Mestrado em Engenharia Informática
Dissertação/Estágio
Relatório Final

Cryptography in GPUs

Samuel Neves
sneves@student.dei.uc.pt

Advisor:
Filipe Araújo
Date: July 10, 2009

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

Cryptography, the science of writing secrets, has been used for centuries to conceal information from eavesdroppers and spies. Today, in the information age, data security and authenticity are paramount, as more services and applications start to rely on the Internet, an unsecured channel. Despite the existence of security protocols and implementations, many online services refrain to use cryptographic algorithms due to their poor performance, even when using cryptography would be a clear advantage.

Graphics processing units (GPU) have been increasingly used in the last few years for general purpose computing. We present and describe serial and parallel efficient algorithms for modular arithmetic in the GPU. Based on these, we developed GPU implementations of symmetric-key ciphers, namely AES and Salsa20, and public-key algorithms, such as RSA, Diffie-Hellman and DSA. We bundled this software into a library that contains the main achievements of this thesis.

We show that our symmetric-key cipher and modular exponentiation implementations included in this library outperform recent Intel CPUs and all previous GPU implementations. We achieve 11686 512-bit modular exponentiations per second, 1215 1024-bit modular exponentiations per second and peak AES-CTR throughputs of 1032 MB/s.

Contents

Chapter 1

Introduction

The preservation of secrets is an activity as old as their existence. Since ancient times humans have sought ways to keep secret information concealed or otherwise protected from third parties or even foes. Some of the ways involved securing the information's medium — using safes, guards or other means to protect the support where the information was stored. Another way would be to encode the information in such a way that an attacker would not be able to recover it even if he managed to get it. The latter method evolved over time to become a whole field, today known as cryptography. Whereas in ancient times cryptography only dealt with information secrecy, today it has 4 main objectives:

Confidentiality Keep information secret to anyone but the intended recipient(s).

Integrity Ensure the information has not been corrupted or tampered with.

Authentication Corroborate the information and/or its sender's origin.

Non-repudiation Prevent a party from denying previous actions or agreements.

For centuries cryptography was employed solely in diplomatic and military circles. From Caesar's cipher to the much more advanced Enigma machine used in the Second World War, cryptography was a tool used to protect little more than national secrets. A thorough insight into cryptography's uses throughout history is given in [42].

With the advent of computers and digital communications in the 1960s and 1970s, private companies started demanding methods to protect their digitally archived data. To fill this demand IBM invested in cryptographic

research and with a team led by Horst Feistel produced what was ultimately known as *Data Encryption Standard* or simply DES [54].

Meanwhile, significant breakthroughs were being made in a completely different direction. In 1976, Whitfield Diffie and Martin Hellman published a milestone paper where they introduced a radical new concept: public and private keys. The *public key* could be known by everyone, including foes; the *private key* should remain known only to its owner. With this concept, Diffie and Hellman introduced a new key exchange mechanism without any previous secret sharing between parties [30]. The security of the method relied on the hardness of solving an intractable mathematical problem[1]that would make possible to derive the private key from the public key. Two years later Rivest, Shamir and Adleman introduced the first encryption and digital signing method based on this very concept, today known as RSA. Unlike Diffie and Hellman's work, RSA relies on the hardness of factoring large integers [71]. Since then, numerous other public key algorithms have been proposed, based on various hard mathematical problems.

1.1 Motivation

With the Internet now an everyday part of commerce and communication, encryption and information protection has never been so important; thus, cryptography plays a crucial role in today's Internet infrastructure. From online shopping to banking and stock markets, secure communication channels are a requirement to many activities done remotely.

However, even though cryptography plays such an important role in the Internet, many services refrain to use it even when they would clearly benefit from it. For instance the DNS protocol, one of the most important protocols in the Internet, has suffered many attacks over the years, some of them allowing malicious attackers to forge replies and thus to direct unwitting users to fake websites. This could be fixed by using digitally signed records. In fact, a secure solution already exists: DNSSEC. However, its adoption has been strikingly slow and one of the reasons pointed out has been performance [6]. Another Internet protocol that has seen slow adoption is HTTPS. Recent study reveals that 70% of the CPU time in such a transaction is spent on cryptographic operations [82]. This performance hit is one of the leading reasons secure communications are still not ubiquitous on the Internet, besides where mandatory (e.g. financial transactions).

[1]In the Diffie-Hellman key exchange, this was the discrete logarithm over a finite field — given $g^x \pmod p$, find x.

Cryptography is also important for secure data-at-rest storage. The loss of personal and other sensitive data has become a large problem that could also be solved by using cryptography. In large companies, backup tapes are often sent to offsite facilities. Encryption would thwart the threat of data theft, but the performance hit here may also be a problem [79]. Databases often are shutdown when performing backups. Thus, the backup process must be as fast as possible to avoid unwanted downtime.

Graphics processing units, or GPUs, are special purpose processors originally designed to perform fast processing and rendering of 3D content. Their growth in processing power has been substantial in the last few years, where they now sport more than 1 TFlop/s of computing power. Furthermore, GPUs have also become more flexible, even allowing to perform more general purpose computations. Consequently, it makes sense to employ a GPU as an accelerator for cryptography given its computing power, ubiquity and relative low price.

1.2 Objectives

It becomes clear that accelerating cryptography has a crucial role in its adoption in real-world applications where security matters. This defines this project's objectives: to employ a 3D graphics card to accelerate the most common cryptographic functions, both symmetric and asymmetric. To do so, we will develop a library that uses the NVIDIA CUDA technology to implement:

- Symmetric-key primitives — AES, Salsa20.
- Asymmetric-key primitives — Diffie-Hellman, RSA, DSA.

1.3 Results

As far as we know, our work resulted in record-setting throughputs for symmetric ciphers — up to 1033 MB/s (2868 MB/s discarding memory transfers) for AES with a 128-bit key in the CTR mode, up from a previous best of 864 MB/s on an NVIDIA 8800GTX and 171 MB/s in our test CPU; the Salsa20 implementation showed throughputs of up to 1358 MB/s (8868 MB/s without memory transfers), up from 495 MB/s in the test CPU.

