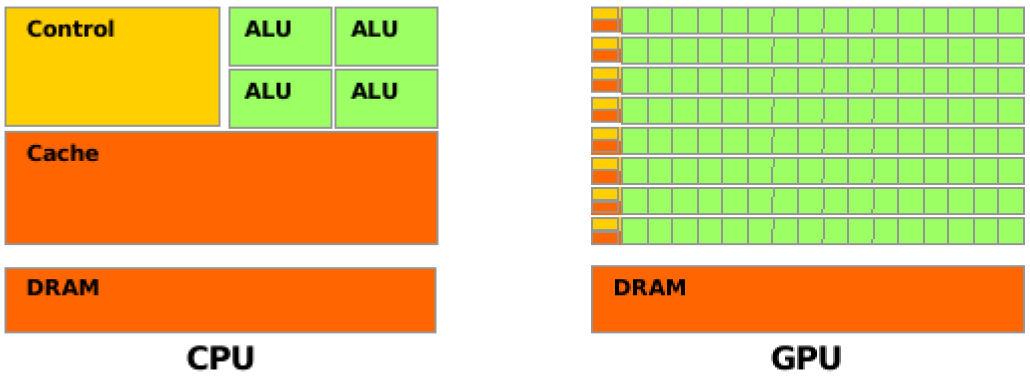


Figure 2.2.2: The GPU devotes more transistors to data processing [10].

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations. Because the same program is executed for each data element, there is lower requirement for sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches [10].

For current thesis this means several things. First, we must reconsider almost everything about optimization on CPU. For example, AES optimization on CPU uses several lookup tables, but on GPU these tables would slow things down instead. Second, it must be possible to calculate the algorithm in parallel. It means that GPU is not suitable for (cryptographic) algorithms where calculation of one data block is related to its prior data block calculation. Third, there must be considerably more arithmetic complexity compared to memory operations.

2.3 Specifications

For current work two different graphic chips were used – NVIDIAs GeForce 6800 GT and 8800 GTS. We now take a brief look at the specifications and general information of those cards.

ASUS GF6800GT PCX

This card is advertised on manufacturers page (http://www.nvidia.com/page/geforce_6800.html) as following: "The groundbreaking new NVIDIA GeForce 6800 GPUs and their revolutionary technologies power worlds where reality and fantasy meet; worlds in which new standards are set for visual realism and quality, performance, and video functionality. The GeForce 6800 GPUs deliver powerful, elegant graphics to drench your senses, immersing you in unparalleled worlds of visual effects for the ultimate PC experience."

This card has Geforce 6800 GT core running at 350MHz and 256 megabytes of GDDR3 memory running at 1GHz. This card uses PCI Express system interface. Memory transfer rate is 32GB/s. It has 16 pixel processors and 6 vertex processors, in our implementation are only pixel processors used.

- http://www.asus.com/products4.aspx?l1=2&l2=6&l3=138&model=406&modelmenu=2
- http://www.viperlair.com/reviews/video/asus/nv4x/6800gt/
- http://www.guru3d.com/article/content/151/

Club 3D 8800GTS

This card is advertised on manufacturers page (http://www.club3d.nl/index.php/products/graphics/item/231) as following: "This is the first Graphics Card of a completely new generation. The Club 3D 8800 is not only limited to boosting the speed and the image quality of your games. With the new developed Quantum Effects engine it will also accelerate movements and particles known as the physics from your game. This will make explosions, movements and collisions look more impressive and realistic."

Club3D 8800GTS has 640 megabytes of GDDR3 memory running at 800MHz and 12 multiprocessors at 600MHz (Club3D homepage says at 500Mhz). Each multiprocessor is composed of eight processors running at twice the clock frequency of multiprocessor. Difference between 8800 and 6800 is also processors ability to make computations – 6800 processors can do only prescribed operations, vertex or pixel calculations, but processors on 8800 can do those calculations which are more needed at the moment.

8800 family chips have new architecture compared to earlier NVIDIA chips. NVIDIA has released a new software framework named CUDA (introduced in section 2.8) which makes easy to use graphics cardspossibilities in programs being developed.

The memory transfer rate is 64GB/s and it is much higher than the bandwidth between the device memory and the host memory. In 0.8 beta version of CUDA, the maximum observed bandwidth between system memory and device memory is 2GB per second [10].

Club 3D says [11] about that card: For optimal performance with the CPU, the fastest bus communication PCI-Express x16 technology is used. This will allow communication with 4GB/s in each direction between VGA card and CPU [12]. This statement gives some hope that in final versions of CUDA, the memory transfer rate will be faster than the current rate.

The memory transfer speed between CPU and GPU also makes a difference in cipher implementations. When block size 4096 is used, there is no chance for GPU to be faster than CPU if there is no urgent need for arithmetic power in given algorithm. For that reason we also changed OpenSSL's encoding block size. More detailed data about that will be given in sections “AES on GPU” and “Results”.

Differences

From programmers point of view, the main differences between those two cards are

- 8800 supports bitwise logic and shift operations
- 8800 gives more flexible access to memory, for both reading and writing.
- Through CUDA it is possible directly define data on GPU, no need for extra copying.
- 8800 can modify more data in one function. When 6800 just returned float4 (16 bytes) as color after end of execution, 8800 implementation is changing 160 bytes directly in memory by default configuration.

2.4 OpenGL

Open Graphics Library is a standard specification defining a cross-language cross-platform API for writing applications that produce 2D and 3D computer graphics. The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. in 1992 and is popular in the video games industry where it competes with Direct3D on Microsoft Windows platforms. OpenGL is widely used in CAD, virtual reality, scientific visualization, information visualization, flight simulation and video game development [13].

OpenGL describes a set of functions that are implemented in device driver. We use NVIDIA's binary driver which is not open source. On driver lays yet another layer which hides real function calls and lets user see this as normal OpenGL implementation. There might be some doubt – could one write better and faster programs if he could get direct access to the card, but at the moment it is not even the most important. By using OpenGL for this implementation we have an advantage compared to CUDA implementation – it can also be used with ATI video cards, as CUDA is only able to run with NVIDIA cards, at the moment.

As designers of OpenGL anticipated the need to extend OpenGL in the future [14] there is possibility to add extensions to OpenGL, for example by hardware vendors, and therefore there is no immediate need for new releases of OpenGL as new features are developed.

OpenGL's homepage is available at http://www.opengl.org/.

2.5 GLUT

The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres, and the Utah teapot. GLUT even has some limited support for creating pop-up menus. The two aims of GLUT are to allow the creation of rather portable code between operating systems (GLUT is cross-platform) and to make learning OpenGL easier. Getting started with OpenGL programming while using GLUT often takes only a few lines of code and requires no knowledge of operating system–specific windowing APIs [15].

In this work we use the GLUT library for initialization (to establish a session with windowing system) and create and hide a window. We can also notice a restriction of one of the created implementations here – 6800 code must be running in a graphical environment because window creation without X (or any other graphical environment where needed libraries can be used) is impossible.

2.6 GLEW

The OpenGL Extension Wrangler Library (GLEW) is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions. It also provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform [16].

When vendor releases new hardware and a new OpenGL extension, it contains functions with specific names. With OpenGLs own tools it is difficult to test which extensions are available and call those functions. GLEW makes it possible to load those extensions and to test if they exist on the current system – and it also redefines those functions into usable standard form.

GLEW library is available at its homepage: http://glew.sourceforge.net/.

2.7 Cg Toolkit

The Cg (C for graphics) is programming language developed by NVIDIA to make programming on graphics hardware easier. It is based on C [17], as this is very popular programming language, and it removes the need for developers to program directly in the graphics hardware assembly language [18][20].

The Cg Toolkit provides a compiler for the Cg language, runtime libraries for use with both leading graphics APIs (OpenGL and DirectX), runtime libraries for CgFX, example applications and extensive documentation. Supporting over 24 different OpenGL and DirectX profile targets, Cg allows incorporating interactive effects into 3D applications [19].

Although Cg is based on C and its constructions are easy to read when familiar with C, there is one thing to pay attention to – data types. There are well known types in like int and float, but new ones like half, half4 and float4:

- The half type is lower-precision IEEE-like floating point [17].
- half4, float4 are vector types, containing 4 numbers of respective type. There are more defined vector types for every standard type [17] but they are no longer discussed here.

Running Cg compiler separately is easy:

```
cgc -profile fp40 -o outputfile inputfile
```

So source file can be compiled into machine code, which is then loaded into GPU. fp40 here means that output code is generated specifically for NVIDIA's fp40 profile as used in 6800 chip. In current work .cg file is converted to GPU code by cg function *`cgCreateProgramFromFile`* which chooses the right target profile automatically.

It must not be forgotten that almost each GPU family is different from others which means that the right profile must be chosen. At the time of writing this paper the 8800 chip is not yet supported by cg, so no comparison can be done on this level for those cards, as 8800 is programmed by using CUDA. This will be discussed in the next section.

Cg is available at its homepage: http://developer.nvidia.com/object/cg_toolkit.html

It is important to point out that version 1.5 of cg came out in February 2007. Previous version (1.4) of cg generated invalid machine code from source. The possible reason, as the tests proved, was that the compiler couldn't handle so many variables as were used by AES implementation. With version 1.5 that problem disappeared.

2.8 CUDA

CUDAs homepage [8] advertises this product the following way: "CUDA (Compute Unified Device Architecture) technology is a fundamentally new computing architecture that enables the GPU to solve complex computational problems in consumer, business, and technical applications. CUDA technology gives computationally intensive applications access to the tremendous processing power of NVIDIA GPUs through a revolutionary new programming interface. Providing orders of magnitude more performance and simplifying software development by using the standard C language, CUDA technology enables developers to create innovative solutions for data-intensive problems. For advanced research and language development, CUDA includes a low level assembly language layer and driver interface".

When programmed through CUDA, the GPU is viewed as a computing device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU. A portion of an application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device as many different threads. To achieve that effect, such a function is compiled to the instruction set of the device and the resulting program is downloaded to the device [10].

