OpenSSL acceleration using Graphics Processing Units

Pedro Miguel Costa Saraiva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Doctor José Carlos Martins Delgado
Supervisor: Doctor Ricardo Jorge Fernandes Chaves
Member of the Committee: Doctor José Carlos Campos Costa

October 2013

Abstract

Cryptography: The study of techniques focused on security. Typically, an implementation of cryptography is computationally heavy, leading to performance issues on general purpose systems. Adding the possibility of offloading cryptographic operations to a Graphics Processing Unit (GPU) onto a widespread, open-source cryptographic library such as OpenSSL would be extremely useful in lightening the CPU load for application logic. GPUs, while originally designed to accelerate graphics processing, have been recently gained usage for unrelated, general purpose computing, due to their massive parallel computing power. As such, two main frameworks designed to take advantage of a GPU for general purpose computing have been developed in the last few years: NVIDIA's proprietary CUDA and the Khronos Group's open standard OpenCL. In this paper we present high-performance acceleration of the OpenSSL library using both OpenCL and CUDA, specifically for the RSA and AES algorithms. Our evaluation shows that AES decryption can be over forty times faster than the standard CPU implementation, and that RSA keys can be generated over ten times faster than on a CPU. We also study the possibilities of CBC encryption and RSA ciphering, and and conclude why those algorithms are unfeasible to run on a GPU from within OpenSSL.

Resumo

Criptografia: O estudo de técnicas de segurança. Tipicamente, uma implementação criptográfica é computacionalmente pesada, o que leva a problemas de performance. Seria útil implementar a possibilidade de fazer o offloading de operações criptográficas para uma placa gráfica (GPU - Graphics Processing Unit) numa biblioteca comum, espalhada e de código livre como o OpenSSL de forma a poder libertar o uso do CPU para aplicações. Apesar das placas gráficas terem sido desenhadas com o intuito de processar gráficos, recentemente têm ganho uso para computação geral, devido ao seu poder paralelo. Assim sendo, duas bibliotecas desenhadas com o intuito de aproveitar o GPU para computação geral foram desenvolvidas: o CUDA da NVIDIA e o OpenCL do Khronos Group. Nesta tese, apresentamos aceleração de alta performance de certos algoritmos do OpenSSL utilizando ambos — especificamente, os algoritmos AES e o RSA. Os nossos testes demonstram que a decifra do AES pode ser até quarenta vezes mais rápida do que uma implementação normal num CPU, e que chaves RSA podem ser geradas até dez vezes mais rápidas do que num CPU. Também estudamos as possibilidades da cifra CBC e RSA, e chegamos à conclusão de que estes algoritmos não devem ser corridos no GPU através do OpenSSL.

Keywords

OpenSSL GPU OpenCL CUDA AES RSA

Contents

List of Figures

1

Introduction

Contents

Cryptography is the study of mathematical techniques focused on information security, including confidentiality, data integrity, and authentication. An implementation of cryptography is typically comprised of computationally intensive algorithms which are used by applications when encrypting, decrypting, and hashing data. Some implementations also include authentication and verification techniques.

One well-known application of cryptography is the Secure Sockets Layer (SSL) protocol. Originally developed by Netscape, SSL is a set of rules that govern authentication and encrypted communication between clients and servers. As the growth of secure website deployment rose rapidly, the SSL protocol became the de facto standard for secure electronic commerce. The SSL protocol is built into all popular web browsers. However, due to the computationally intensive nature of SSL technology, many of these sites struggle as demand rises, as large volumes of SSL traffic can impact the performance of even the most powerful general purpose web server systems. Therein, this thesis proposes adding functionality to take advantage of a GPU for cryptography onto an existing, widespread cryptographic framework (OpenSSL), leaving the CPU mostly free for other tasks without requiring the use of specialised, expensive cryptographic accelerator cards.

1.1 Motivation

Cryptographic operations are computationally intensive. Applications that perform frequent cryptographic operations, such as web servers, either typically take a large amount of CPU cycles from the system, or are deployed in systems incorporating cryptographic accelerator cards, to offload cryptographic operations and save system CPU cycles for application logic. These strategies typically result in complex deployment scenarios. The possibility of offloading cryptographic operations to a common processor present in most modern desktop computers would be extremely useful in lightening CPU load for other applications. Additionally, given the massive parallel capabilities of current GPUs, a significant performance increase can be achieved.

1.2 Objectives and Contributions

The goal of this work is to efficiently offload cryptographic operations onto a Graphics Processing Unit (GPU). The objective is to add functionality to take advantage of a GPU for cryptography onto an existing, widespread cryptographic framework (OpenSSL), leaving the CPU mostly free for other tasks without requiring the use of specialised, expensive cryptographic accelerator cards. Our focus is on algorithms with a good chance to perform well on a GPU (see section 2.2.6 for more details on this), in order to attempt a significant improvement in performance and lighten the load on the CPU.

1.3 Requirements

The requirements for this work are to achieve faster cryptographic operations from within OpenSSL using a Graphics Processing Unit. The result must be implemented in such a way that it can be effortlessly applied to any existing application that already uses OpenSSL for cryptographic operations. In addition, the result must be faster than the original OpenSSL implementation, as well as lighter on CPU load.

1.4 Document Structure

The paper is organised as follows. In section 2, we provide a background on the current state of the art of OpenSSL and GPU programming. We explain the basic functionality of OpenSSL and describe some of the algorithms it implements, and give a rundown on the history of GPU programming and existing frameworks for taking advantage of them. Section 3 describes how the work was implemented and how the GPU was taken advantage of for each individual algorithm that we implemented. In section 4 we evaluate our results, and conclude in section 5.

2

State of the art

Contents

This chapter starts by describing the OpenSSL library. Following this, a description detailing the GPU and how to write programs that take advantage of it is presented, including the two main frameworks used for this purpose: OpenCL and CUDA. Afterwards, these two programming approaches are compared, followed by an in-depth description of the internal workings of OpenCL. And finally, already existing cryptographic implementations that take advantage of the GPU are analysed.

2.1 OpenSSL

OpenSSL is a robust, commercial-grade, full-featured and open source toolkit implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, as well as a full-strength, general purpose cryptography library. It is managed by a worldwide community of volunteers. Originally based on the SSLeay library developed by Eric A. Young and Tim J. Hudson, it is licensed under an Apache-style license, meaning it is free to use for both commercial and non-commercial purposes. It is divided into libssl, a library implementing the SSL and TLS protocols, and libcrypto, a general purpose cryptography library, implementing a wide range of cryptographic algorithms used in various Internet standards, including AES (Advanced Encryption Standard), DES (Data Encryption Standard), RSA (a public-key cryptography algorithm) and many others [5].

The core library is written in the C programming language, and implements the basic cryptographic functions as well as providing various utility functions. There are also wrappers allowing the use of OpenSSL in other languages. In addition to implementing various cryptographic algorithms, OpenSSL gives users the possibility to use specialised accelerator hardware. This is done by using engines which communicate with said hardware to perform encryption and decryption functions through them. A minimum set of functions that an engine must declare include:

- Function to create a new instance of the engine.
- Function to register an engine's ID, name, initialisation and deletion functions.
- Function that returns a list of ciphers supported by the engine.

The cryptographic algorithms included in OpenSSL can generally be categorised into one of three types.

2.1.1 Symmetric-key algorithms

Symmetric-key algorithms, also known as secret key algorithms, are a class of cryptography algorithms that use the same cryptographic keys for both encryption of plaintext and decryption of ciphertext. The keys thus represent a shared secret between two or more parties that can be used to maintain a private information link.

Symmetric-key encryption provides secrecy when two parties (for example, “Alice” and “Bob”) communicate, ensuring that an attacker who intercepts a message cannot understand its contents. To set up a secure communication channel, Alice and Bob first agree on a certain key (let's call it k), which they keep secret to all but each other. Before sending a certain message to Bob, Alice encrypts m with the encryption algorithm E and key k to obtain the ciphertext c. By using the decryption algorithm and the same key k, Bob can decode the ciphertext c to obtain the original message m. These algorithms can be further sub-divided into block ciphers and stream ciphers. Whereas stream ciphers work on plaintext streams of any arbitrary length, block ciphers operate on fixed-size blocks. If the plaintext exceeds the size of a block, then it must be divided into multiple blocks (the inter-block dependency depends on the specific mode of operation). The most commonly used symmetric-key algorithms is AES.

2.1.1.A AES

The Advanced Encryption Standard (AES) is a popular symmetric block cipher algorithm in SSL. AES is a variant of the Rijndael algorithm — while Rijndael supports both block and key sizes of 128, 160, 192, 224 and 256 bits, the AES standard states that the algorithm can only accept 128-bit blocks, and a choice of three keys — 128, 192 and 256 bits. Depending on which version is used, the name of the standard is AES-128, AES-192, or AES-256, respectively.

A number of AES parametres depend on the key length. For example, if the key size used is 128, then the number of rounds is 10, whereas it's 12 for 192 and 256 bits, respectively.

AES's overall structure can be seen in figure 2.1. For both encryption and decryption, the input is a single 128-bit block which is then modified during each stage of the algorithm and copied to an output matrix. The key is then expanded into an array of key schedule words.

The four stages are as follows:

- Substitute bytes
- Shift rows
- Mix columns
- Add round key

The final round leaves out the 'Mix columns' stage. For a more detailed explanation of what each stage incurrs, see [30].

Since AES only works on 128-bit blocks, to encrypt larger messages, the message needs to be broken into blocks. For this, there are various block cipher modes, some of which will be described in detail.