We also implemented public-key cryptographic primitives, based on the speed of modular exponentiation. We have achieved peaks of 11686 512-bit modular exponentiations per second, 1215 1024-bit modular exponentiations

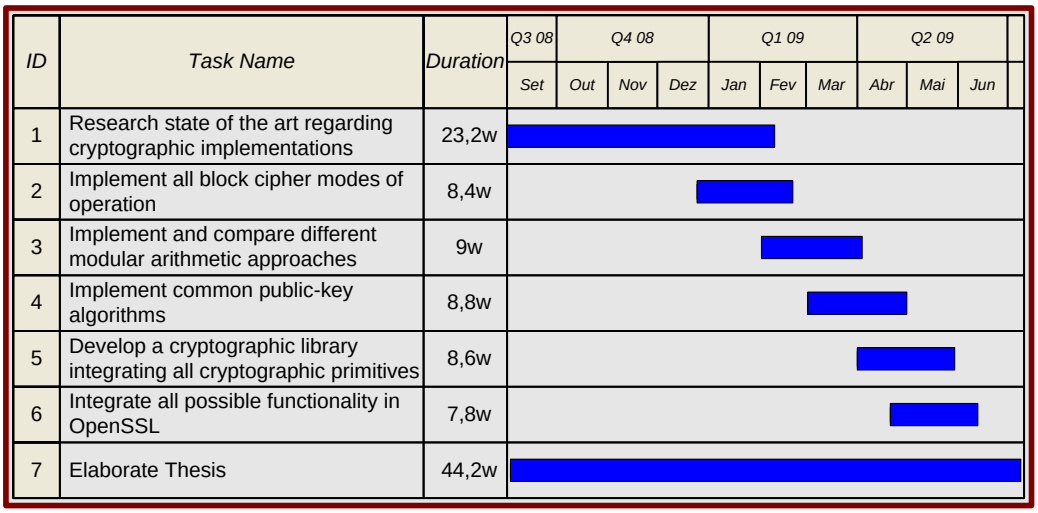

Figure 1.1: Gantt chart for the planned work throughout the year.

per second and 71 2048-bit modular exponentiations per second. This in turn allowed us to achieve over 43457 RSA-1024 encryptions per second, 6022 RSA-1024 decryptions per second, setting again speed records for RSA-1024 decryption in the GPU. We also implemented 1024 and 2048-bit Diffie-Hellman key exchanges, for which we obtained throughputs of 1215 and 71 key exchanges per second respectively. Finally, our DSA implementation achieves 5653 and 3256 signatures and verifications per second, respectively.

1.4 Work Distribution

Figure 1.1 depicts the planned work distribution throughout the duration of the project. Some of the tasks are very much related — for example, in modular arithmetic and public-key cryptography implementations improving the former directly improves and builds upon the latter. Furthermore, if the performance of a public-key algorithm is not satisfactory, one must go back to improving the modular arithmetic, the building block the public-key algorithms.

1.5 Outline

Chapter 2 deals with the mathematic and cryptographic background required to perform the work proposed in the objectives. It also covers the state of the art in algorithms and implementations in both CPUs and GPUs, starting with the AES encryption standard and the Salsa20 stream cipher. We then proceed to describe the most common public-key algorithms today, RSA, DSA and Diffie-Hellman. Finally, we describe the state of the art in multiple precision arithmetic required to implement such public-key algorithms in an efficient manner.

In Chapter 3 we devote special attention to NVIDIA GPUs and the CUDA programming model. It also contains a small introduction to CUDA programming and its API. Finally, the current NVIDIA hardware architecture, GT200, is described in detail.

Chapter 4 describes the objectives, requirements and design choices considered during the course of this work, most notably during the development of the library. It proceeds, then, to describe in detail the functionality of the library, its inputs and outputs and testing procedures performed.

Implementation details related to the low-level algorithms described in Chapter 2 will be explained in Chapter 5. These are the algorithms employ in the library of Chapter 4. Chapter 5 is of particular importance — most performance-related decisions and optimizations are described in this chapter and directly affect the final results.

The results obtained by the implementations described in Chapter 5 will be presented and discussed in Chapter 6. These results are then compared to the state of the art in both CPU and GPU implementations.

Chapter 2

State of the Art

2.1 Mathematical Background

In this section some background is given for the understanding and implementation of the public key algorithms considered.

2.1.1 Groups, Rings and Fields

Definition 1. *Let G be a non-empty set and $\circ$ a binary operation*[1] *in G. The pair $(G, \circ)$ is a* group *if and only if:*

- *$\circ$ is associative, i.e. $\forall a, b, c \in G, a \circ (b \circ c) = (a \circ b) \circ c$.*
- *G has an identify element $e \in G$, such that $\forall a \in G, a \circ e = e \circ a = a$.*
- *For each $g \in G$ there's an inverse element $g^{-1} \in G$ such that $g \circ g^{-1} = e$.*

A group is said to be Abelian or commutative *if $\forall a, b \in G, a \circ b = b \circ a$. The* order *of the group is the number of elements it contains; if this number is finite, the group is said to be* finite *and its order is denoted by $|G|$.*

Definition 2. *The triplet $(G, +, \cdot)$ is a* ring *if and only if:*

- *(G, +) is an abelian group.*
- *$\cdot$ is associative.*
- *The distributive law holds, i.e. $\forall a, b, c \in G, a \cdot (b + c) = a \cdot b + a \cdot c$.*

If $\cdot$ is comutative, $(G, +, \cdot)$ is said to be a commutative ring. *$a \in G$ is* invertible *if there is an inverse element of a in G relative to $\cdot$.*

[1] That is, has 2 distinct operands as input

Definition 3. *The triplet* $(G, +, \cdot)$ *is a* field *if:*

- $(G, +, \cdot)$ *is a commutative ring.*
- *Any element* $g \in G, g \neq 0$ *is invertible.*

*The*order *of an element* $g \in G$*, where* G *is finite, is the value* a *such that* $g^a = 1$*, where 1 denotes the identity with respect to* $\cdot$*.*

2.1.2 The distribution of primes

The public-key algorithms considered in this document use prime numbers and their properties in order to work correctly. Since the keys in these systems are mostly composed of prime numbers, one might wonder: are there enough prime numbers for every application? How many of them are there?