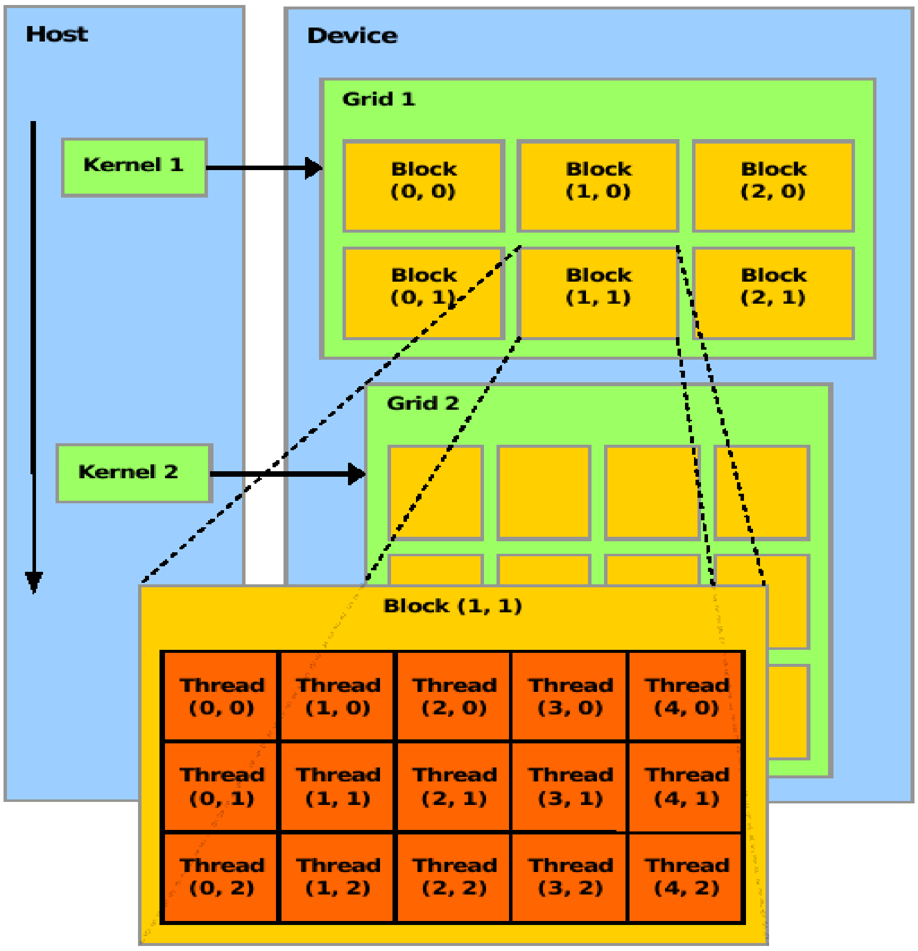


Figure 2.8.1: 8800 Thread Batching [10]

Figure 2.8.1 illustrates how GPU is seen as a computation device – function, called a kernel, is going to run on a grid (see figure 2.8.1), which has predefined configuration given on call. The number of blocks in grid and the number of threads going run in a block are configurable. Example of running a kernel:

```
encrypt_8800<<<grid,threads>>>((uint*)d_idata,
(uint*)d_odata, isbox, (uint*)ikey);
```

Only number of blocks and threads are defined here but it is possible to also declare the amount of shared memory (see figure 2.8.2) which will accessible for threads within one block, but as this is not useful in current implementation, it is not used. All data, which includes key, *sbox* and data to encryption, will lay in Global Memory (see figure 2.8.2) and local variables are in registers.

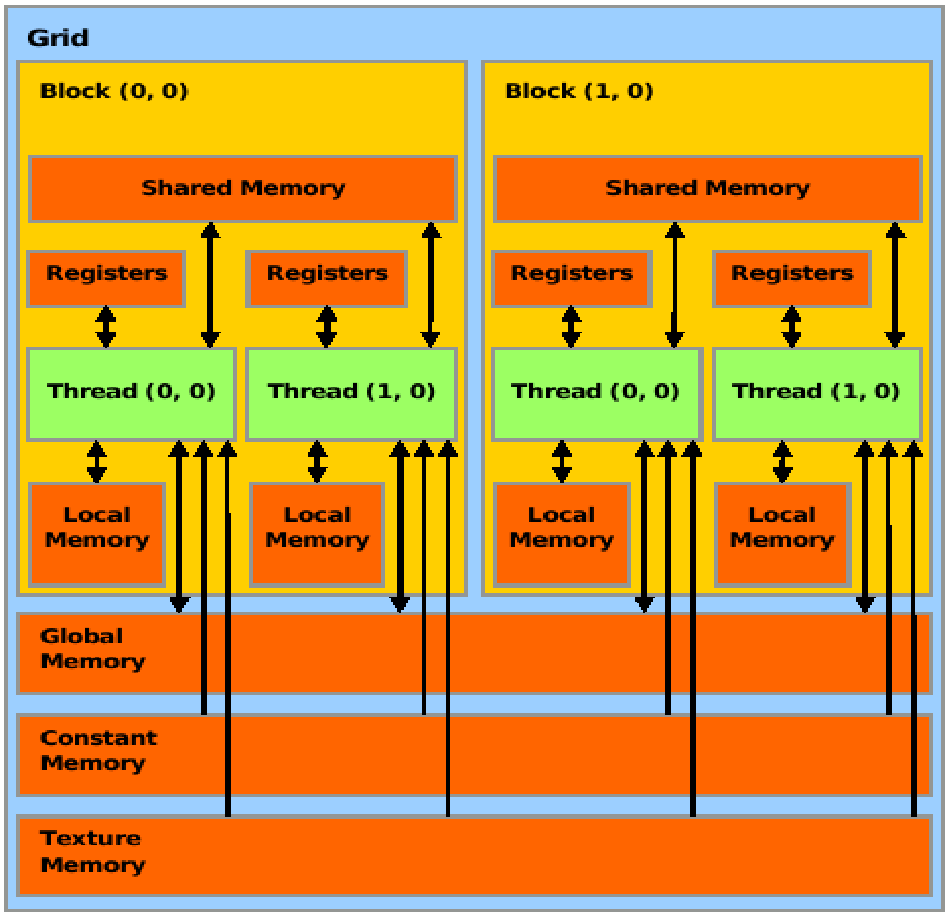


Figure 2.8.2: 8800 Memory Model [10]

3 Advanced Encryption Standard

In cryptography, the Advanced Encryption Standard (AES), also known as Rijndael, is a block cipher adopted as an encryption standard by the U.S. Government. It became effective as a standard on May 26th in 2002. As of 2006, AES is one of the most popular algorithms used in symmetrickey cryptography [20].

In current paper the author focuses only on AES with 128 bit key length, although AES supports also keys with lengths of 192 and 256 bits. Using particular length of key is not very important because it doesn't change algorithm significantly in terms of intense of calculations, and full implementation is not the goal of this paper.

3.1 Algorithm

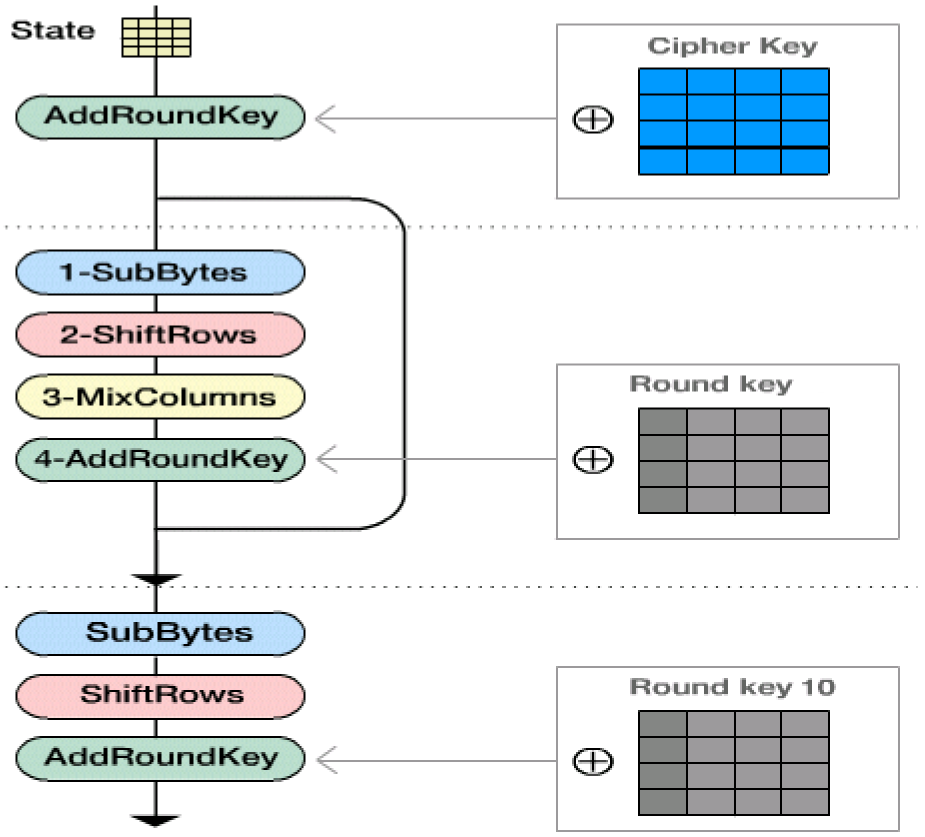


Figure 3.1.1: Encryption process [21].

Figure 3.1.1 shows AES encryption graphically to give a better overview of the whole process. After initial `AddRoundKey` transformation with ten rounds of `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` transformations, on the final round the `MixColumns` transformation is left out from the chain. State is data going to be encrypted, round keys are blocks of the same size as data and they are derived from cipher key using Rijndael key schedule.

`AddRoundKey` – corresponding bytes from data and key are combined by using bitwise XOR.

`SubBytes` – bytes are replaced according to lookup table.

`ShiftRows` – rows are rotated to the left (1st row stays same, 2nd shifted one step, 3rd two steps and 4th shifted three steps)

`MixColumns` – four bytes of each column are mixed by using bitwise XOR and special function which can be replaced with a lookup table.

3.2 Optimization

Optimizations for CPU often mean replacing calculations with table lookups, but on GPU there are complications with memory reads when comparing to CPU. GPU can do more arithmetic operations in one processor cycle than CPU but when reading from memory there is latency and while processor waits for a response on the execution is stalled. Therefore it is very important to measure time ratio between memory reads and arithmetic operations when implementing algoirthms on GPU.