Electronic Codebook (ECB)

This first mode is the simplest of all five modes. Every block of plaintext is encrypted with the

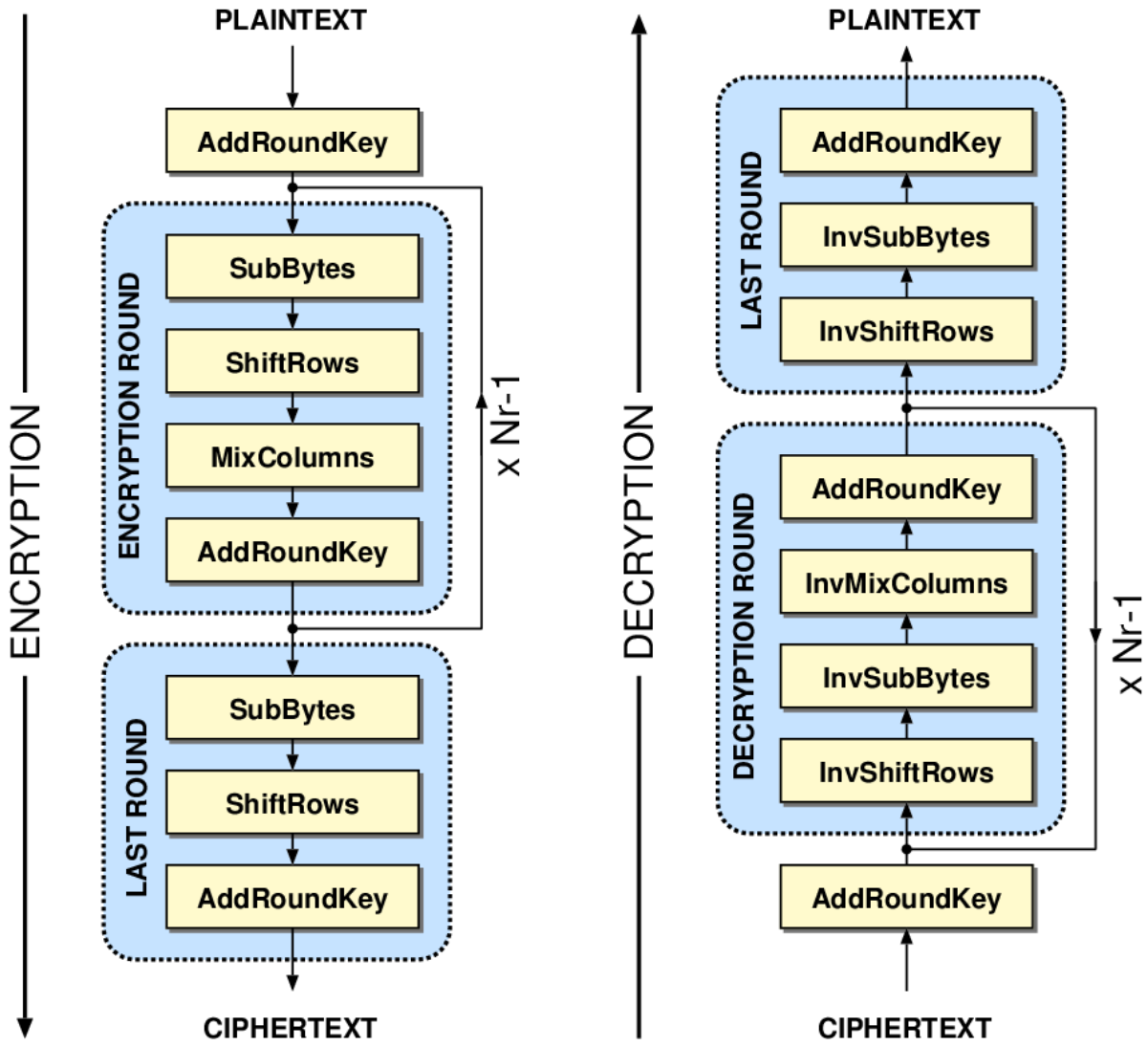

Figure 2.1: Structure of the AES algorithm

same key K. The term $codebook$ is used because, for a given key, there is only one unique ciphertext for every block of plaintext. If the message is longer than the block length, then the message is broken into blocks of the required length, adding padding if necessary. Encryption and decryption are performed one block at a time, always using the same key. The ECB method is unsafe for larger messages because if the same plaintext block appears more than once, then the same ciphertext is produced, which may assist an attacker in breaking the cipher [30].

Cipher Block Chaining (CBC)

Ideally, the same plaintext block should produce a different ciphertext block within a message. Cipher Block Chaining allows this by XORing each plaintext message with the ciphertext from the previous round prior to encryption (with the first round using an initialisation vector). As with ECB, the same key is used for each block. Obviously, the initialisation vector needs to be known by both the sender and receiver. For maximum security, it should be kept secret along with the key[30].

Counter (CTR)

This is a newer mode that was not initially listed within the standard. A counter, equal to the plaintext block size is used. The only requirement is that the counter value must be different for each plaintext block that is encrypted. Typically, the counter is initialised to some value, then incremented by 1 for each subsequent block. For encryption, the counter is encrypted and XORed with the plaintext to produce the ciphertext block. For decryption, the same sequence of counter values is used, with each encrypted counter XORed with a ciphertext block to recover the corresponding plaintext block.

2.1.2 Assymetric-key algorithms

Assymetric-key algorithms, also known as public key algorithms, are based around the concept of using two different keys for encryption and decryption. A person's key is separated into two parts: a public key (usually for encryption, but it can also be the other way around) available to everyone, and a private key that is kept secret by the owner (typically used for decryption, but again, it can also be the other way around). Let's suppose Bob has a key pair (pub-k, pri-k) and Alice wants to send Bob a message m. Like everyone else, Alice knows Bob's public key pub-k. She computes the ciphertext c with an assymetric key algorithm E and Bob's public key pub-k. By using the decryption algorithm D and his private key pri-k, Bob can decode the ciphertext and recover the original message. Of course, this is only secure if it is infeasible to compute the original message with just the ciphertext and the public key. These algorithms typically use modular arithmetic, prime numbers, factorisation and exponentials. They tend to use a small amount of memory and are very computationally demanding, being based on mathematical calculations with extremely large numbers. The most common assymetric-key algorithm in use is RSA.

2.1.2.A RSA

RSA is an asymmetric cipher algorithm widely used for signing and encryption. Each plaintext block is an integer between 0 and $n-1$ for some n, leading to a block size of $log2(n)$. Typically n ranges from 1024 to 4096 bits. **Key generation**
An RSA keypair is generated as follows:

- Pick two large prime numbers p and q. p must be different from q
- Calculate $n = p * q$
- Calculate $o(n) = (p-1)(q-1)$
- Pick e, so that $gcd(e, o(n))$ = 1, with $1 < e < o(n)$
- Calculate d, so that $d * e.mod(o(n))$ = 1 (i.e. d is the multiplicative inverse of e in modulo $o(n)$

Thus the public key PuK is composed of the large numbers e and n, and the private key PrK is composed of the large numbers d and n [31]. **RSA Cipher** To encrypt, a plaintext message is first transformed into a large integer M, then turned into ciphertext C with:

$$C = M^e \mod n \tag{2.1}$$

with a public key (n, e). Decryption with a private key (n, d) can be done with:

$$M = C^d \mod n \tag{2.2}$$

C, M, d and n are k-bit large integers, typically 1024, 2048 or even 4096 bits. Since e is chosen to be a small number (typically 3, 17 or 65537), public key encryption is 20 to 60 times faster than private key decryption. [19]

2.1.3 Cryptographic hashing algorithms

A hashing function is an algorithm that takes an arbitrary block of data and returns a fixed-size bit string, known as its hash value, in a way that any change to the data will change the hash value. The data to be encoded is typically called a message, whereas the hash value is often called a digest. A cryptographic hash function is one that (ideally) has four main properties:

- It's easy to compute the digest for a given message
- It's infeasible to generate a message that has a given digest
- It's infeasible to modify a message while preserving the digest
- It's infeasible to find two different messages with the same hash

Cryptographic hashing functions are extremely useful for, among other uses, digital signatures and ensuring authenticity. Because they depend on the entire block of data being hashed, with one change anywhere affecting the entire digest, they tend to use large amounts of memory. They are also extremely light in terms of computational usage, using additions and shifts. The most commonly used cryptographic hashing algorithms are MD5, SHA-1 and SHA-2 (also known as SHA-256 and SHA-512, depending on the size of the digest).

2.2 GPU

In the past, the most common approach to increase a CPU's processing power was to increase its clock frequency. After some time, heat dissipation became a limiting factor for this approach, leading CPU manufacturers to change their approach by adding computational cores to their CPUs instead of changing the clock frequency. Nowadays, desktop computers tend to have 2 to 8 computational cores in their CPU. The next major step to increase computational power

is likely to be heterogeneous or hybrid computing. These terms are used to refer to systems using different kinds of computational units to perform computations. These units can be CPUs, hardware accelerators or Graphics Processing Units (GPUs), among others. When one or more GPUs and CPUs are used together to perform general purpose calculations, it is called GPU computing, also commonly called General Purpose computing for GPUs (GPGPU).