Theorem 1. *The number of prime numbers is infinite.*

This theorem, stated and proved by Euclid, shows that there is an infinite amount of primes [38]. However, it doesn't say much about how primes are *distributed*. We can, then, reformulate the question: how many primes are there *less than or equal to an arbitrary number* x?

Theorem 2. *Let* $\pi(x)$ *be the number of primes less than or equal to* x*. Thus,*

$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\ln x}} = 1 \tag{2.1}$$

From this result, we see that there are enough primes to avoid any kind of indexing or search. For instance, there are about 3.778×10^{151} primes less than 2^{512}. The chance that an attacker will guess a 512 bit key by picking randomly is for all practical purposes 0.

From Theorem 2 we can also conclude that given a randomly chosen integer x, the probability of this number being prime is about $\frac{1}{\ln x}$.

2.1.3 Fermat's Little Theorem

In 1640 Pierre de Fermat stated a theorem of great importance in Number Theory, which was later proven by Euler and Leibniz. Euler's proof can be seen in [38].

Theorem 3. *If* p *is a prime number and* a *is relatively prime to* p*,*

$$a^{p-1} \equiv 1 \pmod{p} \tag{2.2}$$

This theorem might mislead the reader into believing that a single exponentiation can be used to prove the primality of a number. While the theorem is true for any prime number, it also is true for some composite numbers. A composite number c such that $a^{c-1} \equiv 1 \pmod{c}$ is called *pseudo-prime of base a*. These numbers are rare though, and usually a pseudo-prime of base a is not of base b.

Definition 4. *A composite number c such that $a^{c-1} \equiv 1 \pmod{c}$ for any integer a is called* Carmichael number.

Pomerance proved that there is an upper bound of $x^{1-\ln\ln\ln x/\ln\ln x}$ Carmichael numbers less than or equal to x[27]. Thus, for an input of 2^{512} there is a maximum of 4.60145×10^{107} Carmichael numbers; the probability of randomly picking a Carmichael number is about 3.43191×10^{-47}.

Another useful result of Theorem 3 is obtained by multiplying both sides of Equation 2.3 by a^{-1}:

$$a^{-1}a^{p-1} \equiv a^{-1}1 \pmod{p} \Rightarrow a^{p-2} \equiv a^{-1} \pmod{p} \tag{2.3}$$

If p is indeed prime, one can obtain the multiplicative inverse of an arbitrary number a in $\mathbb{Z}_p^*$ with a single exponentiation. While this is not the asymptotically fastest algorithm to obtain inverses, this result might prove useful when divisions are prohibitively slow or one wants to avoid branching.

2.1.4 Chinese Remainder Theorem

The Chinese Remainder Theorem is an old method that enables one to obtain an integer value given its residues modulo a system of smaller moduli, often called *basis*. One of the earliest applications of this method was to count soldiers by counting the 'remainder' when they were lined up in justified rows of varying number [27, Section 2.1.3].

Theorem 4. *Let $m_1, \ldots, m_r$ be positive, pairwise coprime moduli, whose product is $M = \prod_{i=1}^{r} m_i$. Let r residues n_i also be given. Then the system*

$$n \equiv n_i \pmod{m_i}, 0 \leq n < M, 1 \leq i < r \tag{2.4}$$

Has a unique solution given by

$$n = \sum_{i=1}^{r} n_i v_i M_i \pmod{M} \tag{2.5}$$

where $M_i = M/m_i$ and $v_i = M_i^{-1} \pmod{m_i}$.

We can see, then, that any positive integer less than M can be represented uniquely by the set of residues modulo each m_i. We can also point out that the reconstruction can be done in r multiplications, since $v_i \times M_i$ do not change for each basis and thus can be precomputed.

2.2 Symmetric-Key Cryptography

Typically, the algorithms used in bulk encryption make use of a single key for both encryption and decryption — secret-key algorithms. Symmetric-key encryption algorithms are usually divided in two main categories: *block ciphers and stream ciphers*.

Block ciphers, as the name implies, work by invertibly mapping an n-bit block of plaintext to an n-bit block of ciphertext. The cipher takes as parameter a k-bit key, on which security rests upon. Examples of block ciphers are AES, Blowfish, DES, IDEA, etc [54, Chapter 7].

Stream ciphers, on the other hand, encrypt a message one bit (or more commonly in computing applications, byte) at a time. They are especially important when buffering is limited or when bytes must be individually processed as they are received. Also, since each byte is encrypted/decrypted individually there is no error propagation beyond the error rate of the channel itself. Most stream ciphers work by generating a pseudo-random sequence of bits based on a *seed* or key. The encryption/decryption is then simply done by mixing the plaintext/ciphertext with the generated sequence using the XOR operation. These are called *Synchronous stream ciphers* [54, Chapter 6].

2.2.1 AES

NIST[2] announced in 1997 their intent to choose a successor to the old DES cipher, which since its inception had become vulnerable to attacks given its relatively small key (56 bits). This successor was to be named Advanced Encryption Standard or simply AES, and input was sought from interested parties both on how the cipher should be chosen and in cipher proposals. NIST stipulated that the candidates should be block ciphers with block sizes of 128 bits, and key sizes of 128, 192 and 256 bits. The winning proposal, Rijndael, was announced in 2001 as the new standard for symmetric encryption [62].