According to Wikipedia [20] AES algorithm can be optimized on 32 bit system by converting *`SubBytes`*, *`ShiftRows`* and *`MixColumns`* transformations into tables. It is useful on CPU, but when programming on GPU, we must consider that there will be 200 to 300 clock cycles of memory latency [10:53], so there is a big chance that making extra calculations on GPU is still faster than memory lookup.

As seen later in results' section, it really depends on arithmetic intensity, whether one or the other method is faster, so the best results will be gotten by measuring the results of testing.

4 OpenSSL

The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The project is managed by a worldwide community of volunteers who use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation [22].

The core library (written in the C programming language) implements the basic cryptographic functions and provides various utility functions. Wrappers allowing the use of the OpenSSL library in a variety of programming languages are available [23].

OpenSSL implements many cryptographic algorithms and additionally gives users a possibility to use special-purpose accelerator hardware. This is done by using cryptographic engines which will communicate with the special-purpose hardware and perform encryption/decryption processes on them. Minimum set of functions which must be declared in an engine is:

- function that creates a new engine instance
- function that registers id, name, init an finish functions
- function that returns the list of supported ciphers.

Our benchmarking is focused on OpenSSL's command line functionality and other possibilities are left out of focus. Example of running OpenSSL on testing:

```
openssl aes-128-ecb -bufsize 16777216 -in test1.dat -out data.cpu -k abcd -nosalt
```

where “aes-128-ecb” means that AES cipher will be used for encryption, it is in ECB mode (it means that key is unchanged when encrypting different data blocks) and it will use 128 bit key. “*-bufsize 16777216*” means that OpenSSL will read 16777216 bytes of data at once and will forward it to the engine for encrypting. Still, the engine will not get directly the same size of buffer as determined on command line but it is chunked a smaller size defined in OpenSSL file *crypto/evp/bio_enc.c* – *ENC_BLOCK_SIZE*, which is 4096 bytes by default. But as mentioned earlier, and shown later, GPU can't give good results on small data blocks, so this value is changed during testing. “*-in*” and “*-out*” are describing input and output files correspondingly. “*-k*“ sets password to “abcd” and “*-nosalt*” means that there is no random data added to the key. This is useful for repeatability of the results, and to test whether CPU and GPU get the same results.

It is important to bear in mind that the ECB version of AES is used because that algorithm could be calculated in parallel way, but versions that change key depending on previous data would make this impossible. But if no parallel computing is possible to apply, there is no sense in using the GPU.

5 AES on GPU

5.1 Creating a new OpenSSL engine

Work on creating a new engine for OpenSSL started at studying the whole code base. It must be said that OpenSSL is quite confusing for new developers. Internal documentation is almost missing, there are few comments in code and a lot of function definitions and calls are made through preprocessors definitions and through many layers of abstraction.

First goal of the current work was set to create an empty engine and learn how data gets there. To let OpenSSL know that there is a new engine, it must be loaded by *ENGINE_load_builtin_engines* function (*eng_all.c* in *crypto/engine* directory) like this:

```
ENGINE_load_gpu();
```

The alternative way of loading additional engines would be loading the engine explicitly from the application that uses OpenSSL, but since we build our engine into OpenSSL, we modified OpenSSL to also automatically load it. In function *ENGINE_load_gpu* GPU implementation creates a new engine instance and adds pointer to it into global list of known engines. In initialization process function *gpu_bind_helper* is called which registers important data and functions into ENGINE function table – engines id, name, init and finish function, and finally a function which returns the list of implemented ciphers. It is called always when OpenSSL is started.

If the engine is not specified for the command-line utility, OpenSSL internal engine is used, assembler or C version of AES. We need to add new keyword to command line to activate and use our new engine:

```
openssl aes-128-ecb -bufsize 16777216 -engine gpu -in
test1.dat -out data._gpu -k abcd -nosalt
```

Engine is chosen by its ID which is set in helper function. If no such engine is registered in OpenSSL, it will fall back to C or assembler implementation.

From here on there will be a difference between 6800 and 8800 chip implementations. Although function calls are the same, they have different inner implementation. Overall process on encrypting/decrypting is the same: *gpu_init* → *gpu_aes_init_key* →

gpu_aes_cipher → *gpu_finish* where *init* does all initialization needed to be able to use GPU, *init_key* initializes key according to AES key schedule algorithm, cipher encrypts given block of data and finish frees used memory.

And now a few final changes to original OpenSSL code to make the new engine work.
6800 implementation changes:
in root directory *Makefile.org* must be changed, *PEX_LIBS* should be

```
PEX_LIBS= -lCg -lCgGL -lglut -lX11 -lm -lpthread -lGLEW
```

It forces compiler to use libraries needed for graphical programming.
And in *crypto/engine/Makefile* "*eng_gpu.c*" should be added to *LIBSRC* and "*eng_gpu.o*" to *LIBOBJ*. After that one should run "*make depend*" command in OpenSSL source directory and then the code will be ready to use.

8800 implementation changes:
only *crypto/engine/Makefile* must be changed:

```
all: lib
```

should be replaced with

```
all: cuda lib

cuda:
     nvcc -c -o eng_gpu.cu.o eng_gpu.cu -I
/usr/local/cuda/include -I/opt/NVIDIA_CUDA_SDK/common/inc
-DUNIX -O3
```

And "*eng_gpu.c*" should be added to *LIBSRC* and "*eng_gpu.o eng_gpu.cu.o*" to *LIBOBJ*. This additional target will compile CUDA file into object file which can be linked to OpenSSL binary. After that one should run "*make depend*" command in OpenSSL source directory and then the code will be ready to use.
For a note – "*-I/opt/NVIDIA_CUDA_SDK/common/inc*" should refer to directory where NVIDIA CUDA software development kit is installed.

5.2 Implementing AES algorithm on GPU

In this section the author will give overview of 6800 and 8800 implementations.

When GPU engine is chosen, first function called before encoding/decoding is `gpu_init` which does all GPU initialization.

5.2.1 6800 implementation

Data is sent to GPU as textures which means that data is treated as picture where one pixel contains 16 bytes of data.

First variable defined is `texSize` – it will represent the length of texture edges (square) in which data (which should be encrypted) will be copied to GPUs memory. Maximum texture size is 4096 x 4096, so texSize must be between 1..4096.

When data is mapped into GPUs memory, its size will be 16*texSize*texSize bytes. It's 16 times bigger than defined texture but it's not a mistake – 16 bytes will be represented as one pixel, containing four 32bit floats (Red, Green, Blue, Alpha), which are all in turn separately "unpacked" to four bytes.

After the GPU is initialized static data (`xor`, `sbox` and `mix` tables) will be written into GPUs memory. `Xor`, and also `mix` is set into table not because of the optimization but due to the fact that 6800 chip can't do bitwise logic operations so this data will be precomputed on CPU. `Mix` is subdata for `MixColumns` operation.

The next step is to initialize cg ("C for Graphics") – context is created, GPU profile data initialized and cg program loaded and translated into format of current profile. Final steps in initialization are creating named parameters and making them related to corresponding textures.

Function named `gpu_aes_init_key` is called once before real encyption/decryption starts. As one of the input parameters is `key`, 16 bytes, it will be expanded by Rijndael's key schedule algorithm and after that key data is copied to GPU. Currently `gpu_aes_init_key` will return error if action is something else than encryption, because only encryption is currently implemented.

From CPU side the most important function is `gpu_aes_cipher` which handles the following things:

- copying data to GPU with OpenGL function `glTexSubImage2D`
- starting GPU side program by calling `glBegin`
- after GPU is done, copying data from GPU back to main memory with function `glReadPixels`.

This function is called as many times as *`ENC_BLOCK_SIZE`* fits into data size and additionally one last time with final data. Here lays also a possibility to win some extra time by making `glTexSubImage2D` and `glReadPixels` depend on the number of bytes going to be encrypted (`nbytes`).

6800 GPU side implementation lays in file `gpu_aes_encrypt.cg` and it contains code written in cg.

Cg file can also be compiled separately forfast testing:

```
cgc -profile fp40 -o outputfile inputfile
```

For every pixel on input texture a main function is called with return type float4 (which in current implementation means 16 bytes). Main function gets the following input parameters:

- `coords` – float2 (two floats) indicates to the program which pixel on texture must be handled by this current call
- `data` – input texture, two-dimensional array of data which will be encrypted. It is accessed by normal int type indexes
- `key` – key and its expanded subkeys
- `sbox`, `mix` and `xor` – precalculated tables with results to logical calculations which could not be done in GPU.

In code, there is roughly 7 different blocks, which will now be closely observed:

1) initial block, where data is prepared for calculations
2) add round key
3) subbytes
4) shift rows

5) mix columns
6) add round key
7) final block, data is packed and prepared for returning

Blocks 3, 4, 5 and 6 are in for loop (described in AES algorithm section) and block 5 is left out on final round.

In block 1:

```
float4 cf = tex2D(data, coords);
```

Cf represents data matrix, containing 16 bytes which will be encrypted. The key will later be extracted in the same way. But `cf` is not usable in this format (`cf.x` is 32 bits, 4 bytes), so we need to “unpack”it:

```
half4 c1 = ceil(unpack_4ubyte(cf.x)*255);
```

c1..c4 will now hold columns (data is represented in column-major order) in 4x4 matrix. As half4 is a type containing four elements, four first bytes are accessed like this: `c1.x`, `c1.y`, `c1.z` and `c1.w`. Multiplication with 255 is needed due the fact that in initial state all numbers are in float format between 0..1 but for table lookup we need it in “byte” format.

As data and key are already available, it is time to do first step of the ciphers – *addRoundKey*, block 2:

```
c1.x = texRECT(xor, int2(c1.x, kr.x)).x;
```

It means that the first element in the first column (`c1.x`) gets a new value which is the first value (*textRECT* will return float4) from result of xor table at coordinates `c1.x` and `kr.x` (first key byte). For now, first byte is calculated, and in that way it is repeated for the rest of the 15 bytes. Next four bytes of the key are read when four bytes of data is done. Reading the key between calculations is done in that way because it gives GPU more chance to make calculations while reading data from memory.

SubBytes operation is similar to XOR operation, block 3:

```
c1.x = texRECT(sbox, int2(c1.x, 0)).x;
```

As seen, the xor table is just replaced with the sbox table. As it is a two-dimensional texture with only one row, then the second coordinate is 0.