GPU computing has gained momentum during the past years due to the massive parallel processing power that a single GPU contains when compared to a CPU. A high-end graphics card has roughly ten times the single precision floating point processing capability of a high end CPU at roughly the same price. A CPU's processing power increases according to Moore's law, but the increase of GPU processing power is outpacing it. Another advantage GPUs have over CPUs is their extremely low energy consumption compared to their processing power. For example, an AMD Phenom II X4 CPU operating at 2.8GHz has a ratio of 0.9 gigaflops per watt, whereas a mid-class ATI Radeon 5670 GPU has a ratio of 9.4 gigaflops per watt [1]. GPUs deploy a hardware architecture that NVIDIA calls Single Instruction Multiple Thread (SIMT). Unlike modern x86 processors' Single Instruction Multiple Data (SIMD) architecture, which focuses on data vectors and performing operations on them, the SIMT architecture focuses on the thread that executes an operation. In the SIMT architecture, groups of lightweight threads execute a set of instructions on scalar or vector datatypes. These groups of lightweight threads are called warps or wavefronts. The term warp is used in reference to CUDA and consists of 32 threads, whereas a wavefront consists of 64 threads and is used by the ATI Stream architecture. Every thread in the group has a program counter and a set of private registers allowing them to branch independently. The SIMT architecture also also allows fast thread context switching – the primary mechanism for hiding GPU DRAM memory latencies. GPUs and CPUs differ in their design due to the different requirements imposed on them. GPUs are, first and foremost, designed for graphics objects and pixel processing, which requires thousands of lightweight hardware threads, high memory bandwidth, and little control flow. CPUs, on the other hand, are mostly designed with sequential processing in mind. They have few hardware threads, large cache memories to keep memory latency to a minimum, and advanced control flow logic. New CPUs sold on today's market commonly have 4 to 12MB of cache memory in three levels, while the best graphics cards have less than 1MB of cache in two levels [14]. The cache memory and the advanced control flow logic require a lot of space on a silicon die, which in the GPU is used for additional arithmetic logic units (ALUs).

2.2.1 Programming on the GPU

2.2.1.A History

At the end of the 90s, graphics cards had fixed vertex shader and pixel shader pipelines that couldn't be programmed. This, however, changed in 2001 with the release of DirectX 8 and the OpenGL vertex shader extension. Pixel shader programming capabilities were added

a year later with DirectX 9. Programmers were now capable of writing their own vertex shader, pixel shader and geometry shader programs. Vertex shader programs mapped vertices into two-dimensional or three-dimensional space, while geometry shader programs operated on geometric objects defined by several vertices. Pixel shader programs calculate the color and shade of pixels. At the time, programmers who wanted to use the massive parallel power of their graphics cards had to express their general purpose computations in terms of textures, vertices, and shader programs. This was not particularly easy or flexible, which in 2006 lead NVIDIA and ATI to release their proprietary frameworks aimed at GPU computing. NVIDIA named their framework Compute Unified Design Architecture (CUDA), a framework which is still under active development and used in a large amount of GPU computation projects. ATI's framework was called Close-to-The-Metal (CTM) and gave low-level access to the graphics card hardware. CTM later became known as the Compute Abstract Layer (CAL). In 2007, ATI introduced a higher level C-based framework called ATI Brook+, based upon the BrookGPU framework developed at Stanford University. This framework was a layer built on top of graphics APIs such as OpenGL and DirectX, to provide the programmers with high-level access to the graphics hardware. ATI Brook+, however, used CTM to access the underlying hardware.

In 2008, both NVIDIA and ATI (which had then been bought by AMD) joined the Khronos Group in order to participate in the development of an industry standard for hybrid computing. The proposal for the free Open Computing Language (OpenCL) standard was made by Apple, and in December 2008, version 1.0 of the standard was ratified, making it the first industry standard for hybrid computing. OpenCL version 1.1 was released in August 2010, and version 1.2 was released in November 2011.

2.2.1.B Current GPUs

Modern GPUs have hundreds of processing cores that can be used for general-purpose computing. A GPU executes code in the SIMD fashion that shares the same code path working on multiple sets of data at the same time. For this reason, a GPU is ideal for parallel applications requiring high memory bandwidth to access different sets of data. The code executed on a GPU device is called a kernel. To make full use of the massive number of processing cores within a GPU, many threads are launched simultaneously and run concurrently to execute the kernel code. This means more parallelism generally produces better usage of GPU resources. GPU kernel execution takes four steps:

- The DMA controller transfers input data from host memory to device (GPU) memory
- A host program instructs the GPU to launch the kernel with a certain number of threads
- The DMA controller transfers the results back to host memory from device memory.

The fundamental difference between CPUs and GPUs comes from how transistors are composed in the processor. A GPU devotes most of its die area to a large array of Arithmetic Logic Units. In contrast, most CPU resources serve a large hierarchy of caches and a control plane for sophisticated acceleration of single threads (e.g. out of order execution, speculative loads, branch prediction)[19].

2.2.2 OpenCL

OpenCL is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other types of processors, allowing software deelopers portable and efficient access to the power of these heterogeneous processing platforms. It supports a wide range of applications, ranging from embedded and consumer software to High Performance Computing (HPC) solutions, due to a low-level portable abstraction. By creating an efficient programming interface, OpenCL forms the foundation layer for a parallel computing system of platform-independent tools. It is maintained by the non-profit technology consortium Khronos Group, and has been adopted by Intel, AMD, NVIDIA, and ARM Holdings.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors, and a cross-platform programming language. It supports data and task-based parallel programming models, and uses a subset of ISO C99 with extensions designed for parallel programming, but ommitting the use of function pointers, recursion, bit fields, variable-length arrays, and standard C99 header files.

2.2.3 CUDA

CUDA is a proprietary hardware and software architecture designed by NVIDIA for issuing and managing computations on the GPU as a data-parallel computing device without the need to map them to a graphics API. The CUDA software stack is composed of several layers: a hardware driver, an Application Programming Interface (API) with its runtime, and two high-level mathematical libraries (CUFFT and CUBLAS). The hardware was designed to support lightweight driver and runtime layers, resulting in high performance. It is programmed with 'C for CUDA' (C with NVIDIA extensions and restrictions, as well as some C++ extensions), compiled through a PathScale Open64 C compiler, to code algorithms for execution on a GPU. Third party wrappers are also available for Python, Perl, Fortran, Java, Ruby, Lua, Haskell, MATLAB, IDL, and is supported natively in Mathematica.

CUDA gives developers access to the virtual instruction set and the memory of the parallel computational elements in CUDA GPUs. With CUDA, NVIDIA GPUs can become accessible for computation like CPUs. CUDA has been used to accelerate both graphical and non-graphical applications including computational biology, cryptography, and other fields by many orders of magnitude. It provides both a low level and a higher level API. CUDA works with all NVIDIA GPUs

from the G8x series onwards (including the GeForce, Quadro and Tesla lines), and is compatible with most standard operating systems.

2.2.4 OpenCL vs CUDA

Given the two existing frameworks available for GPU computing, we must compare them to decide which one to use. The advantages and disadvantages of each are listed below:

- CUDA kernel code has full pointer support, whereas OpenCL code doesn't support pointers to functions.
- CUDA supports some C++ constructs such as templating.
- Fully GPU accelerated libraries are available for CUDA, including math libraries, random number generators, Fast Fourier Transform (FFT) libraries, and others.
- CUDA is extremely well documented, whereas OpenCL's documentation is scarce.
- CUDA performs significantly more efficiently in NVIDIA graphics cards than OpenCL.
- OpenCL doesn't have a generic address space for program-scope variables.
- OpenCL has a much wider platform support, as OpenCL supports AMD, NVIDIA and Intel GPUs equally, as well as non-GPU platforms (such as the Cell processor).
- OpenCL is an open standard, unlike CUDA which is proprietary and locked to NVIDIA.
- OpenCL is easier to integrate into existing projects, as it does not require a separate compiler.
- OpenCL can fall back to the CPU if a compatible GPU is unavailable.

The first item is not very relevant to this work, as pointers to functions aren't used often in OpenSSL in the first place. C++ constructs are useful to avoid having to write redundant code for variations of algorithms, but not essential. The built-in CUDA library for random numbers is useful, but third-party random number generators for OpenCL also exist, such as Random123[28]. However, the inefficient performance of OpenCL on NVIDIA platforms is extremely problematic. As such, to avoid this issue, we opted to create two implementations of these cryptographic algorithms: one with OpenCL, and one with CUDA.

2.2.5 How to program in OpenCL

In this section, the OpenCL architecture is presented via the abstract models defined in its specification [2]. It defines four different models: the platform model, the memory model, the execution model and the programming model.

2.2.5.A Platform model

An OpenCL platform consists of a host and a number of compute devices. An OpenCL application consists of two different parts: the host code and the kernel. The host code runs on a host device (generally the main CPU), and it manages the compute devices and the resources necessary to execute the kernels. The host itself can also function as a compute device. A function written for a compute device (i.e. a GPU, CPU or accelerator) is called a kernel. The OpenCL application must define the number and type of compute devices to be used - GPUs, for instance, are better suited for data parallelisation problems than the CPU, which is better suited for task parallelisation. Each compute device is comprised of several compute units and each OpenCL compute unit contains a collection of processing elements. The OpenCL standard has different versions, which have to be taken into consideration when writing portable code. The runtime and hardware may provide support for multiple versions of the standard, and as such the programmer has to ensure that the OpenCL platform version, compute device version and compute device language version are correct. The platform version is the version supported by the OpenCL runtime on the host system. The compute device version describes the capabilities of the hardware, and the compute device language version describes what version of the API can be used. The compute device language can never be lower than the compute device version, but it can be higher if the newer version's language features are supported on older hardware. The compute device language version cannot be queried in OpenCL version 1.0. Every version of the OpenCL API has two profiles, a full profile and an embedded one, which is a subset of the full profile intended for portable devices.