AES operates on a 4×4 array of bytes, corresponding to the block size. The cipher itself is composed of a simple initial round, a variable number

[2]U.S. National Institute of Standards and Technology

(depending on the key size) of rounds and a final round. In each of these rounds, the elements of the 4×4 array are replaced using a substitution table, cyclically shifted, multiplied by a polynomial over a finite field and finally mixed with the round key using the XOR operation. Since each round has a different *round key*, the key schedule of the AES is responsible to derive the key for each round from the main given key [62].

2.2.2 Salsa20

The NESSIE (New European Schemes for Signatures, Integrity and Encryption) project was an European research project aimed at finding secure cryptographic primitives, covering all objectives of cryptography. When no stream ciphers could be selected, since all had been broken during the selection process, a new competition called eSTREAM was created in order not only to find good stream ciphers but also to incentive study in this area [33]. Two profiles were created in this competition: stream ciphers designed to be run in software applications and stream ciphers designed to be implemented in hardware. Salsa20 is one of the most successful proposals in the software profile and thus was chosen as part of the final portfolio.

Salsa20 works by hashing a 256-bit key, a 64-bit IV (initialization vector) and a 64-bit block counter into a pseudo-random 64 byte block that is mixed with the plaintext using XOR. The block counter is incremented and the hash function is computed again each time a 64 byte boundary is reached during encryption or decryption.

The Salsa20 core function builds a 4×4 table of 32 bit words from the key, IV, counter and constants. Then a series of rounds composed of additions, bit rotations and exclusive-ors are performed to achieve a random permutation of the original inputs [15]. Originally the number of rounds was set to 20; however, the version of cipher included in the eSTREAM portfolio was reduced to 12 rounds, for performance reasons. This reduced round version of Salsa20 is denominated Salsa20/12 [9].

2.3 Public-Key Cryptography

As previously mentioned, public-key cryptography relies on a pair of keys for its operation. The public key, as the name implies, can be known to everyone; the private key must remain known only to its owner. This concept allows to devise numerous cryptographic methods for various uses, such as key exchanges, encryption, digital signatures, etc. This document will only cover a small part of all of these, namely the most common and trusted

methods available.

2.3.1 Diffie-Hellman Key Exchange

The method introduced by Diffie and Hellman in [30] for key exchanges was the first to use the above concept of public and private keys. Algorithm 1 describes the exchange of keys between two parties, called A and B.

Algorithm 1 Diffie-Hellman Key Exchange

1: A e B agree on a prime p and a generator α in $\mathbb{Z}_p^*$.
2: A picks a random number x in [2, p-2], computes $\alpha^x \bmod p$ and sends the result to B.
3: B picks a random number y in [2, p-2], computes $\alpha^y \bmod p$ and sends the result to A.
4: A receives $\alpha^y \bmod p$ and computes the final key $K = (\alpha^y)^x \bmod p$.
5: B receives $\alpha^x \bmod p$ and computes the final key $K = (\alpha^x)^y \bmod p$.

In this case, x is the *private key* of A and $\alpha^x \bmod p$ is A's public key. Similarly, y is the private key of B and $\alpha^y \bmod p$ is its public key. The key exchange consists then in each party sending his public key to the respective recipient, while keeping their private key private. The key exchange works due to the commutative property of exponentiation: $(\alpha^x)^y \bmod p \equiv (\alpha^y)^x \bmod p \equiv \alpha^{xy} \bmod p$.

Suppose that a foe has the ability to eavesdrop all communications between A and B. This means he's able to known both $\alpha^x \bmod p$ and $\alpha^y \bmod p$. The problem of finding $\alpha^{xy} \bmod p$ given $\alpha^x \bmod p$ and $\alpha^y \bmod p$ is called *Computational Diffie-Hellman Problem*. The best known way to solve this problem is to solve the discrete logarithm for either one of the public keys, obtaining either x or y. There are no known polynomial time algorithms to solve the latter, for appropriately chosen fields and generators.

The choice of good fields and generators is indeed important — some attacks are able to ruin the security of this scheme if p is poorly chosen. For example, if $p - 1$, the order of the field, can be factored in relatively small primes, the discrete logarithm can be computed modulo each of these small primes and the complete discrete logarithm can be recovered using Theorem 4, rendering all key exchanges in this field unsecure [65]. Thus, it is recommended that the field prime used for this particular use be a *safe prime* — a prime p such that $\frac{p-1}{2}$ is also prime. IETF[3] proposes several adequate

[3]Internet Engineering Task Force

prime fields for Diffie-Hellman key exchanges in RFC 4306 and RFC 3526 [44] [46].

2.3.2 RSA

RSA, named after its inventors, was the first published algorithm that allowed both key exchange, encryption and digital signatures making use of the public-key concept [71]. However, the first digital signature standard based on RSA only appeared decades later, as ISO/IEC 9796. The key generation process for RSA is slightly more complex than Diffie-Hellman's and is described in Algorithm 2.

Algorithm 2 RSA Key Generation

Require:
Ensure: N, E, D
1: Randomly choose two prime numbers p, q.
2: $N = pq$
3: $\varphi = (p-1)(q-1)$
4: Choose a public exponent E coprime to φ.
5: $D = E^{-1} \pmod{\varphi}$

In the RSA algorithm, the public key is the pair (N, E). The private key is D. The security of this algorithm relies on the hardness of finding D given E and N. Given these, the easiest way to obtain D is to factor N and repeat the key generation process knowing the original p and q. As with the discrete logarithm problem, no known polynomial time algorithms are known for factoring large numbers, making this a hard problem.

The encryption of a message using RSA consists of a single exponentiation, as shown in Algorithm 3.

Algorithm 3 RSA Encryption

Require: M, N, E
Ensure: C
$C = M^E \bmod N$

Algorithm 4 RSA Decryption

Require: C, N, D
Ensure: M
$M = C^D \bmod N$

It is possible to verify why RSA works by referring to a generalization of Theorem 3 presented by Euler:

$$a^{\varphi(n)} = 1 \pmod{n} \tag{2.6}$$

Since we know that $E \times D = 1 \pmod{\varphi(N)} \Rightarrow E \times D = 1 + k \times \varphi(N)$, we have:

$$\begin{aligned} a^{ED} &= a \cdot (a^{\varphi(N)})^k \pmod{N} \equiv \\ a^{ED} &= a \cdot 1^k \pmod{N} \equiv \\ a^{ED} &= a \pmod{N} \end{aligned} \tag{2.7}$$

The signature process works in the inverse way as encryption and decryption: the private key is used to sign the message and the public key can be used to verify its authenticity. Both signature and verification are described in Algorithm 5 and 6 respectively.