Block 4 contains shiftRows operation where matrix rows are rotated left getting the following result:

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

1	5	9	13
6	10	14	**2**
11	15	**3**	7
16	**4**	8	12

In block 5 `mixColumns` operation can be seen, this is half-way implemented with bok-up table due to the lack of bitwise operations. Algorithm of this operation is nicely discussed on Wikipedias page: http://en.wikipedia.org/wiki/Rijndael_mix_columns. C implementation is there given as following:

```
void gmix_column(unsigned char *r) {
        unsigned char a[4];
        unsigned char b[4];
        unsigned char c;
        unsigned char h;
        /* The array 'a' is simply a copy of the input array 'r'
         * The array 'b' is each element of the array 'a' multiplied by 2
         * in Rijndael's Galois field
         * a[n] ^ b[n] is element n multiplied by 3 in Rijndael's Galois field */
        for(c=0;c<4;c++) {
                a[c] = r[c];
                h = r[c] & 0x80; /* hi bit */
                b[c] = r[c] << 1;
                if(h == 0x80)
                        b[c] ^= 0x1b; /* Rijndael's Galois field */
        }
        r[0] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1]; /* 2 * a0 + a3 + a2 + 3 * a1 */
        r[1] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2]; /* 2 * a1 + a0 + a3 + 3 * a2 */
        r[2] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3]; /* 2 * a2 + a1 + a0 + 3 * a3 */
        r[3] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0]; /* 2 * a3 + a2 + a1 + 3 * a0 */
}
```

from which the following part is implemented with table look up:

```
        for(c=0;c<4;c++) {
                a[c] = r[c];
                h = r[c] & 0x80; /* hi bit */
                b[c] = r[c] << 1;
                if(h == 0x80)
                        b[c] ^= 0x1b; /* Rijndael's Galois field */
        }
```

As seen on visualization (table 5.2.1.1) of output of this chunk of code, it is quite symmetrical, so it is also implemented as function `mix2`

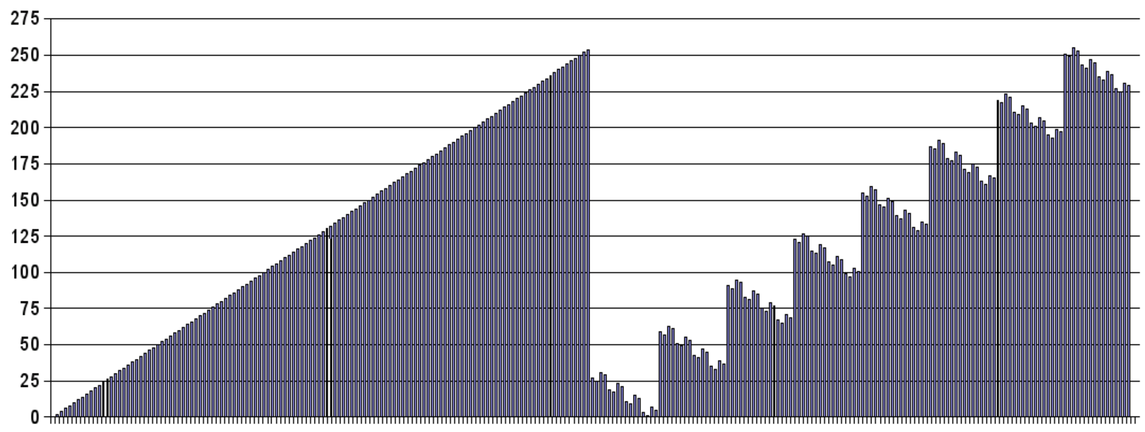


Table 5.2.2.1: Visual representation of subtable in *mix_table* function.

```
half mix2(half b){
      half x;
      if(b<=127){
            x=b*2;
      }else{
            x=b-128;
            half k=(int)(x/16); //number of block
            x=((int)x%16)+1;
            x=16-x;
            half n=(int)x/4;
            half diff=(n*2+1)*2-x;
            x=(diff+((int)x%2)*2+n*4)*2;
            x=x+k*32+1;
      }
      return x;
}
```

On 6800 chip this function is slower than table lookup, so it is commented out, but there is a big chance that on newer chips it turns out to be faster.

Block 6 is the same as block 2, only a new key is read from memory, and finally in block 7 data is divided by 255 to transform it back to “float form” and packed into one variable which is returned as color.

There is also *xor2* function left in cg file (commented out) which can replace xor table lookups. It is much slower than memory lookup itself but it is still interesting to test on newer cards.

5.2.2 8800 implementation

Programming with CUDA is much faster, easier and shorter compared to OpenGL as used in 6800 implementation – for example there is no need for extra initialization or transferring data to GPU. Mix table data can be defined as device-side data directly:

```
__device__ int mix[] = {
```

There is some difference in overall structure compared to 6800 implementation – while in the case of 6800 there was a new C file created which did all GPU initialization and static data moving to GPUs memory and GPU-side code in cg file is loaded and compiled on the fly, in the case of 8800 all GPU-related code is in cu file. OpenSSLs side still has functions *`gpu_init`*, *`gpu_finish`*, *`gpu_aes_init_key`* and *`gpu_aes_cipher`* but they just make calls to functions implemented in file *`eng_gpu.cu`*. During the building process, this file is compiled separately by nvcc (compiler from CUDA) to object file and at the end linked with other object files as usual.

Before giving an overview of functions, some information about defined constants is necessary:

- *`NUM_BLOCKS`* – how many blocks will be run on GPU (See figure 2.8.1). For better performance it should be multiple of 12 (number of multiprocessors), at least 24 in case of 8800GTS [10:49].
- *`NUM_THREADS`* – how many threads will be run in each block. Thread should be considered as unit computing one pixel. Maximum number is 512 [10:49] while 64 is minimal and better is 192 or 256 threads per block [10:63] in case of 8800GTS.
- *`ROUNDS_IN_THREAD`* – Although a thread should be treated as a process for calculating one pixel, or one 16 bytes block of data, tests showed that it is possible to gain better results if one thread can do more calculations.

Functions in eng_gpu.cu file, CPU side functions

- *`init_8800`*: initialize GPU. *`CUT_CHECK_DEVICE`* macro is defined in cutil.h and this file can be found under SDK. After initialization memory on GPU is allocated for *`sbox`*, input- and outputdata. Also sbox is copied to GPU. *`Sbox`* could also be defined as __*`device`*__ type and therefore there would be no need for memory allocation but *`sbox`* is also used on CPU-side – as the key is still expanded on CPU.
- *`finish_8800`*: frees all allocated memory, will be called as the last function after

encryption process is completed.

- `init_key_8800`: key expansion
- `cipher_8800`: copies data to GPU, calls GPU-side function `encrypt_8800` (with extra parameters which tell how many blocks and threads must be run on GPU), and as GPU has finished, copies encrypted data back to CPUs side memory. Here lays also one point which can be improved – size of memory (which is copied and encrypted) doesn't take into account that the last block of encryption is only 16 bytes.

Two last functions, `mix_column` and `encrypt_8800` are GPU side functions:

- `mix_column`, with keyword `__device__`, is callable only from device, implements Rijndael column mixing. Compared to Wikipedias C implementation it has some improvements – one variable is left out, shift to left is done once on all four bytes, also XOR with 0x1b (constant coming from algorithm) is done once on all bytes.
- `encrypt_8800`, with keyword `__global__`, also named as "kernel" and is callable from host (CPU side) only, is AES encryption implementation.

As programming with CUDA is like programming in C then the algorithm itself will not be longer discussed here. Comparing to 6800 implementation the new things are the way memory is used and the way data gets into the encryption process. First, there are two new built-in variables – `blockIdx.x` and `threadIdx.x`. The first one says which block is currently running and second one says which thread is in current block. As data is held in global memory and "kernel" gets only a pointer to it, function must calculate the address where to start reading data for encryption. The formula for that is:

```
unsigned long int sb = (bid * NUM_THREADS * ROUNDS_IN_THREAD * 4) + (tid * ROUNDS_IN_THREAD * 4);
```

As blocks' and threads' numbers are starting from zero, first thread from first block becomes the starting byte address 0 in pointer. Later ones must already consider that the maximum number of bytes is the number of threads started in block multiplied by number of rounds in one thread multiplied by 4 (32bit unsigned integers are used).

For a note, there is one more benefit programming with CUDA compared to programming in OpenGL – CUDA code can be compiled with an extra parameter `-deviceemu` which gives possibility to run CUDA code on the host without GPU. It is useful for testing, because neither CUDA environment nor GPU gives any debugging support.

5.3 Problems

In this section the author will discuss problems experienced during implementing AES block cipher on GPU.

When talking about 6800 chip, problems can mostly be divided into three groups: problems related to limitations of the chip, problems related to limitations of the programming tools and finally physical problems.

Problems related to chip

- 6800 does not have bitwise logical operations, so all bitwise calculations must be implemented as lookup tables, but on GPU it's a slow operation. There is a native XOR at the end of the rendering pipeline, but it is unusable because this operation is applied only to the final stage ofthe rendering process [3].
- Natively GPU handles floats faster than integers, but on table-indexing there is need for integers. So all numbers used are not byte, but integers and float. This casting takes some time. First tests on XOR table were for example indexed by float, but problems occurred at bytes above ASCII code 240 (due to progressive error in distance between floats if they are handled as integers).
- As XOR is implemented as a table, there is no possibility to make fast calculations like four bytes XORed by four bytes at once because it would take too much memory.

Problems related to tools

- cg is still in stage of development. Problems like buggy compiler rose due to too large amount of variables with cg version 1.4.
- Graphical environment is needed to run this 6800 implementation.

Physical problems

- Testing on 6800 gave interesting results – when a 500MB file was encrypted the result was correct, but 700MB file was full of errors at the end. So what's the catch? As problem was studied more closely author discovered that the chip was starting to give errors when its temperature went above 114 degrees by Celsius. For authors fortune, problem disappeared when cards cooling system was cleaned from dust. After the cleaning, GPU's temperature was around 65 degrees by Celsius in the middle of calculations.
- Copying data to GPU is a relatively slow action, therefore GPUs algorithm must be really fast to win back time lost in copying.