2.2.5.B Execution model

Every OpenCL application must define a context containing one or more command queues for every compute device being used [2]. A command queue is used to enqueue memory operations, kernel execution commands and synchronisation commands to a compute device. The commands in question can be performed in order or out of order. If more than one command queue is tied to a compute device, then the commands will be multiplexed between the queues. A thread executing a kernel function is called a work-item. All work-items belong in an index space called NDRange, and they are identified by their point in said space, providing a global ID for every item. Each work-item executes the same code, but the specific execution pathways through the code and the data they operate on can vary between each one. Work-items are organised into work-groups, which provide a decomposition of the index space. Work-groups are assigned unique work-group IDs with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within their work-group, which means that a work-item can be identified both by its global ID as well as a combination of its work-group ID and its local ID. Work-items in a given work-group execute concurrently on a compute unit's processing elements.

2.2.5.C Memory model

The memory model has a hierarchical structure, where memory is defined as either global, local, constant or private [2]. Global memory can be written to or read by both the host device and compute device, and memory objects can be allocated in both the host or the compute devices' memory space. On the other hand, memory objects allocated in local memory (which is typically located on-chip) are only accessible by work-items belonging to the same work-group. The private memory space consists of registers that are only available to the work-item that allocated them. Private memory is used by default for scalar objects defined within a kernel. Non-scalar objects are by default stored in global memory space. Constant memory is similar to global memory except that it cannot be written to by the compute device. Local and private memory cannot be accessed by the host device, but local memory can be statically allocated through kernel arguments. All memory used by the kernels must be explicitly managed in OpenCL. Transferring data between host and device memory can be done either explicitly, or implicitly by mapping a compute device memory buffer in the host's address-space. Memory operations can be blocking or non-blocking. OpenCL has relaxed memory consistency, meaning that it's the programmer's job to ensure that all necessary memory operations have completed in order to avoid data hazards. These can be avoided by explicitly defining synchronisation points within kernels and queues, however, no built-in mechanism exists to synchronise work-groups with each other.

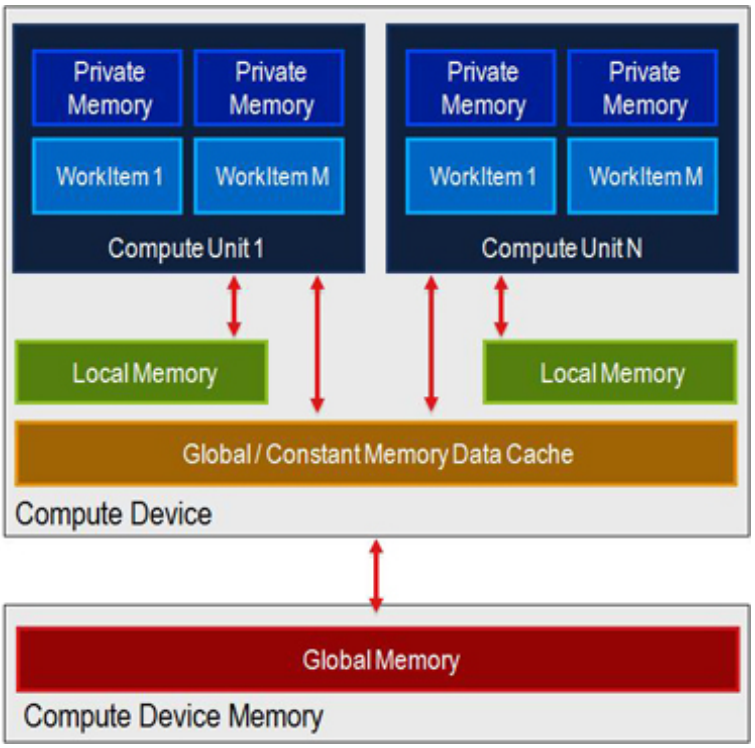

Figure 2.2: OpenCL memory model

2.2.5.D Programming model

OpenCL supports two different programming models: the data parallel model, and the task parallel model [2]. Data parallel programming uses the OpenCL index spaces to map a work-item to a set of data. It does not, however, require a strict one-to-one mapping. In task-parallel programming, the NDRange consists of a single work-item, like when executing sequential code. This model draws its benefits from being able to execute multiple kernels simultaneously, given the tasks have no inter-dependencies. This model is not recommended for use with GPUs.

2.2.6 Main challenges in GPU programming

The GPU is well-suited for solving extremely parallel problems due to its SIMT architecture. These are problems that can be divided into independent tasks that don't require interaction between each other. Data can be shared to a limited extent between GPU threads, but they should have as little dependencies to each other as possible. The most important aspect in terms of performance is high usage of the arithmetic logic units. The most significant factor impacting GPU usage is diverging execution paths, that is to say, "if-else" statements. When GPU threads diverge, the different execution paths are serialised. Therefore, an algorithm with many diverging execution paths will perform badly on a GPU. Another factor affecting the ALU usage is the access patterns to the graphics card's DRAM. A scattered data access pattern will lead to many fetch operations, decreasing the performance as the memory in the graphics cards has little to no cache. It is also beneficial if the algorithm is computationally intensive, seeing as this will hide the latency from DRAM access.

Memory in graphics cards is limited. When an application allocates all of the graphics card's DRAM, the data isn't swapped to central memory or the HDD in the same way as with regular applications – while regular applications swap memory pages, the GPU swaps entire buffers, leading to extreme performance penalties. Furthermore, when data is transferred from central memory to dedicated graphics cards, it has to go through the PCIe (Peripheral Component Interconnect Express) bus, which has a theoretical transfer rate of 8 GB/s [1]. This therefore becomes a performance-limiting factor, meaning transfers over the PCIe bus should be kept to a minimum. Therefore, it follows that I/O intensive operations are unsuited for GPU computation as they would cause intensive data transfers over the PCIe bus. Other unfavourable operations involve file and standard I/O operations, as these cannot be performed by code running on the GPU.

When developing code for both frameworks, several complications also arise deriving from differences between OpenCL and CUDA. For example, while functions in CUDA only need types passed on their arguments, in OpenCL, the address spaces need to be explicitly mentioned. This is simple enough, however, if an argument is passed with the wrong address space, instead of just giving a normal compilation error, it just crashes with a generic segmentation fault. This is made worse by the fact that debugging tools for OpenCL simply do not exist, forcing any debugging

to be performed by commenting pieces of code and adding early return statements just to find out where the application is crashing. Furthermore, while most CUDA device-side calls have OpenCL equivalents, the host-side interface is significantly different (and significantly less intuitive in OpenCL), requiring a complete rewrite of the host code.

2.2.7 Existing implementations of cryptography on the GPU

Some implementations of cryptographic algorithms on the GPU already exist. In this section, some of them shall be briefly described.

2.2.7.A OpenSSL-GPU

OpenSSL-GPU[6] was a project implementing AES-128 encryption in OpenSSL. This project was created by Urmas Rosenberg as part of his master's thesis [7], which studied the possibility of using the GPU on block ciphers. It includes two different implementations: One using a (now outdated) version of CUDA, and one written purely with OpenGL primitives. This implementation is especially notable because it was written before GPU computing became widespread. However, since it was done purely for study purposes, it only implemented AES encryption (without decryption) with 128-bit keys in ECB mode (as ECB mode, which encrypts each block independently, can be parallelised, whereas other modes where blocks depend on previous blocks' ciphers cannot). It showed comparable performance between the CPU and GPU with large buffer sizes.

2.2.7.B CryptoCL

CryptoCL[9] is a library developed by James Sweet designed to implement various cryptographic algorithms via OpenCL. At the time of this writing, only the AES algorithm in CBC mode has been implemented, and the project appears to be stalled. The host code is written in C++. Unlike the OpenSSL-GPU implementation, this library supports decryption as well as encryption, and is not limited to 128-bit keys. No results were made available for this implementation.

2.2.7.C Fast Implementation of Two Hash Algorithms on NVIDIA CUDA GPU

Right now, SHA-2 is the most trustworthy hashing algorithm being used online. However, there have been fears that the algorithm may soon be broken, which would risk the integrity of secure communications. As such, the National Institute of Standards and Technology (NIST) created the SHA-3 hash competition, to devise a new standard for secure hashing. Clearly, the most important feature in the winning algorithm will be its strength, and how difficult it is to break. However, speed and performance are also relevant factors.

As part of a thesis, Gorka Lerchundi Osa from the Norwegian University of Science and Technology implemented the Blue Midnight Wish algorithm in CUDA. At the time of its writing, the Blue Midnight Wish algorithm was a candidate in the SHA-3 hash competition. On November 2011,

however, it didn't pass to the final round of the competition [11]. Furthermore, the thesis revealed that the algorithm was actually slower when running on the GPU than it was on a normal CPU, as the communication costs between the cores far outweighed the computational costs when running it on a single CPU.

2.2.7.D OCLCrypto

The OCLCrypto library [12] is an OpenCL-accelerated cryptographic library developed by Kazuki Oikawa implementing GPU-accelerated versions of several cryptographic algorithms. As of this writing, these include standard AES, bitsliced AES, the Camellia cipher[13], 256-bit elliptic curves, as well as the hashing algorithms SHA-256 and Luffa. The host code is written in the C# language. However, this library is still not in a functional state and its development seems to have stalled.

2.2.7.E SSLShader

The SSLShader library [19] is a CUDA-accelerated cryptographic library featuring implementations of AES (128-bit keys only), RSA (with two different implementations - one based on RNS and one on Montgomery division), and SHA-1. It's written in C++. It shows a significant performance gain on 128-bit AES CBC decryption. The results presented by it are mostly focused on parallelising as many operations as possible, for use in extremely loaded servers, and achieve positive results when encrypting large sets of data (e.g. 1024 simultaneous messages for RSA).