Algorithm 5 RSA Signature creation

Require: M, N, D
Ensure: S
$H_m = H(M)$
$S = H_m^D \bmod N$

Algorithm 6 RSA Signature Verification

Require: S, M, N, E
$H_m = H(M)$
if $H_m^E \bmod N = S$ **then**
return Valid signature
else
return Invalid signature
end if

2.3.3 DSA

Digital Signature Algorithm, also known as Digital Signature Standard, is a standard first proposed in 1991 by NIST[4] and is the first digital signature scheme to ever be recognized by any government. The scheme itself

[4]U.S. National Institute of Standards and Technology

is loosely based on the ElGamal method [5, 34], and is a digital signature with appendix, i.e. the signature is appended to the message to send. When otherwise omitted, the hash function used by DSA is SHA-1 [5, 4].

The key generation process is described in Algorithm 7. a is the entity's private key, whereas p, q, α and y can be known to everyone.

Algorithm 7 DSA Key Generation

Require:

Ensure: p, q, α, y, a

1: Randomly pick a prime q with 160 bits.
2: Select a 1024 bit prime p, where $p - 1$ divides q.
3: Find a generator α in $\mathbb{Z}_p^*$ of order q.
4: Pick a random integer a, $0 < a < q$.
5: Compute $y = \alpha^a \bmod p$.

Algorithm 8 DSA Signature Generation

Require: M, p, q, a, α

Ensure: r, s

1: Randomly pick an integer k, $0 < k < q$
2: $r = (\alpha^k \bmod p) \bmod q$
3: $s = (k^{-1}(SHA1(M) + ar)) \bmod q$
4: **return** r, s

Similarly to most other public key schemes, DSA bases its strength on a hard problem — the discrete logarithm problem. Logically, if one is able to derive a from $y = \alpha^a$, one would be able to forge anyone's signature without effort. But as the discrete logarithm is a hard problem for properly chosen parameters as discussed in Section 2.3.1, this signature scheme is considered secure.

2.4 Cryptography in GPUs

As far as the author is aware, there is not a single cryptographic primitive library oriented to GPUs. However, some research has been done in this direction recently; we proceed to summarize some important work done in the area.

Cook et al. studied the feasibility of implementing symmetric-key ciphers in a GPU using the OpenGL API. While this API was not general purpose and very limited, it was possible to implement the AES using it [23]. The

Algorithm 9 DSA Signature Verification

Require: r, s, M, p, q, y, α
1: **if** $0 < r < q$ and $0 < s < q$ **then**
2: $w = s^{-1} \bmod q$
3: $u1 = w \times SHA1(M) \bmod q$
4: $u2 = rw \bmod q$
5: $v = (\alpha^{u1}\alpha^{u2} \bmod p) \bmod q$
6: **if** $v = r$ **then**
7: **return** Valid Signature
8: **else**
9: **return** Invalid Signature
10: **end if**
11: **else**
12: **return** Invalid Signature
13: **end if**

performance obtained was low, since common CPU implementations were up to 40 times faster than the OpenGL implementation. Yamanouchi used newer OpenGL extensions specific to the NVIDIA Geforce 8 series to implement the same cipher, AES [81]. The performance figures obtained in this implementation were much higher, with the GPU's throughput going as high as 95 MB/s, against 55 MB/s on the reference CPU[5].

More recently Rosenberg and, independently, Manavski used the NVIDIA CUDA technology to implement AES [72, 52]. Rosenberg extended the OpenSSL library, adding a GPU engine that can be used by any application using OpenSSL. The performance obtained closely matched the one of a 3.2 GHz CPU; Manavski's work, on the other hand, obtained throughputs of 1 GB/s using a NVIDIA 8800GTX.

GPUs have also been used to attack cryptographic systems. One example is the bruteforcing of passwords, usually stored as their hash computed using a *hash function*. Due to their inherent parallelism, GPUs are able to compute thousands of hashes simultaneously, accelerating by an order of magnitude the search for a password given its hash. In the particular case of the MD5 hash function, GPUs are able to compute or verify up to 600 million hashes per second; a CPU can only compute up to 30 million hashes per second (per core)[1].

Scott and Costigan used the IBM Cell processor to accelerate RSA in the OpenSSL library [26]. Compared to the general purpose processor of Cell,

[5]The CPU used in the benchmarks was an Intel Pentium 4 3.0 GHz; the GPU was a NVIDIA Geforce 8800GTS 640 MB

the specialized vector cores (SPUs) were around 1.5 times faster. When all SPUs were used to compute the same operation in cooperation, a speedup of about 7.5 was obtained. However, the instruction set and architecture of this CPU, while oriented to vector operations is considerably different than that of a common GPU. More recently, Costigan and Schwabe implemented a fast elliptic curve Diffie-Hellman key exchange in the Cell processor [25]. They conclude that in terms of throughput and performance/cost ratio, the Cell is competitive with the best current general purpose CPUs and implementations — the 6-SPU Sony PlayStation 3 could perform 27474 elliptic curve operations per second, whereas an Intel Q9300, using all 4 cores, performed 27368.

Payne and Hitz implemented multiple-precision arithmetic in GPUs using residue number systems (RNS) [64]. The authors conclude this approach provides good performance, but the overhead involved in transferring data from RAM to the GPU's memory makes the method worth it only for numbers of considerable size.