- Copying bigger portions of data is a considerably hard task to CPU, it means that big chunks are making performance results worse.
- At the end of encryption GPU algorithm makes an extra block by size of 16 bytes. But currently GPU doesn't count on real size of data, so it spends the same time as usual on that last block, depending on testing settings, for example 16MB.
- Results, while testing on small files, vary very much (see Table A6, file #7). Reasons for that may lay in cache of both operating system and GPU.

Problems encountered while creating 8800 implementation were mostly related to the tools. Although, memory copying speed issue is mostly the same, and as 6800 implementation, 8800 implementation doesn't also count the last block size.

- CUDA is beta I. For example – there is up to 16KB shared memory per multiprocessor, but if larger amount of shared memory is allocated at running CUDA program, it doesn't give any error but returns some random result.
- CUDA is beta II. When errors occurred (allocated too much memory, etc), CUDA program failed, printing just an error message "terminate called after throwing an instance of 'bool'". It made debugging difficult.
- CUDA is beta III. At some point, after a line "unsigned char r[4];" there were no errors as there was assignment to `r[0]..r[2]`. Assignment to r[3] showed up an error message:

```
### Assertion failure at line 1432 of ../../be/cg/cgemit.cxx:
### Compiler Error in file /tmp/tmp_00007a9f-1.i during Assembly phase:
### incorrect register class forresult 0
nvopencc INTERNAL ERROR: /usr/local/cuda/open64/lib//be returned non-zero status.
```

As 6800 implementation relies on OpenGL standard, it is independent from graphics card used (it must be supported in cg's profiles). 8800 implementation relies on NVIDIA's CUDA and graphics driver, so it is unusable with cards manufactured by other firms or by NVIDIA's older cards.

One reason to use 8800 as a coprocessor is its speed and tests (for example "simulation of the dispersion of airborne contaminants in the Times Square area of New York City" [24] which showed that GPU cluster is 4.6 times faster than CPU cluster) empower those expectations, but speed-bonus may get shadowed by power consumption problems and by alternatives which do not need so much power. When comparing pictures C1 and C2 it shows that in one case 8800GTS system uses 81 watts more than idle system. Pure numbers vary, but one number said [25] is that G80 family (it means 8800 chip) uses about 177W when loaded. Considering that AMD wants to add new coprocessor [26] which is able to calculate 25GFlops at only 10 watts, it might end up NVIDIAs wish [27] to enter into high-performance computing market with current GPUs.

6 Results

6.1 Description of test system

All programming and testing were done on one PC using two different cards to test different implementations:

PC
CPU: Intel P4 640 3.2GHz
RAM: 2 x 512MB DDR2
Motherboard: Abit NI8-SLI C19 s.755 DDR2
HDD: Western Digital, 250GB, SATAII

6800 card
http://www.asus.com/products4.aspx?l1=2&l2=6&l3=138&model=406&modelmenu=2
GPU: NVIDIA 6800GT
RAM: 256MB

8800 card
http://www.club3d.nl/index.php/products/graphics/item/231
GPU: NVIDIA 8800GTS
RAM: 640MB

Operating System
Name: Gentoo Linux
Kernel: 2.6.18-gentoo-r6
gcc version: 3.4.6
glibc version: 2.5
X: 7.2

KDE: 3.5.5

Graphics and other libraries
video driver: NVIDIA 9751 (CUDA compatible)
nvidia-cg-toolkit version: 1.5.0
glew: 1.3.5
freeglut: 2.4.0
CUDA Toolkit: 0.8
CUDA SDK: 0.8
OpenSSL: 0.9.8d

6.2 Speed tests

AES implementation was tested and compared in “real-life situation” with OpenSSL where “user” had a file on hard drive and it had to be encrypted with AES algorithm. Tests also included error checking – will CPU and GPU implementations get the same result? As talked in “Problems” section, for example, one “bug” was found when comparing results with cmp:

```
cmp data.cpu data.gpu
```

and although small files (below 500MB) were correctly encrypted 700MB files already got errors at the end while using 6800 card.

First tests were focused to find configurations of implementations (such as buffer size, texture size, etc.) which give best results and then stay focused on them. Some testing was also done during programming to also find out which area of code takes most of the time of execution. It was done by commentingof lines or blocks and then comparing results. In that way a lot of optimizations were made which now lay in final code. But this sort of testing needs attention – both compilers, cgc and nvcc, eliminate all calculations which are not related to return value, so for example when one comments out the last line in some blocks where there is ascription and therefore code becomes several times faster, it doesn't mean that loss of time happens because of this ascription.

For final testing, 8 different files were generated, although last of them got less attention due to big variance among results.

#	Name	Size (bytes)	Content
1	test1.dat	884 736 000	Blocks with bytes 0..255
2	test2.dat	884 736 000	All bytes are 0xAA
3	test3.dat	884 736 000	Random bytes in increasing range(0..255*rand())
4	test4.dat	884 736 000	Random bytes (255*rand())
5	test5.dat	442 368 000	Random bytes (255*rand())
6	test6.dat	442 368 000	Random bytes (255*rand())
7	test7.dat	117 964 800	Random bytes (255*rand())
8	test8.dat	58 982 400	Random bytes (255*rand())

Table 6.2.1: Files used in testing and their contents.

6.2.1 Results of 6800 implementation

Three texture sizes were mostly tested: 256x256, 512x512 and 1024x1024. Best results were gained with size 512x512, as 256x256 took longer, although CPU usage was lower than 512x512. 1024x1024 was slower than 512x512 (on file #1 1.029 times and on file #3 1.027 times), as seen when comparing tables A5 and A6, and a noticeable fact is that CPU usage at the same time was between 90..99% and overall response of computer was bad almost the whole time. Therefore 512x512 was considered to be the best texture size for encryption on GPU and most testing was done with that size. Trying to lower the overhead that could come from HDD I/O (bufsize twice as GPU could do in one round) showed same or worse results.

When comparing times there will be comparison between whole calculation times and between “pure computing times”. While the first represents whole running time, the second is time which does not take into account engine overhead. We get this time from full time by subtracting “Null engine” times found in table A7. It becomes important on 6800 implementation where variance among results is bigger and copying time hides real relation between CPU and GPU. Comparing full time it will give a sence of encryption time to end-user who will get results after whole process is completed, at the point of programmer is “pure computing time” comparsion more informative about CPU/GPU speed relation.

Analyzing table A5, we first notice that in case of using the 6800 chip, GPU implementation is slower than CPU implementation. On file #1 CPUs average is 46.609s and GPUs average 87.548s, it shows that GPU is 1.87 times slower than CPU. Pure computing time in the case of CPU is 11.670s and in the case of GPU is 46.393s, which means that GPU is 3.9 times slower.

With file #2 the results are 44.804s for CPU and 81.952s for GPU which means that GPU is 1.82 times slower, and by pure calculation time (CPU is 9.865s, GPU 40.797s) GPU is 4.1 times slower.

Results on encrypting files #3 and #4 as seen in table A5 are surprisingly different from previous results – GPU implementation takes 201 seconds (2.3...2.4 times slower comparing to results on files #1 and #2) to complete while CPU results are almost the same or even faster. Comparing pure computation times on file #3 (CPU 9.244s and GPU 160.301s) shows that GPU is 17.34 times slower (by full time GPU is 4.55 times slower!).

Despite of trying different configurations, results stayed the same: encrypting files #3 and #4 was much slower than files #1 and #2. As there were no chances that this difference could come due to I/O problems (encrypting same files with CPU gave same results!) the next possible explanation to this is that encryption time on 6800 depends directly on data which is currently in process – file #1 is containing blocks with bytes with ASCII code 0..255 and file #2 containing bytes with ASCII code 170 (Hex: 0xAA) while files #3 and #4 contained random data – in case #1 and #2 GPU has higher possibility to hit a value in the cache because data which is needed on next read from lookup table is near to data which was just used. But in case of files #3 and #4 GPU must read data from different area of memory during each table lookup and because GPU has relatively small cache it doesn't get that accelerating effect.

Even if encryption on GPU takes longer time, there is no good reason to take load off from CPU and put it on GPU (in case 6800 implementation) because, as seen from table B3, while encrypting files #1 and #2 on GPU, CPU is still much more utilized, and it gets even worse on files #3 and #4 where CPU usage is over two times higher. There is also a chart of CPU usage during encrypting file #3 with bigger bufsize – as seen from table B3 – CPU usage is even more higher and overall response ofPC was very jumpy at that time.

File #5, which was half the size of previous ones, gave the following results – average of CPU 20.704s and average of GPU 101.649s, real computing time of CPU 5.593s and of GPU 86.538s – shows that GPU is 15.4 times slower.

6.2.2 Results of 8800 implementation

8800 testing was done in two main stages: without X (graphic driver was not in use) and with it (KDE was running and using graphics driver). Author tried to find proof to the hypothesis that in case GPU is used only to compute AES cipher, it will be done faster, but tests showed, on the contrary, that encryption was almost always faster when X was running as seen when comparing tables A1, A2, A3 and A4 (see appendix A): with any configuration, the average results of files #1, #3 and #4 in table A4 are better than in A1, A2 and A3. Only encrypting file #2 (all bytes 0xAA) was done a little bit faster as seen in table A2. C implementation also performed better when X was running.

Comparing results of 6800 and 8800 chips, one good conclusion can be made – 8800 implementation doesn't depend on data which is encrypted as 6800 did – table A5 shows that encrypting files #3 and #4 on 6800 takes 201..202 seconds while files #1 and #2 took 81..87 seconds while table A4 shows that files #1, #2, #3 and #4 are all encrypted in 39..41 seconds.

And finally, as tables A1, A2, A3 and A4 show when encrypting large files #1..#4, 8800 implementation is generally fasterthan C implementation:

File #	CPU average	GPU average	GPU faster
1	43.208s	41.949s	1.0300
2	40.647s	40.474s	1.0043
3	42.383s	41.824s	1.0134
4	41.786s	40.314s	1.0365

Table 6.2.2: General comparision of CPU and GPU implementations on 8800.

When comparing CPU usage during encryptions on 8800, seen in table B1 and B2 (see appendix B), conclusion is that CPU usage in case of CPU and GPU encryptions is the same, or just a little bit better in case of GPU (table B1, bufsize 29 491 200, and table B2, there are bigger CPU usage drops on GPU usage). Still, question remains – what is the CPU doing while encryption process is running on GPU? Testing showed that at least type casting in GPU programs is thrown back to CPU.