2.3 Conclusion

OpenSSL is a robust toolkit implementing a wide amount of cryptographic operations, and it's used by an extremely large amount of applications. During the past years, GPU computing gained momentum due to its massive parallel processing power in comparison to a normal CPU, with the frameworks CUDA and OpenCL making the use of GPUs for general-purpose computing viable. A high-end graphics card could be a valid way to offset computational costs of the cryptographic operations implemented within OpenSSL and severely accelerate some of those algorithms. While there are several implementations of cryptographic algorithms available that take advantage of the GPU in some form, they are either either extremely incomplete or require applications to be programmed specifically towards it. With OpenSSL's engine module, it's possible to allow the GPU to be used from within OpenSSL itself, avoiding having to program applications to use a specific GPU-oriented library.

3

Implementation

Contents

3.1 Introduction

We start this section by explaining how an ENGINE object integrates with OpenSSL and its existing implementations. Following that, for each of the implemented algorithms, we start by briefly explaining how the algorithm can be parallelised, followed by a detailed description of how said algorithm was implemented within our work.

3.2 OpenSSL

Starting with OpenSSL 0.9.6, a component was included to support alternative cryptography implementations, most commonly used for interfacing with accelerator cards. This component is called ENGINE. It also provides dynamic binding to external engine implementations with a special engine called 'dynamic'. Using the dynamic engine, it's possible to load a separate shared library containing alternative implementations of cryptographic protocols and have OpenSSL use them instead of the default ones. This is how the GPU engine operates.

3.2.1 GPU Engine

The GPU engine implements three different operations: AES, RSA Key Generation, and RSA Cipher. A bind_helper function lets OpenSSL know what algorithms are supported by the engine.

First, the ID name and description of the engine (in this case, "gpu" and "GPU-accelerated engine") are set. Following that, the engine must inform OpenSSL of what algorithms it implements, with pointers to the functions that implement said algorithms. Finally, the engine must also inform the OpenSSL library of what cipher modes it supports. An engine object can be either built into OpenSSL or called from within it as a dynamic library — we opted for the latter, as it means users would be able to use the engine with a standard, unmodified system installation of OpenSSL. For an application to use the engine, it only needs to load the engine library and define that it should be used by default for a specific type of operation.

3.3 AES

3.3.1 Parallelism

Some cipher modes can have independent parts of the plaintext operated on independently. For those cases, there's a potential for parallelisation. The most commonly used cipher mode with AES is CBC. With it, encryption can't be parallelised because the ciphertext for a block is needed to create the ciphertext for the next block — so they can't be computed out of order. However, for decryption, things are different. Since decryption of a ciphertext block only requires the previous block's ciphertext, all blocks can be decrypted in parallel. Additionally, with the ECB cipher mode,

all blocks in the plaintext are encrypted and decrypted independently, so they can all be encrypted or decrypted in parallel.

3.3.2 Execution flow

Application-side An engine that implements AES must provide an initialisation function and a cipher function. We opted to use a modified version of the AES GPU implementation in SSLShader, as we felt it was well done. **Initialisation function**

This function receives a pointer to an OpenSSL context, a key, an IV, and a boolean informing whether this is an encryption or decryption operation. We opted to perform the key expansion on the CPU, as this operation is extremely fast, simple, and being unparallelisable, would have no discernable gain on the GPU. The context pointer is used as a key for a hashtable, on which the expanded key is stored.

Cipher function

The cipher function receives the same OpenSSL context, a pointer to the input and output buffers, and the number of bytes to proceed. Thus, the function must initialise the GPU, allocate host device memory for the input data, key and IV, and call a modified library of SSLShader's libgpucrypto. After the operation is completed, the last block of ciphertext is stored, as the function may be called again to continue the operation, should the buffer size be smaller than the complete message.

Libgpucrypto operation

When libgpucrypto is called, it must allocate memory on the GPU, transfer all the input data into it, call the GPU kernel, and wait for the result. This is done with the following steps:

- Memory on the GPU is allocated for the data (input buffer, pre-expanded key, initial IV)
- The input data is transferred to the GPU
- The GPU kernel is called with the pointer to the input, and the offset values.

Worth noting that all data is transferred into the GPU memory within a single transfer, i.e. the input buffer, pre-expanded key and IV are all transferred as a single 'blob' of data. This strongly reduces the overhead of host-to-device data transfer, as initialising a transfer of data to the GPU is a slow process.

The GPU kernel being called behaves differently depending on whether it's a CBC Encryption operation, a CBC Decryption operation, or an ECB operation.

CBC Encryption

During a CBC encryption operation, one thread on the GPU is called that goes through every block individually and encrypts it. The previous block is then used as an IV for the next block. This is all done serially (no parallelisation is possible with CBC Encryption). Control is given back to the CPU afterwards.

CBC Decryption (or ECB operation)

During a CBC decryption operation, one thread on the GPU is called for every individual block which are then processed in parallel. Once all operations are completed, control is given back to the CPU.

Once the GPU threads are complete, the output data is transferred from the GPU memory to the host. This is then returned to the OpenSSL engine, which returns the data.

3.4 RSA Key Generation

3.4.1 Parallelism

RSA Key Generation involves generating a large number of random large numbers (type BIGNUM in OpenSSL) and testing if they're prime before repeating the process again and again until a suitable prime number is found. This process can be executed in parallel — i.e. a large amount of BIGNUMs is generated at the same time and tested each in its own threads, all simultaneously.

3.4.2 Execution flow

CPU side

The RSA Key Generation code in eng_gpu.so is similar to the normal, non-GPU based code in OpenSSL, with only two major differences: At the start of the process, the gpu_genprimes is called which calls the GPU to generate a large number of large numbers (80 by default). Afterwards, in any point of the code where the standard OpenSSL BN_generate_prime_ex would be called, an alternative function (dubbed BN_generate_prime_ex_gpu for simplicity, although it doesn't do any actual generation) is called instead, which merely picks the next generated prime from the list of previously-generated primes and uses it, deleting it afterwards to ensure it won't be reused again.

GPU side

Unlike the AES and RSA ciphers, RSA key generation was not implemented in libgpucrypto and had to be instead implemented completely from scratch. Initially, when gpu_genprimes is called, a GPU random number generator is initialised with the current time in milliseconds as the seed (Random123 in OpenCL, curand in CUDA). Afterwards, device memory is allocated for the defined number of BIGNUMs to generate, followed by a call to the GPU kernel, generatePrimes_kernel. This kernel, which runs in parallel for a number of threads identical to the desired number of

BIGNUMs to generate, follows the following flow:

- The thread ID is calculated, and the output pointer is appended to the appropriate location.
- A random BIGNUM is generated.
- The BIGNUM p is tested for primality with BN_is_prime_fasttest_ex. This function, ported to the GPU from OpenSSL, performs a Miller-Rabin probabilistic primality test with $nchecks$ iterations. A number of iterations is used that yields a false positive rate of at most 2^{-80} for random input.[23]
- If the returned BIGNUM is determined to be prime, then it's written into global memory. If not, then the bytes 'NOTP' are written into global memory instead.

After all threads are done executing the kernel, the output data is copied from the GPU to the host. Notably, implementing this kernel required adapting the entirety of OpenSSL's BIGNUM library to CUDA and OpenCL code, which required retooling it to work around the limitations imposed on GPU code, particularly the inability to allocate memory from within device code.

3.5 RSA Cipher

3.5.1 Parallelism

Since the word size of most computers tends to be 32 or 64-bit, the large integers used in RSA operations must be broken down into small multiple words. Multiple threads can be executed, each of which processes a word. However, some serial processing is required to coordinate the outcome of per-word operation between threads, due to carrying bits and base extension. In addition, when using the private key where the p and q values are available, the Chinese Remainder Theorem can be used to split the operation from one k-bit modular exponentiation into two $k/2$-bit modular exponentiations, each requiring roughly eight times less computation power than the original calculation.

3.5.2 Execution flow

The standard Multi-Precision algorithm is the most convenient way to represent large integers in a computer[29]. A k-bit integer A is broken into $s = k/64$ words. In Montgomery multiplication, the multiplication of two s-word integers is performed three times. A serial multiplication algorithm has a complexity of O(s^2). This implementation is instead an O(s) parallel algorithm with linear scalability, running in s threads working in two phases. The first phase accumulates s partial products in $2s$ steps (one step for each high bit and one for one low bit), with carries being accumulated in a separate array. Each step translates into a small number of GPU instructions without involving any cascading carry propagation. The second phase repeatedly adds the carries

to the intermediate result and renews the carries, and stops when all carries become 0. The number of iterations is $s - 1$ in the worst case, but it usually only takes one or two iterations, since small carries rarely produce additional ones.

4

Results

Contents

4.1 Introduction

In this chapter, we present experimental results of our work. First we test the decryption and encryption of CBC mode, as it is currently the most commonly used cipher mode for AES. Afterwards we test RSA key generation and finally we test the RSA cipher. For each algorithm, we first present the results as a table, and afterwards we present an analysis of the results and their performance. All tests in this section were performed on an Intel Core i7 950 CPU clocked at 3.07GHz, with an NVIDIA GeForce GTX 580 GPU. AES tests were performed with PAPI (Performance Application Programming Interface), whereas RSA tests were performed with the UNIX time tool by taking ten results and averaging the result (this was necessary to ensure that the same key was used on both the GPU and CPU, and therefore have accurate results). For results with a heavy CPU load, the stress tool was executed running 300 threads, 100 looping on sqrt, 100 on malloc/free, and 100 on sync.