Moss, Page and Smart used the NVIDIA Geforce 7800 GTX graphics accelerator card to compute modular exponentiations, the main bottleneck in RSA, also using residue number systems [60]. Due to the overhead of copying data from the GPU and back, the authors conclude that a speedup is only achieved by computing numerous different exponentiations simultaneously; their speedup when computing 100000 modular exponentiations was of up to 3, compared to the reference CPU. Also on the NVIDIA 7800GTX, Fleissner implemented an accelerated Montgomery method for modular exponentiation in GPUs [35]. However, he only worked with 192-bit moduli, far too small to be useful in cryptographic applications.

Recently, Szerwinski et al. employed the newer G80 architecture from NVIDIA and the CUDA API to develop efficient modular exponentiation and elliptic curve scalar multiplication [77]. Their work, which included implementations of both Montgomery and RNS arithmetic, yielded a throughput of up to 813 modular exponentiations per second in an NVIDIA 8800GTS; the minimum latency for this throughput, however, was of over 6 seconds.

2.5 Implementing Public-Key Algorithms

In order to have acceptable security margins, the numbers used in the algorithms of the previous section tend to be much larger than the processor's natural register length. Thus, methods to represent and perform arithmetic with large numbers become necessary. This section describes some of the most common methods used in cryptographic libraries implementing public-

key algorithms.

2.5.1 Classic arithmetic

Representation

Typically, multiple precision numbers are represented as vectors of digits of the CPU's register size length. As an example the number represented by a vector v of n digits in base $\beta = 2^w$ is given by

$$\sum_{i=0}^{n-1} v_i \times \beta^i$$

Addition and Subtraction

The algorithms to add and subtract numbers represented in this manner are rather simple; in fact, they're similar to the methods learned in elementary school. Both operations have linear complexity. However, due to the carry propagation across words, these operations are hard to parallelize. Algorithm 10 and 11 describe the algorithms to add and subtract large numbers respectively.

Algorithm 10 Multiple Precision Integer Addition

Require: x, y
Ensure: $z = x + y$
1: $c \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad t \leftarrow x_i + y_i + c$
4: $\quad z_i \leftarrow t \bmod \beta$
5: $\quad c \leftarrow \lfloor t/\beta \rfloor$
6: **end for**
7: $z_{n+1} \leftarrow c$

Multiplication and Division

Classical multiplication and division are identical to those learned in elementary school, as shown in Algorithm 12 and 13. Their complexity is $O(n^2)$ in both cases, for numbers of n digits. There are faster methods for multiplication and division, but they are usually only of advantage for relatively large numbers. A detailed account of multiplication and division algorithms can be consulted in [47, Section 4.3] [27, Chapter 9] [22] [54, Chapter 14].

Algorithm 11 Multiple Precision Integer Subtraction

Require: $x, y, x >= y$
Ensure: $z = x - y$
1: $c \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad t \leftarrow x_i - y_i + c$
4: $\quad z_i \leftarrow t \bmod \beta$
5: $\quad c \leftarrow \lfloor t/\beta \rfloor$
6: **end for**
7: $z_{n+1} \leftarrow c$

Algorithm 12 Multiple Precision Number Multiplication

Require: x, y
Ensure: $z = x * y$
1: **for** $i = 0$ to $n + t$ **do**
2: $\quad z_i \leftarrow 0$
3: **end for**
4: **for** $i = 0$ to $t - 1$ **do**
5: $\quad c \leftarrow 0$
6: $\quad$ **for** $j = 0$ to $n - 1$ **do**
7: $\quad\quad p \leftarrow z_{i+j} + x_j * y_i + c$
8: $\quad\quad z_{i+j} \leftarrow p \bmod \beta$
9: $\quad\quad c \leftarrow \lfloor p/\beta \rfloor$
10: $\quad$ **end for**
11: $\quad z_{i+n} \leftarrow c$
12: **end for**
13: **return** z

Algorithm 13 Multiple Precision Number Division

Require: x, y
Ensure: $z = \lfloor x/y \rfloor, r = x \bmod y$
1: **for** $i = 0$ $to n - t - 1$ **do**
2: $z_i \leftarrow 0$
3: **end for**
4: **while** $x \geq y \times \beta^{n-t}$ **do**
5: $z_{n-t-1} \leftarrow z_{n-t-1} + 1$
6: $x \leftarrow x - y \times \beta^{n-t}$
7: **end while**
8: **for** $i = n - 1$ to t **do**
9: **if** $x_i = y_t$ **then**
10: $z_{i-t-1} \leftarrow \beta - 1$
11: **else**
12: $z_{i-t-1} \leftarrow \lfloor \frac{x_i \times \beta + x_{i-1}}{y_t} \rfloor$
13: **end if**
14: **while** $z_{i-t-1}(\beta y_t + y_{t-1}) > (x_i \beta^2 + x_{i-1}\beta + x_{i-2})$ **do**
15: $z_{i-t-1} \leftarrow z_{i-t-1} - 1$
16: **end while**
17: $x \leftarrow x - z_{i-t-1} y \beta^{i-t-1}$
18: **if** $x < 0$ **then**
19: $x \leftarrow x + y\beta^{i-t-1}$
20: $z_{i-t-1} \leftarrow z_{i-t-1} - 1$
21: **end if**
22: **end for**
23: $r \leftarrow x$
24: **return** z, r

Exponentiation

Exponentiation can be done in $O(\log e)$ multiplications, where e is the exponent, using the binary method described in Algorithm 14. The complexity of exponentiation in elementary operations is directly related to the multiplication algorithm employed or, in the case of modular exponentiation, the modular reduction method used. If Algorithm 12 is used along with the binary method, the complexity is $O(n^3)$.

The binary method can be seen as a specific case of an *addition chain*.