6.3 Thoughts for the future

According to figure 2.2.1 and considering the results of testing 8800 implementation it becomes clear that work on studying GPUs must continue. Current work is done at the very right time – CUDA, thanks to which this kind work has been done at all, came out in March 2007 and has proven that it makes programming on GPUs and using them in non-graphic applications easier and faster. But, as it has been public for such a short time, and is still beta, author hopes that those implementations will also be tested with the next versions of CUDA and newer GPUs, as 8800 was used for programming only for a week.

There were ideas not put into practice due to lack of time, need of additional testing:

- both implementations should have dynamic texture/block/thread resizing according to the number of bytes going to be encrypted (nbytes) in function `gpu_aes_cipher` as there is always one 16 byte block at the end of AES cipher. It will also give better results on smaller files with sizes smaller than hardcoded *`ENC_BLOCK_SIZE`*.
- 6800 chip is significantly slower compared to the CPU, but there might still be some ways to make the code run faster on GPU as this was the first time for the author to write programs for GPU.
- For 8800 "pure table lookup version" (AES is reduced to four table lookups and four XORs) [2] should also be implemented which would probably be faster than the current implementation.
- 8800 code should be tested more with different configurations to find faster ways to run. Some more profiling is also needed to find slower places in code. Although it has been optimized in quite a hard way (look for example `mix_column` function which is faster than Wikipedias C implementation), there may still be lines of code which can be made faster considering the fact that it runs on GPU.
- As all this code is tested only on Linux and PC architecture, it would be appropriate to test it on other platforms as well.
- As there is now a mostly working engine for OpenSSL, it is now easy to add new ciphers. As AES doesn't need very much computing power, there might be other algorithms which could perform (asymmetric cryptography for example) better on GPU.
- Running two NVIDIA SLI-Ready cards in a single system (more detailed description available at http://www.slizone.com/page/slizone_learn.html) can give up to double better graphics performance – should be tested, does it give better results on

cryptographic algorithms too.

- Although this paper is focused on real-life use there are signs showing, that at least on case CPU implementation there is bottleneck linked to HDD I/O and avoiding that could make CPU implementation more faster when compared to 6800 implementation.

There has also been an idea that perhaps in parallel computing course, where the whole classroom is full of computers connected to each other supporting students' practical work while creating parallel programs, GPUs could be considered as good alternatives – they are easier to install, easier to maintain and communication between processes is many times faster compared to parallel computing with fortran or grid.

7 Conclusions

Goal of this thesis was to study possibilities of using GPU in non-graphics calculations, like cryptography. Author tried to figure out problems rising when moving arithmetic calculations from CPU to GPU and to determine when this move is reasonable. AES block cipher was chosen because it is common and popular and is suitable for file encryption which was the main target (leaving aside possibilities of asymmetric and stream encryption).

Author created a new engine for OpenSSL cryptographic framework and implemented AES encryption algorithm on two different chips – NVIDIA 6800GT and 8800GTS using different graphics toolkits:

- 6800 implementation was coded as a graphics application using OpenGL and data was handled as textures
- 8800 implementation was coded with a brand new tool – NVIDIAs CUDA which gave fast (no need for extra and difficult GPU initialization) and common (programming with CUDA means to write C-like code) way to use GPUs resources.

Test results show that older GPUs (6800) are not suitable to take over CPUs tasks, at least when considering AES block cipher which has not very high arithmetic intensity. But one of the fastest chips at the moment on the market, NVIDIAs 8800, is quite considerable on overtaking CPUs tasks. But when calculations are moved from CPU to GPU it must be noticed, that probably all algorithms must be remade because optimizations which give better results on CPU may on the GPU slow things down.

8 Resümee

Käesoleva magistritöö peamiseks eesmärgiks oli uurida graafikaprotsessorite (GPU) kasutamise võimalusi mittegraafiliste arvutuste jaoks, nagu näiteks krüptograafia. Autor uuris probleeme, mis võivad tekkida arvutuste üleviimisel põhiprotsessorilt GPU'le ning samuti seda, millal selline üleviimine võiks osutuda põhjendatuks. Töö aluseks sai valitud populaarne ja levinud AES plokkšiffer, mis sobib hästi failide krüptimiseks, mis oli ka põhiliseks uurimissuunaks. Voo krüptimine ja asümmeetriline krüptograafia said teadlikult kõrvale jäetud.

Autor realiseeris antud magistritöö raames OpenSSL krüptoraamistikule uue krüptomootori (engine) ning realiseeris kahe erineva kiibi – NVIDIA 6800GT ja 8800GTS – jaoks AES algoritmi, kasutades selleks erinevaid vahendeid:

- 6800 implementatsioon on kirjutatud kui graafikaprogramm, kasutades OpenGL ning kus kõiki andmeid on käsitletud tekstuuridena
- 8800 kirjutamise jaoks on kasutatud täiesti uut vahendit – CUDA, mille NVIDIA lasi välja 2007. aasta märtsis, ning mis andis kiire (erinevalt OpenGL'ist, ei ole CUDA puhul vaja näha lisavaeva GPU initsialiseerimisega jne) ja üldise võimaluse (CUDA abil programmeerimine tähendab praktiliselt C keelse koodi kirjutamist) kasutada GPU resursse.

Testimise tulemused näitavad, et vanemad GPUd (6800) ei sobi CPU ülesandeid täitma, vähemalt mitte AES plokkšifri puhul kus arvutuste osakaal on väiksem kui mäluoperatsioonide osakaal. Kuid hetkel üks võimsamaid kiipe, 8800GTS, on põhiprotsessori (CPU) abistamisel ning arvutuste ülevõtmisel täiesti arvestatav. GPU kasutamisel aga tuleb arvesse võtta asjaolu, et osaliselt vajavad algoritmid ümbertegemist kuna mitmed optimiseerimised, mis töötavad CPU peal, võivad GPU puhul hoopiski tulemuse aeglasemaks muuta.

9 Bibliography

1. Flynn's Taxonomy. http://en.wikipedia.org/wiki/Flynn's_Taxonomy, March, 2007
2. Cook, D., Baratto, R., Keromytis, A., Luck, J. (2005) Secret Key Cryptography Using Graphics Cards. http://www.cs.columbia.edu/techreports/cucs-002-04.pdf, March, 2007
3. AES Block Cipher Encryption Implementation and Analysis on Commodity Graphics Processing Units. https://www.cs.tcd.ie/~harrisoo/publications/AES_On_GPU.pdf, April, 2007
4. Cook, D., Baratto, R., Keromytis, A. (2004) Remotely Keyed Cryptographics, Secure Remote Display Access Using (Mostly) Untrusted Hardware – Extended version. http://www1.cs.columbia.edu/~library/TR-repository/reports/reports-2004/cucs-050-04.pdf, April, 2007
5. Graphics processing unit. http://en.wikipedia.org/wiki/Graphics_processing_unit, March, 2007
6. Preview: NVIDIA GeForce 6800 Ultra, (2004), http://www.behardware.com/articles/491-1/preview-nvidia-geforce-6800-ultra.html, April, 2007
7. A Personal History of 3D Graphics, (2006), http://www.extremetech.com/article2/0,1697,1911275,00.asp, April, 2007
8. NVIDIA CUDA Homepage. (2007). http://developer.nvidia.com/object/cuda.html, March, 2007
9. General-Purpose Computation On GPUS:A Primer, http://http.download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch29.pdf, April, 2007
10. NVIDIA CUDA Compute Unified Device Architecture.(2007). http://developer.download.nvidia.com/compute/cuda/0_8/NVIDIA_CUDA_Programming_Guide_0.8.pdf, March, 2007
11. 8800GTS. http://www.club3d.nl/index.php/download/pdf/CGNX_GTS8820.pdf, March, 2007
12. GPU Gems 2, The GeForce 6 Series GPU Archidecture. (2005). http://http.download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf, April, 2007
13. OpenGL. http://en.wikipedia.org/wiki/Opengl, March, 2007
14. Using OpenGL Extensions. http://www.mesa3d.org/brianp/sig97/exten.htm, April,

2007

15. OpenGL Utility Toolkit. http://en.wikipedia.org/wiki/OpenGL_Utility_Toolkit, March, 2007
16. OpenGL Extension Wrangler Library. http://en.wikipedia.org/wiki/OpenGL_Extension_Wrangler_Library, March, 2007
17. Appendix A: Cg Language Specification. http://http.download.nvidia.com/developer/cg/Cg_Specification.pdf,April, 2007
18. The Cg Tutorial: The Definite Guide to Programmable Real-Time Graphics. http://http.download.nvidia.com/developer/cg/Cg_Tutorial/Chapter_1.pdf, April, 2007
19. Cg Toolkit 1.5. http://developer.nvidia.com/object/cg_toolkit.html, March, 2007
20. Advanced Encryption Standard, http://en.wikipedia.org/wiki/Advanced_Encryption_Standard, March, 2007
21. Rijndael cipher. http://www.conxx.net/rijndael_anim_conxx.html, March, 2007
22. OpenSSL, http://www.openssl.org/, April, 2007
23. OpenSSL, http://en.wikipedia.org/wiki/OpenSSL, April, 2007
24. GPU Cluster for High Performance Computing, (2004), http://portal.acm.org/citation.cfm?id=1048933.1049991&coll=&dl=ACM&type=series&idx=1048933&part=Proceedings&WantType=Proceedings&title=Conference%20on%20High%20Performance%20Networking%20and%20Computing&CFID=15151515&CFTOKEN=6184618,April, 2007
25. GeForce 8800: the DirectX 10 era begins, http://www.firingsquad.com/hardware/nvidia_geforce_8800_preview/default.asp, April, 2007
26. AMD considers Clearspeed math co-processor, (2006) http://arstechnica.com/news.ars/post/20060315-6392.html, April, 2007
27. Nvidia Banks on Split-Personality Chip, http://www.thestreet.com/_dm/smallbusinesstech/smallbusinesstech/10347867.html , April, 2007
28. Nvidia's GeForce 8800 graphics processor,(2006), http://techreport.com/reviews/2006q4/geforce-8800/index.x?pg=1, April, 2007