4.2 Results

4.2.1 AES

The most widely used cipher mode for AES is the Cipher Block Chaining mode, or CBC. This mode works by XORing every plaintext block with the previous block's ciphertext. This means that when encrypting with CBC, the blocks need to be encrypted sequentially, as encrypted block n is required before encryption of block $n-1$ can begin, and as such, CBC encryption cannot be parallelised. However, decryption is a different story — as the XOR is performed before encryption, decryption of a block can be performed in parallel. As such, CBC encryption and decryption need to be tested separately.

4.2.1.A CBC Decryption

Given how the implementation of AES CBC Decryption is parallelised at the block level, we felt it important to test how much data needs to be decrypted at once to net a performance increase, as well as how much data can be decrypted at one time before the GPU memory limit and/or thread limit is exceeded. These tests were executed using PAPI and the time results are measured in its internal unit (PAPI_TOT_INS).

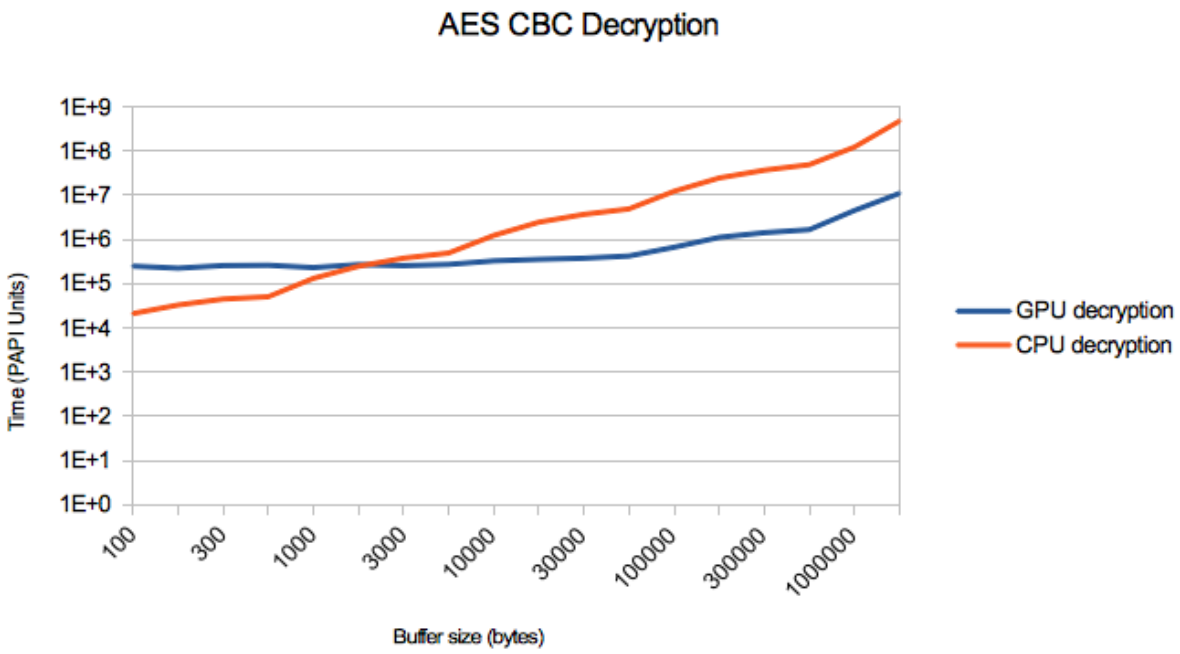

Figure 4.1: AES CBC Decryption Times

The results presented in figure 4.1 clearly show that while CPU-based decryption of AES is linear, GPU-based decryption is logarithmic. Whereas with 100 bytes (only six blocks), the GPU is around ten times slower than the CPU, when the amount of data to be decrypted reaches 3KB (18750 blocks), decryption becomes faster. With the maximum buffer size of 3.7MB (a higher size results in too many threads for this particular GPU to handle), CBC decryption on a GPU can be up to 43 times faster than on the CPU.

4.2.1.B CBC Encryption

Given that CBC Encryption cannot be parallelised at all as encrypting a block requires the ciphertext for the previous block, it's an operation that's extremely unfriendly to running on a GPU. As such, we opted to run two sets of tests — one where the operation is executed normally, and one where the operation is executed while the CPU is under heavy load. This heavy load was achieved by running the stress tool spawning 100 threads looping on sqrt, 100 threads looping on sync, and 100 threads looping on malloc and free.

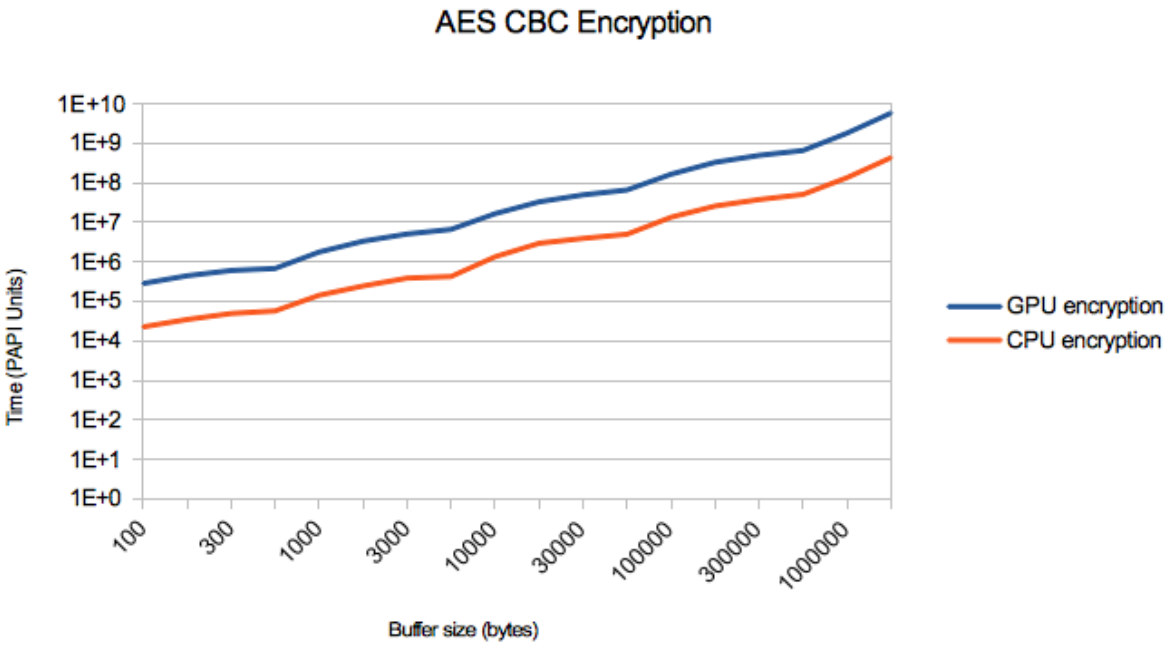

Figure 4.2: AES CBC Encryption Times

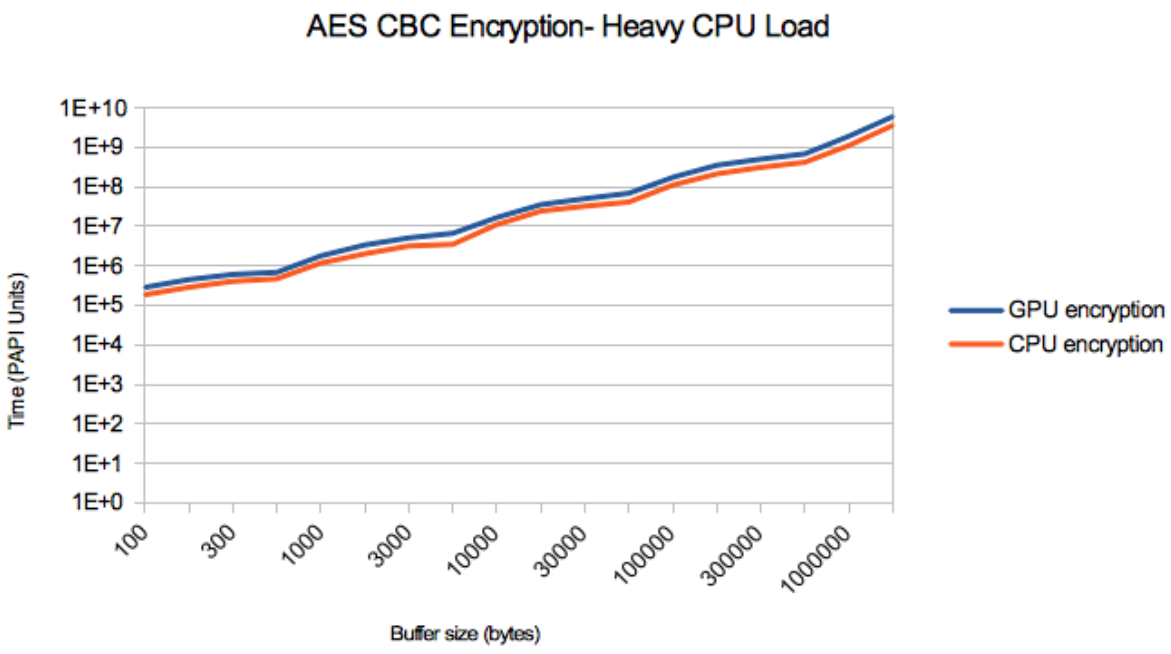

Figure 4.3: AES CBC Encryption Times - Heavy CPU Load

Unsurprisingly, figure 4.2 shows that the GPU is significantly slower at executing CBC encryption than the CPU, given its non-parallelisable nature. However, the difference between execution times on the CPU is significantly higher when under heavy CPU load, suggesting that using the GPU as a co-processor is a viable option for situations where encryption is lower priority and the

CPU is necessary for other, more prioritary operations. Figure 4.3 shows that executing CBC encryption on the GPU in the background only has a 2.7% impact on the CPU load, allowing higher priority applications to use the CPU to its fullest extent.