Definition 5. *An addition chain a of length d for some integer n is a sequence of integers $a_0 \ldots a_d$, where $a_0 = 1$ and $a_d = n$ and for every $1 \leq i \leq d$ there exist j and k such that $s_i = s_j + s_k$.*

Algorithm 14 Integer Exponentiation — Binary Method

Require: x, y
Ensure: $z = x^y$
1: $A \leftarrow 1$
2: $S \leftarrow x$
3: **while** $y \neq 0$ **do**
4: **if** $y \bmod 2 = 1$ **then**
5: $A \leftarrow A \times S$
6: **end if**
7: $y \leftarrow \lfloor y/2 \rfloor$
8: $S \leftarrow S^2$
9: **end while**
10: **return** A

The problem of finding the *shortest* addition-chain containing m integers has been proven to be NP-complete [32]. However, there are methods to find *good* addition-chains in linear time. The binary method described above takes a maximum of $2 \log_2 e$ multiplications. This bound can be improved by generalizing this method to higher bases.

The *k-ary method* performs a small pre-computation of 2^k exponents, then processes k bits of the exponent per iteration. The total maximum number of multiplications is reduced to $2^k + \log_2 e + (\log_2 e)/k$. For adequately chosen k, this provides a significant, albeit constant, speedup over the binary method. Algorithm 15 describes this method in detail.

Algorithm 15 Integer Exponentiation — k-ary Method

Require: x, y
Ensure: $z = x^y$
1: $z \leftarrow 1$
2: Precompute $g_i = x^i, 0 \leq i \leq k$
3: **for** $i = \log_k y - 1$ to 0 **do**
4: $z \leftarrow z^{2^k}$
5: $z \leftarrow z g_{y_i}$
6: **end for**
7: **return** z

As a practical example, consider a 1024-bit exponent. The binary algorithm takes a maximum of $2 \times 1024 = 2048$ multiplications; the k-ary method, using a window size of 5 bits, takes a maximum of $2^5 + 1024 + 1024/5 = 1260$. Although these figures are for the worst case scenario, similar speedups are

seen in the average case.

There are many other algorithms and improvements to addition chains in the literature. Binary exponentiation goes back to 200 B.C. [47, Section 4.6.3]; the k-ary method was first introduced by Brauer in [21]. Since then, many improvements were made; a detailed survey of the state-of-the-art in exponentiation methods can be consulted in [13].

2.5.2 Barrett Modular Multiplication

Barrett in [11] devised a faster modulo reduction scheme than the one described in Algorithm 13. The first observation was that division can be performed by multiplication by the reciprocal of the divisor. Another observation is that modular reduction can be performed by the expression

$$x \bmod p = x - p\lfloor \frac{x}{p} \rfloor \tag{2.8}$$

Since there isn't such a thing as a reciprocal in the integer ring, one can simulate the reals by fixed-point arithmetic — a real r is represented by an integer n with $p+q$ bits, where $r = \lfloor n/2^q \rfloor$. Thus, division can be performed by the simple, all-integer expression

$$\lfloor \frac{x}{y} \rfloor \approx \lfloor (x\lfloor 2^q/y \rfloor)/2^q \rfloor \tag{2.9}$$

If the divisor y is known *a priori*, one can simply precompute $\lfloor 2^q/y \rfloor$, thus performing the division with one multiplication.

Plugging this result into Equation 2.8 and computing $\mu = \lfloor 2^q/p \rfloor$, we get

$$x \bmod p = x - p\lfloor x\mu/2^q \rfloor \tag{2.10}$$

As long as $2^q \geq x$, the quotient will be either correct or off by 1. In this case, a correction step is needed to ensure the correctness of the modular reduction. Since for cryptographic purposes x is usually less than p^2, we have $2^q \geq p^2$.

As an example, fix $x = 23, y = 17$ and $p = 31$. First, one precomputes μ: $\mu = \lfloor 2^{10}/31 \rfloor = 33$. xy is easily calculated using Algorithm 12: $xy = 23 \times 17 = 391$.

The modular reduction step then requires the calculation of $391 - 31\lfloor (391 \times 33)/2^{10} \rfloor = 391 - 31 \times 12 = 19$. Since $19 < 31$, no further steps are necessary.

Algorithm 16 Barrett Modular Multiplication

Require: $x, y, N, q = 2\lceil \log_2 N \rceil, \mu = \lfloor 2^q / N \rfloor$
Ensure: $z = xy \bmod N$
1: $t \leftarrow xy$
2: $z \leftarrow t - N\lfloor t\mu / 2^q \rfloor$
3: **while** $z >= N$ **do**
4: $\quad z \leftarrow z - N$
5: **end while**

2.5.3 Montgomery Multiplication

Montgomery multiplication is one of the most used algorithms when several operations modulo the same value are necessary. It requires a small overhead before and after the actual calculation, but effectively eliminates the need for divisions from the modular reductions after each multiplication.

Basically Montgomery's method replaces the number to reduce $x \pmod N$ by another, given by $x' = xR \pmod N$. With x in this representation, one can compute modular reductions without explicit divisions such as the one in Algorithm 13 [58].

Algorithm 17 Montgomery Reduction — REDC

Require: $N, R, x, N' = -N^{-1} \bmod R$
Ensure: $z = xR^{-1} \bmod N$
1: $t \leftarrow x + N(xN') \bmod R$
2: $z \leftarrow \lfloor t/R \rfloor$
3: **if** $z >= N$ **then**
4: $\quad z \leftarrow z - N$
5: **end if**

The algorithm works because $N(xN') \equiv -x \pmod R$, thus t/R is guaranteed to be an integer. Plus, t is known to be congruent to $x \bmod N$ — $x + x(NN') \equiv x \bmod N$.

In order to convert an integer to Montgomery's representation, one can use Algorithm 17 to reduce the product $xR^2R^{-1} \bmod N \equiv xR \bmod N$. Thus, it is useful to precompute $R^2 \bmod N$, if N and R are known *a priori*. To convert a number from Montgomery's representation to the classic one, one can again use Algorithm 17 on the output of the last reduction: $xR \bmod N$ — $xRR^{-1} \bmod N \equiv x \bmod N$. Note that there *must* exist an inverse of $N \bmod R$ — N and R must be coprimes.