Appendix A: Test results, Speed

File #	Test #	CPU	GPU 8800
1	1	47.856s	42.238s
	2	43.887s	42.815s
	3	45.957s	43.935s
	average	45.900s	**42.996s**
2	1	40.158s	40.750s
	2	41.328s	40.965s
	3	43.493s	41.920s
	average	41.659s	**41.211s**
3	1	41.802s	40.620s
	2	43.242s	39.443s
	3	41.883s	41.030s
	average	42.309s	**40.364s**
4	1	42.365s	43.539s
	2	43.684s	43.482s
	3	42.750s	42.655s
	average	**42.966s**	43.225s

Table A1: Results with configuration: GPU program – Blocks: 360, Threads: 512, Rounds in thread: 10; bufsize: 29491200 (One read per GPU round); X is not running

File #	Test #	CPU	GPU 8800
1	1	42.915s	42.405s
	2	42.663s	42.600s
	3	43.619s	43.935s
	average	43.065s	**42.980s**
2	1	36.840s	39.691s
	2	39.915s	40.368s
	3	37.776s	38.306s
	average	**38.177s**	39.455s
3	1	42.080s	37.892s
	2	38.806s	37.607s
	3	40.251s	39.958s
	average	40.379s	**38.485s**
4	1	42.071s	40.301s
	2	39.994s	42.218s
	3	40.836s	40.377s
	average	40.967s	**40.965**

Table A2: Results with configuration: GPU program – Blocks: 360, Threads: 512, Rounds in thread: 10; bufsize: 58982400 (One read per two GPU rounds); X is not running

File #	**Test #**	**CPU**	**GPU 8800**
1	1	43.082s	43.628s
	2	43.988s	42.469s
	3	43.863s	42.038s
	average	43.644s	**42.711s**
2	1	42.269s	40.133s
	2	40.310s	42.327s
	3	41.558s	40.243s
	average	41.379s	**40.901s**
3	1	40.501s	41.577s
	2	42.538s	42.327s
	3	40.299s	40.913s
	average	**41.112s**	41.605s
4	1	42.583s	42.643s
	2	44.158s	41.263s
	3	43.450s	42.332s
	average	43.397s	**42.079s**
5	1	9.918s	21.718s
	2	20.248s	19.860s
	3	12.297s	13.307s
6	1	21.995s	16.764s
	2	19.970s	16.847s
	3	7.784s	8.984s

Table A3: Results with configuration: GPU program – Blocks: 240, Threads: 256, Rounds in thread: 10; bufsize: 9830400 (One read per GPU round); X is not running

File #	Test #	CPU	GPU 8800
1	1	43.319s	41.143s
	2	43.871s	41.507s
	3	42.291s	42.868s
	average	43.160s	**41.839s**
2	1	38.865s	39.670s
	2	39.358s	40.648s
	3	42.673s	38.681s
	average	40.298s	**39.666s**
3	1	41.431s	39.138s
	2	41.910s	38.153s
	3	42.371s	40.415s
	average	41.904s	**39.235s**
4	1	41.168s	39.820s
	2	43.584s	41.788s
	3	40.605s	39.946s
	average	41.785s	**40.518s**
5	1	20.457s	21.369s
	2	18.095s	20.190s
	3	19.284s	20.289s
6	1	17.945s	20.090s
	2	19.747s	17.688s
	3	19.549s	20.441s
7	1	1.967s	2.469s
	2	1.931s	2.306s
	3	1.894s	2.303s
8	1	1.068s	1.557s
	2	1.098s	1.922s
	3	1.076s	2.684s

Table A4: Results with configuration: GPU program – Blocks: 360, Threads: 512, Rounds in thread: 10; bufsize: 58982400 (One read per two GPU rounds); X is running

File #	Test #	CPU	GPU 6800
1	1	46.197s	86.895s
	2	46.895s	86.970s
	3	46.736s	88.780s
	average	46.609s	87.548s
2	1	45.512s	84.737s
	2	42.789s	80.460s
	3	46.112s	80.659s
	average	44.804s	81.952s
3	1	45.006s	201.782s
	2	43.940s	200.754s
	3	43.604s	201.832s
	average	44.183s	201.456s
4	1	44.859s	202.042s
	2	45.938s	202.385s
	3	45.561s	202.922s
	average	45.452s	202.449s
5	1	21.061s	102.798s
	2	20.883s	101.390s
	3	20.170s	100.759s
6	1	21.547s	101.819s
	2	19.722s	103.049s
	3	20.162s	101.380s
7	1	2.009s	26.282s
	2	1.856s	26.498s
	3	1.905s	26.462s

Table A5: Results with configuration: GPU program – texture size: 512 x 512; bufsize: 4194304 (One read per GPU round)

File #	Test #	CPU	GPU 6800
1	1	44.231s	88.954s
	2	43.415s	90.592s
	3	43.821s	90.857s
3	1	40.993s	206.346s
	2	41.173s	207.879s
	3	40.071s	206.622s

Table A6: Results with configuration: GPU program – texture size: 1024 x 1024; bufsize: 16777216 (One read per GPU round). Note: CPU highly utilized.

File #	Test #	Null engine	Without GPU “run” 6800	Without GPU “run” 8800
1,2,3,4	1	34.073s	40.500s	34.257s
	2	34.855s	41.775s	35.831s
	3	35.891s	41.190s	36.244s
	average	34.939s	41.155s	35.444s
5	1	14.920s	20.952s	16.849s
	2	15.495s	18.620s	14.653s
	3	14.918s	18.450s	16.281s
7	1	3.491s	6.022s	4.367s
	2	0.697s	1.672s	4.411s
	3	0.697s	1.499s	3.478s
8	1	1.635s	2.140s	2.609s
	2	0.602s	1.174s	3.297s
	3	1.141s	1.174s	1.709s

Table A7: Results to engine speed tests. “Null engine” is result of empty function named `gpu_aes_cipher` (pure OpenSSL running time) and “Without GPU “run”” is `gpu_aes_cipher` where data coping to/from GPU is done, but real GPU-side program is not started (to measure how much time it takes to copy data between CPU and GPU).

Appendix B: Test results, CPU usage

Graphs are made with program GkrellM (available at http://www.gkrellm.net/). Each horizontal line on picture is at 25% of CPU usage, full scale is 100%.

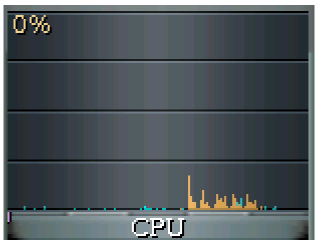


Figure B1: CPU usage while coping file #1 on HDD to new location.

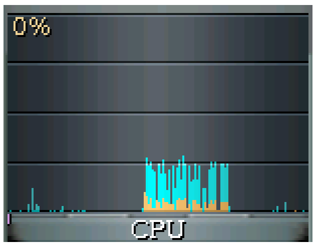
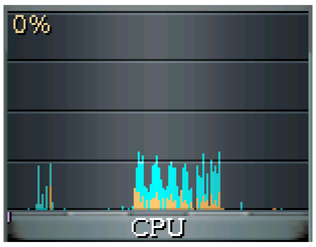
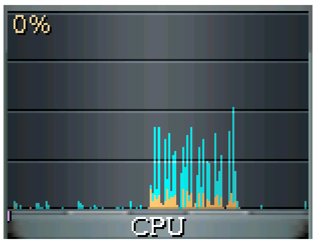
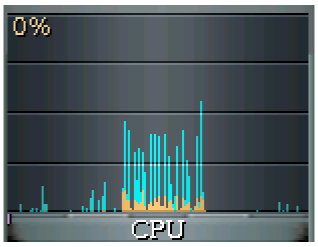
bufsize	CPU	GPU 8800
29 491 200		
58 982 400		

Table B1: CPU usage during encryption of file #1 with `ENC_BLOCK_SIZE` 29491200 (GPU configuration: 360 blocks, 512 threads, 10 rounds)

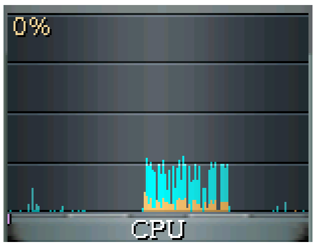
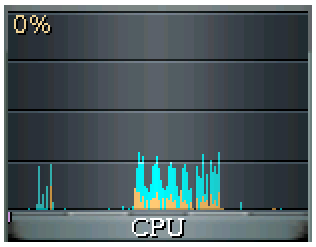
bufsize	CPU	GPU 8800
9 830 400		

Table B2: CPU usage during encryption of file #1 with `ENC_BLOCK_SIZE` 9830400 (GPU configuration: 240 blocks, 256 threads, 10 rounds)