4.2.1.C ECB

Unlike in CBC mode, in ECB mode blocks are encrypted individually, with no inter-dependencies. This means that both encryption and decryption can be parallelised. Additionally, since this lack of inter-dependencies also means that there will be less memory access clashes.

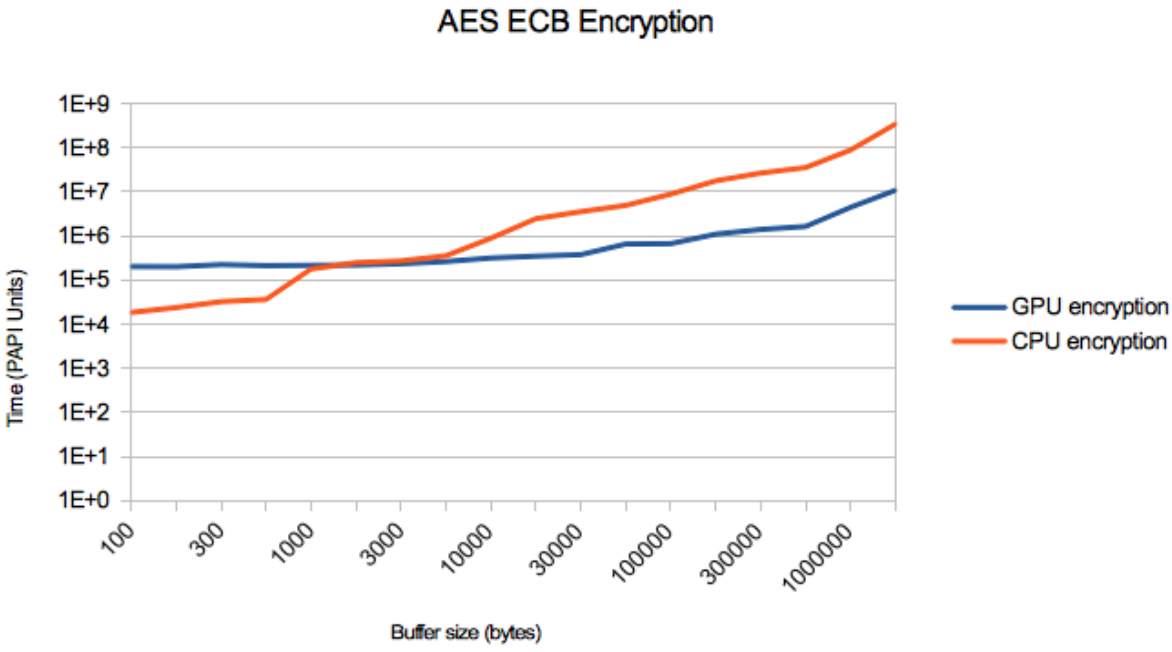

Figure 4.4: AES ECB Encryption Times

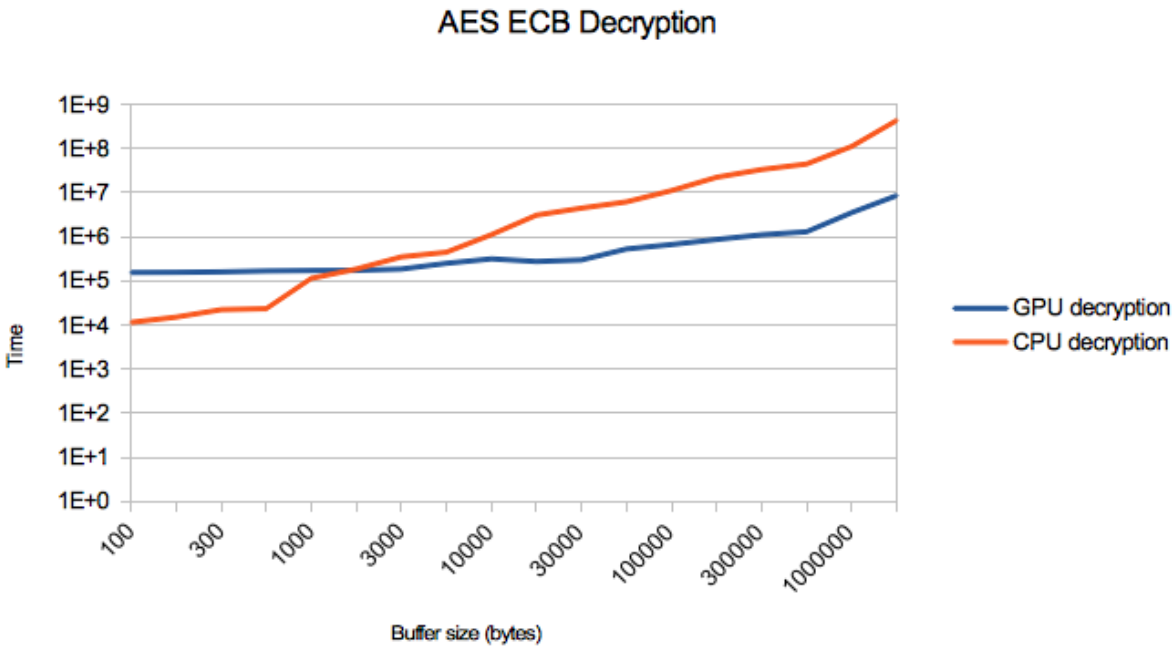

Figure 4.5: AES ECB Decryption Times

The results presented in figures 4.4 and 4.5 show that similarly to CBC decryption, ECB operations, while being slower than the CPU for very small block sizes (15 times slower for 100 bytes), gain an extremely large advantage when executed on the GPU in parallel when the buffer is 2KB or larger. Since with ECB, the blocks are encrypted (or decrypted) individually with no dependencies on other blocks, the GPU threads don't have to compete with each other for access to the blocks, resulting in an operation running in roughly 77% the time it takes to run CBC decryption. Likewise, at its peak, AES ECB encryption is 31 times faster than on the GPU than on the CPU, and ECB decryption is over 50 times faster than on the CPU.

4.2.2 RSA

4.2.2.A Key Generation

For RSA key generation, we must test the generation of keys with different bit lengths. Additionally, given how the GPU key generation code was programmed to cache generated prime numbers in order to avoid having redundant, unnecessary calls to the GPU, we felt it important to also test generating multiple keys in a row, as to show how our implementation would perform on an application that generates multiple keys on a regular basis, such as a Certificate Authority. We start with testing 1024-bit keys, followed by 2048-bit keys and finally 4096-bit keys. Since key generation is largely dependent on a random factor (how many random BIGNUMs the generator must go through before finding a prime number), we decided to run the key generation 10 times (with individual executions of OpenSSL, as to avoid prime numbers being cached before the first

execution and 'cheating' the result) and average the results. We also averaged the CPU usage of each execution, to ascertain how much the CPU might be lightened by moving this operation to the GPU.

-	GPU time	CPU time	GPU — CPU cycle time	CPU — CPU cycle time
1024 bits, 1 key	0.107s	0.038s	0.009s	0.027s
1024 bits, 2 keys	0.109s	0.084s	0.009s	0.066s
1024 bits, 5 keys	0.111s	0.181s	0.010s	0.108s
1024 bits, 10 keys	0.117s	0.360s	0.012s	0.288s
1024 bits, 100 keys	0.816s	4.347s	0.092s	2.889s
2048 bits, 1 key	0.131s	0.192s	0.009s	0.119s
2048 bits, 2 keys	0.133s	0.393s	0.009s	0.370s
2048 bits, 5 keys	0.135s	1.064s	0.010s	1.016s
2048 bits, 10 keys	0.137s	1.908s	0.011s	1.815s
2048 bits, 100 keys	0.997s	16.098s	0.154s	14.564s
4096 bits, 1 key	0.225s	2.180s	0.009s	2.078s
4096 bits, 2 keys	0.228s	4.119s	0.009s	4.085s
4096 bits, 5 keys	0.232s	10.592s	0.011s	10.516s
4096 bits, 10 keys	0.233s	18.782s	0.012s	18.648s
4096 bits, 100 keys	11.594s	2m38.819s	6.824s	2m36.880s

Table 4.1: RSA key generation times

-	GPU time	CPU time	GPU — CPU cycle time	CPU — CPU cycle time
1024 bits, 1 key	0.108s	0.065s	0.009s	0.026s
1024 bits, 2 keys	0.109s	0.159s	0.009s	0.092s
1024 bits, 5 keys	0.110s	0.206s	0.010s	0.108s
1024 bits, 10 keys	0.119s	0.539s	0.012s	0.285s
1024 bits, 100 keys	0.856s	6.145s	0.012s	2.889s

Table 4.2: RSA key generation times with a heavy CPU load

When generating 1024-bit keys, a prime number is found relatively quickly. As such, the CPU performs better than the GPU when generating a single, or even two 1024-bit keys, as seen in table 4.1. However, even when generating 1024-bit keys, the GPU gains ground as the number of generated keys is increased, due to the aforementioned prime caching system. When the number of bits in the key is increased, the more attempts have to be made to generate prime numbers. As such, even when generating a single key, the GPU is slightly faster than the CPU for 2048-bit keys, and significantly faster for 4096-bit keys. For the generation of a single 1024-bit key, usage of the CPU would be recommended, whereas for generating 2048-bit or higher keys, the GPU is a better option. Additionally, the GPU is especially useful for applications that need to generate a large number of keys 'en masse', such as Certificate Authorities. The results also show that in every circumstance, CPU usage is significantly lighter when the GPU is used than when the CPU is used, which means that even for low-bit key generation, the GPU is a worthwhile option for generating keys when the CPU is otherwise loaded, as seen in table 4.2.