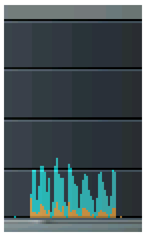
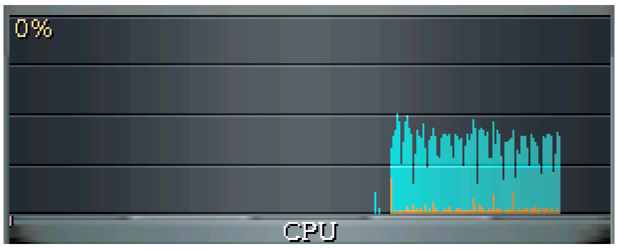
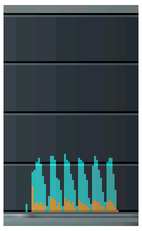
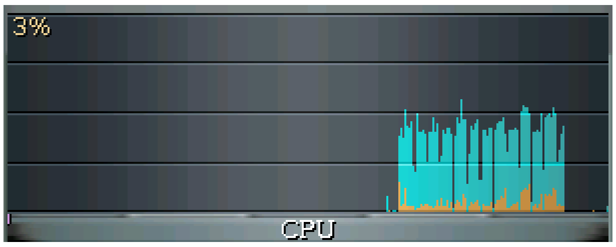
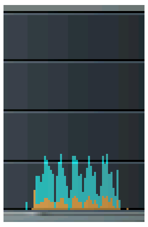
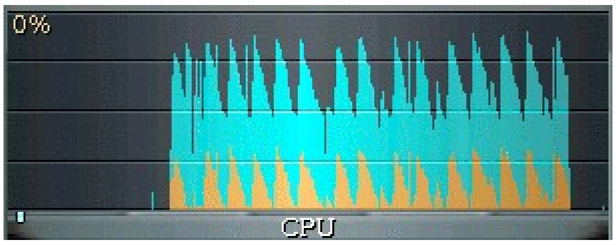
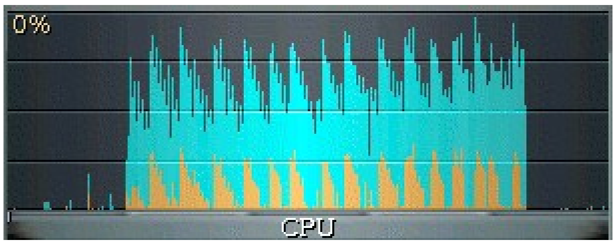
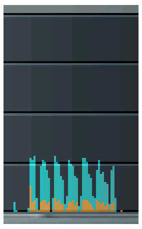
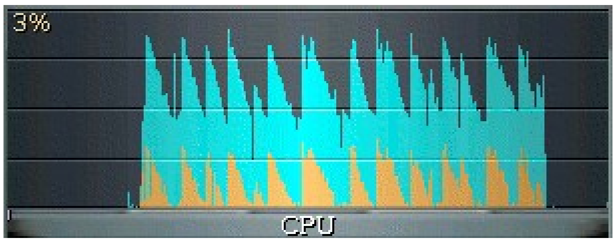
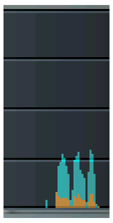
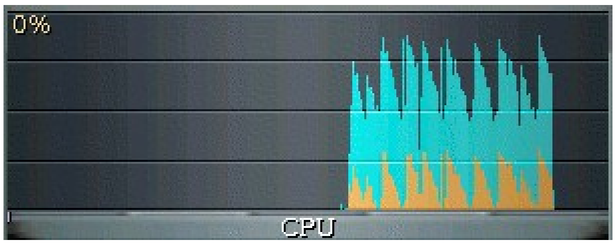
File #	CPU	GPU 6800
1	 0m42.767s	 1m25.006s
2	 0m41.492s	 1m22.954s
3	 0m41.584s	 3m19.159s  3m19.615s (bufsize 8388608)
4	 0m42.534s	 3m21.873s
5	 0m19.523s	 1m42.262s

Table B3: CPU usage during encryption of file #1 with `ENC_BLOCK_SIZE` 4194304 (Texture size 512 x 512) and `bufsize` 4 194 304

Appendix C: Figures

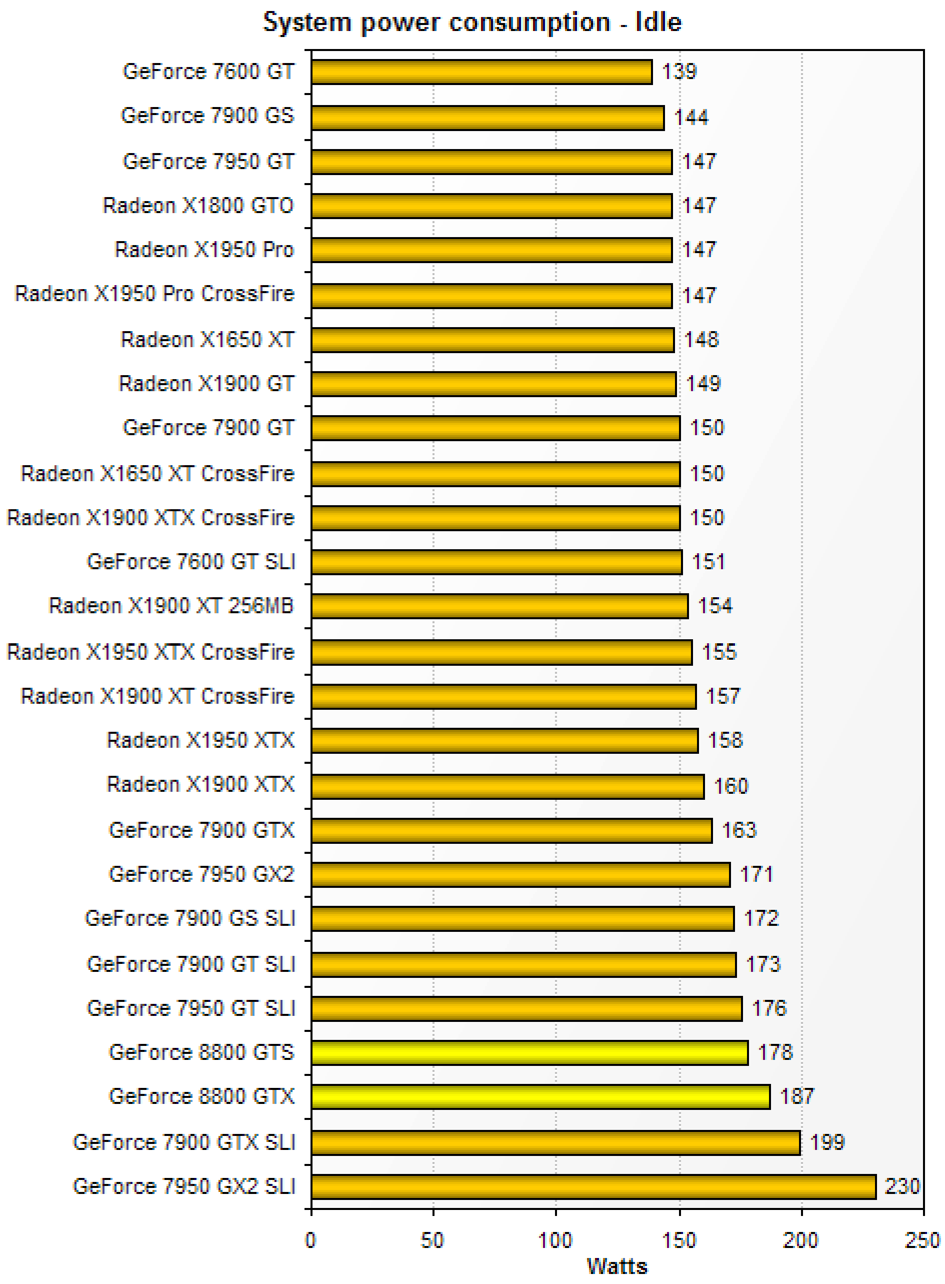


Figure C1: System power consumption when idle [28:16]

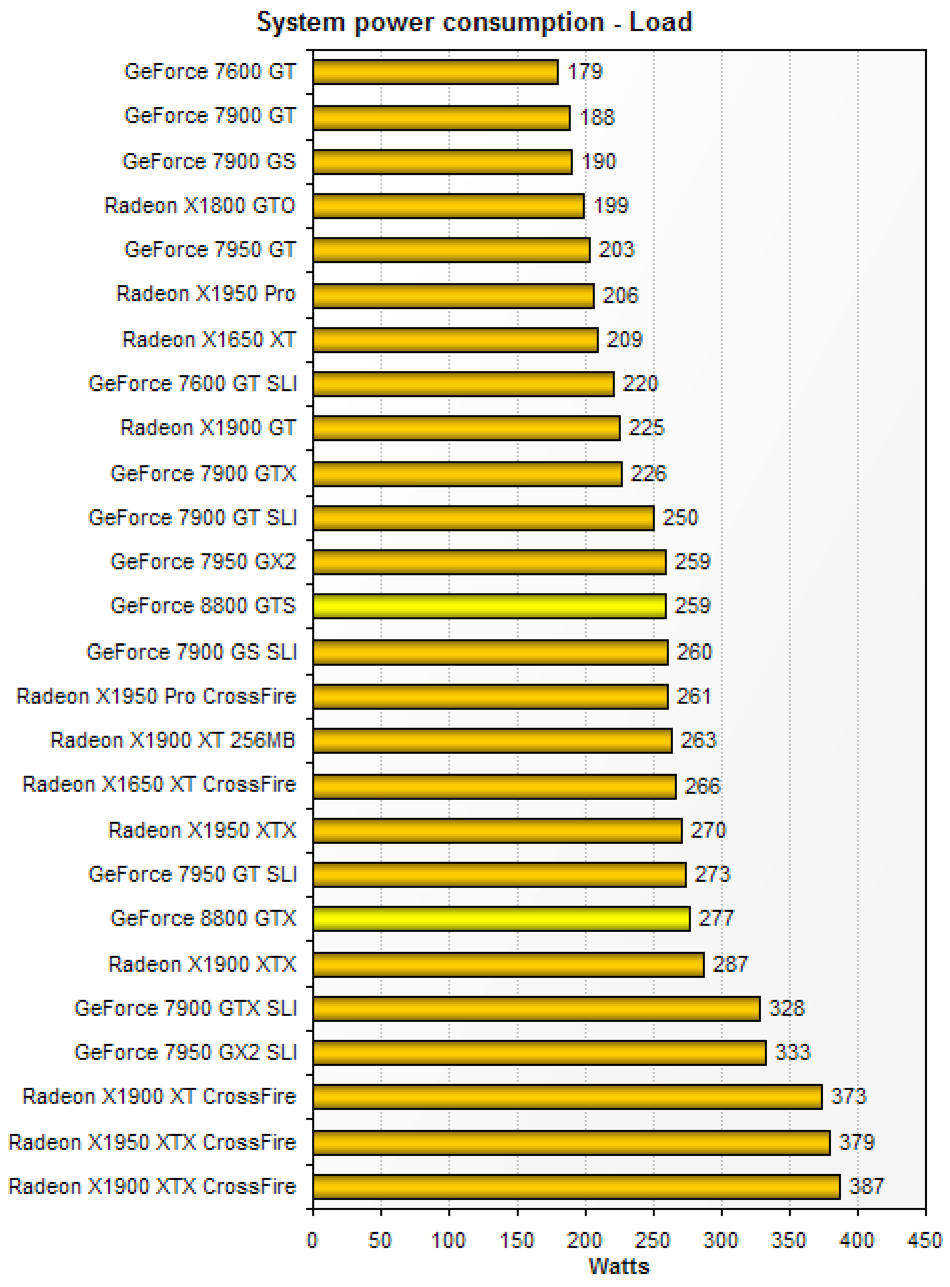


Figure C2: System power consumption when system is utilized [28:16]