4.2.2.B Cipher

Since encryption and decryption are exactly the same operation when dealing with RSA, we opted to only test encryption. As RSA is generally used to encrypt or decrypt single messages (typically a hash for a signature or a symmetric key as part of key negotiation), we encrypted a single message with a 1024-bit, 2048-bit and a 4096-bit key. Additionally, we also tested the possibility of encrypting multiple messages in parallel with a 4096-bit key, with separate calls to OpenSSL. These results are the average of 10 executions.

Key size (bits)	GPU time	CPU time
1024	1.654s	0.074s
2048	6.634s	0.448s
4096	12.2833s	2.902s

Table 4.3: RSA cipher execution - single message

Key size (bits)	GPU time	CPU time
1024	1.742s	0.142s
2048	6.682s	0.543s
4096	12.395s	3.645s

Table 4.4: RSA cipher execution - Heavy CPU Load

Number of Messages	GPU time	CPU time
1	12.283s	2.900s
5	1m02.673s	3.727s
10	2m17.780s	5.777s
15	2m36.600s	5.841s

Table 4.5: RSA cipher execution - multiple messages (4096-bit key)

The results in table 4.3 show that RSA performs significantly poorly on the GPU. When using a 1024 or 2048-bit key, using the GPU shouldn't even be considered — the performance penalty is tremendous. For 4096-bit keys, the gap is narrower, but still significant. However, the results under heavy CPU load (table 4.4) show that for 4096-bit keys, it's worth using the GPU when RSA execution is lower priority and the CPU needs to be free for other, more important tasks. Even running multiple threads does not net a gain (table 4.5), since setting up the GPU actually takes longer than a standard execution of RSA on a CPU, leading the performance to actually get worse as the number of threads is increased. While a more optimised implementation of RSA on the GPU could be created that would process multiple messages at once (by loading them all into the GPU at once and executing them within a single kernel, for instance), any code that used it would have to be tailored specifically to such an implementation, and would not work within a general purpose library such as OpenSSL, where every message is its own independent call, which in turn requires the CPU to call the GPU separately for each message. In addition, the GPU is not designed to have a large number of different host threads calling it individually (as opposed to

having multiple device threads invoked simultaneously from the same host thread), as it leads to memory fragmentation, which causes the GPU's memory to run out extremely quickly.

4.3 Conclusion

The results show a significant gain for AES ECB operations and CBC decryption, for any amount of data greater than 2KB, achieving performance improvements up to fifty times faster. RSA key generation also gains a performance improvement — a smaller improvement when generating a single key, and a large improvement when generating multiple keys. However, the RSA cipher and AES CBC encryption seem to perform worse. AES CBC encryption can still be worthwhile in situations where it needs to run in the background while more important processes need to use the CPU. A similar situation applies to RSA when using high-bit keys, though for 1024-bit keys, the execution on a CPU is so minimal that it wouldn't be of any use.

5

Conclusions

Contents

5.1 Conclusion

OpenSSL is a widely-used library used by a large number of applications around the world. Rewriting them all to use a different library in order to take advantage of the GPU would be an unfeasible, monumental task. With this work, we present an engine that allows existing applications to skip that complex stage and take advantage of the GPU to increase performance and lighten CPU load on the most commonly used encryption algorithms, implemented with both OpenCL and CUDA. Our evaluation shows that significant performance gains are obtained on RSA key generation (especially when generating a large number of keys), AES ECB operations, and AES CBC decryption, and these gains can be applied right now to existing applications with minimal effort. Unfortunately, the RSA cipher did not gain any performance, given the way OpenSSL works, the GPU has to be called individually for each operation which results in too large a performance loss. However, even in such a situation, the GPU can be used as a co-processor to lighten CPU load for other, more important tasks. The same can be applied for AES CBC Encryption, which while performing over ten times slower on the GPU, only requires 2.7% of the CPU processing power (chapter 4.2.1.B). CBC Decryption can be up to 43 times faster (chapter 4.2.1A), whereas ECB encryption can be 31 times faster, and ECB decryption can be over 50 times faster (chapter 4.2.1.C).

5.2 Future work

As stated in the previous section, given the way OpenSSL works, the GPU has to be called individually for each operation, resulting in too large a performance hit. However, it may be feasible to create a manager that caches RSA requests and sends them all at once to the GPU. Another possibility that was not implemented was AES's CTR cipher mode, which similarly to ECB, can be parallelised in both encryption and decryption — this was due to the fact that CTR mode in OpenSSL is still not considered stable (and is in fact disabled by default), however, it seems to be gaining more usage lately. We leave these issues to future work.

Bibliography

[1] ATI Stream Computing, Programming Guide
http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf
Fetched on May 15th, 2012.

[2] The OpenCL Specification, version 1.2
http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
Fetched on May 19th, 2012.

[3] About the OpenSSL Project
http://www.openssl.org/about/
Fetched on May 21st, 2012.

[4] Porting CUDA Applications to OpenCL
http://developer.amd.com/zones/OpenCLZone/programming/pages/portingcudatoopencl.aspx
Fetched on May 21st, 2012.

[5] OpenSSL
http://openssl.org
Fetched on May 22nd, 2012.

[6] OpenSSL-GPU
http://labs.sasslantis.ee/openssl-gpu/
Fetched on May 23rd, 2012.

[7] Using Graphic Processing Unit in Block Cipher Calculations
http://labs.sasslantis.ee/wp-content/uploads/2011/03/mastersthesis.pdf
Fetched on May 23rd, 2012.

[8] SHA1 Hash Algorithm OpenCL implementation
http://royger.org/opencl/?p=12
Fetched on May 23rd, 2012.

[9] Omegaice (James Sweet) GitHub
https://bitbucket.org/Omegaice/
Fetched on May 23rd, 2012

[10] Fast Implementation of Two Hash Algorithms on nVidia CUDA GPU
http://upcommons.upc.edu/pfc/bitstream/2099.1/7933/1/Masteoppgave.pdf
Fetched on May 23rd, 2012.

[11] THIRD (FINAL) ROUND CANDIDATES
http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/submissions_rnd3.html
Fetched on May 23rd, 2012.

[12] OCLCrypto
https://github.com/kazuki/oclcrypto
Fetched on May 23rd, 2012.

[13] Camellia algorithm
http://info.isl.ntt.co.jp/crypt/eng/camellia/index.html
Fetched on May 23rd, 2012.

[14] NVIDIA's Next Generation CUDA Compute Architecture
http://www.nvidia.com/content/PDF/fermi_white_papers
/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
Fetched on May 18th, 2012.

[15] OpenSSL: Documents, engine(3)
http://www.openssl.org/docs/crypto/engine.html
Fetched on June 1st, 2012.

[16] Introduction to Cryptography: Principles and Applications
By Hans Delfs and Helmut Knebl

[17] OpenSSH
http://www.openssh.org/
Fetched on June 3rd, 2012.

[18] Using the Cryptographic Accelerators in the UltraSPARC T1 and T2 processors
http://www.oracle.com/technetwork/server-storage/archive/a11-014-crypto-accelerators-439765.pdf
Fetched on June 4th, 2012.

[19] SSLShader: Cheap SSL Acceleration with Commodity Processors
http://shader.kaist.edu/sslshader/sslshader.pdf
Fetched on July 10th, 2013.

[20] Porting CUDA Applications to OpenCL
http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/porting-cuda-applications-to-opencl/
Fetched on July 12th, 2013.

[21] CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures
http://scholar.lib.vt.edu/theses/available/etd-07282011-122302/unrestricted/Martinez_GE_T_2011.pdf
Fetched on July 12th, 2013.

[22] OpenSSL: engine(3)
http://www.openssl.org/docs/crypto/engine.html
Fetched on July 14th, 2013.

[23] OpenSSL: BN_generate_prime(3) https://www.openssl.org/docs/crypto/BN_generate_prime.html
Fetched on Sep 2nd, 2013.

[24] PAPI: Performance Application Programming Interface
http://icl.cs.utk.edu/papi/
Fetched on Sep 10th, 2013.

[25] OpenCL Reference Pages
http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/
Fetched on Sep 10th, 2013.

[26] CUDA Documents
http://docs.nvidia.com/cuda
Fetched on Sep 14th, 2013.

[27] cuRAND
https://developer.nvidia.com/curand
Fetched on Sep 14th, 2013.

[28] Random123: a Library of Counter-Based Random Number Generators
http://www.thesalmons.org/john/random123/releases/1.08/docs/index.html
Fetched on Sep 14th, 2013.

[29] D. E. Knuth. The Art of Computer Programming, volume 2
Addison-Wesley, 3rd edition, 1997.

[30] The Advanced Encryption Standard (AES) http://www.facweb.iitkgp.ernet.in/ sourav/AES.pdf
Fetched on Oct 7th, 2013.

[31] The RSA Algorithm
https://tao.truststc.org/Members/yuanxue/cyptography_new/PublicFetched on Oct 7th, 2013.