The choice of R in this method is crucial to the speedup provided by this method. Good choices for R are typically powers of 2 that are also

multiples of β, since they avoid both logical operations and explicit shifts, thus accelerating modular multiplications.

To perform one single modular multiplication, this method is far from optimal, since the precomputation overhead might be too large. However, when several modular multiplications are performed in sequence, such as modular exponentiations or elliptic curve group operations, this method saves quite a few divisions (for good choices of R), resulting in increased performance.

A simple example is now presented: $5 \times 6 \bmod 11$. We compute $x' = 5 \times 32 \bmod 11 = 6, y' = 6 \times 32 \bmod 11 = 5, N = 11, R = 32, -N^{-1} \bmod R = 29$.

Now, the modular multiplication and reduction:

$$\begin{aligned} xy &= 5 \times 6 = 30 \\ t &= 30 + 11((30 \times 29) \bmod 32) = 96 \\ z &= \frac{96}{32} = 3 \end{aligned}$$

Note that the mod32 operation can be easily performed using a logical AND, and the division by 32 can be as easily done with a shift right by 5. All there is left to do is convert the result from Montgomery's representation $xR \bmod N$ to $x \bmod N$. This can be done by reducing the result once again:

$$\begin{aligned} t &= 3 + 11((3 \times 29) \bmod 32) = 256 \\ z &= \frac{256}{32} = 8 \end{aligned}$$

Thus, $5 * 6 \bmod 11 = 8$.

Montgomery multiplication can easily be plugged into any conventional modular exponentiation algorithm, such as Algorithm 14. Therefore, given an exponent e one can trade $O(\log e)$ divisions by $O(\log e)$ multiplications and cheap logical operations, faster on typical computer architectures.

2.5.4 Special Moduli

It is possible to take advantage of the special form of some moduli to simplify modular reduction. When a modulus is close to a power of 2, one can exploit the binary nature of computers to carry out the reduction more efficiently.

Suppose one wants to reduce a positive integer $x < 2^{64}$ modulo $p = 2^{32} - 5$. Assuming a base $\beta = 2^{32}$, one can represent x as a number comprised of two 32-bit digits:

$$x = x_0 + 2^{32}x_1$$

Also observe that $2^{32} \bmod 2^{32} - 5$ is easy to compute — 5. Thus,

$$x = x_0 + 5x_1 \pmod{p}$$

More generally,

$$x \bmod (2^t - c) = x \bmod 2^t + c\lfloor x/2^t \rfloor \pmod{(2^t - c)} \tag{2.11}$$

It is straightforward to generalize this process to more than 2 digits and make it iterative. Note that the smaller the c coefficient, the faster this method will be. It is the fastest for *Mersenne numbers*, where $c = 1$ — only modular addition is required. Algorithm 18 describes this modular reduction method for numbers of arbitrary length.

Algorithm 18 Reduction by Special Modulus

Require: $x, p = 2^t - c$
Ensure: $z = x \bmod p$
1: $z \leftarrow x$
2: **while** $z > 2^t$ **do**
3: $\quad t \leftarrow \lfloor z/2^t \rfloor$
4: $\quad z \leftarrow z \bmod 2^t$
5: $\quad z \leftarrow z + ct$
6: **end while**
7: **if** $z \geq p$ **then**
8: $\quad z \leftarrow z - p$
9: **end if**

2.5.5 Residue Number Systems

In a residue number system, each number n is represented by the set of its residues $n_i \bmod m_i$, where m_i is a set of pairwise coprime moduli whose product is $M = \prod_{i=1}^{r} m_i$. By Theorem 4 we know that any positive integer less than M can be unambiguously represented by its residues modulo m_i.

All operations in a residue number system can be done independently for each residue. This makes this system look highly attractive for parallel architectures. Although additions, subtractions and multiplications are trivial to perform in these systems, divisions can only be done when the result is known to be exact (i.e. no remainder). Magnitude comparisons, with the exception of verifying equality, are very hard to perform without converting the

number back to a classic representation. Several schemes to perform modular multiplication and exponentiation in residue number systems have been proposed that keep a high parallelization potential for their implementation [76, 60, 10, 18].

As an example take the numbers $x = 29, y = 21$ and moduli $B = \langle 5, 7, 11, 13 \rangle$. Then, $M = 5 \times 7 \times 11 \times 13 = 5005$. As both x and y are less than M, they can be uniquely represented by their residues modulo B:

$$x' = \langle 4, 1, 7, 3 \rangle$$
$$y' = \langle 1, 0, 10, 8 \rangle$$

Multiplying x' by y' is simply done by the individual multiplication of each component:

$$\begin{aligned} xy' &= \langle 4 * 1 \bmod 5, 0 * 1 \bmod 7, 7 * 10 \bmod 11, 3 * 8 \bmod 13 \rangle \\ &= \langle 4, 0, 4, 11 \rangle \end{aligned}$$

In order to recover xy' to a conventional representation, we'll use Equation 2.5:

$$\begin{aligned} xy &= 4 \times \frac{5005}{5} \times 1 + 0 \times \frac{5005}{7} \times 1 + 4 \times \frac{5005}{11} \times 3 + 11 \times \frac{5005}{13} \times 5 \pmod{5005} \\ &= 609 \end{aligned}$$

We can confirm that 21×29 is, indeed, 609.

As previously mentioned, division is particularly hard to perform under residue number systems. Thus, modular reduction, as required by modular exponentiation, is also non-trivial to perform. This section describes two known methods to perform modular reductions under these number systems.

RNS Montgomery Multiplication

One approach to perform a modular reduction in RNS is to create an analog of Algorithm 17. As described in Section 2.5.3, if we choose an integer R such that division by such integer is easy, one can perform a modular reduction $xR^{-1} \bmod N$. In a RNS, R is the product of the moduli set, i.e. $\prod_{i=1}^{r} m_i$. Thus, one can compute $xR^{-1} \bmod N$ as $(x_i + n_i(-x_i n_i^{-1}))m_i^{-1} \bmod m_i$ for each modulus m_i.

The reader may notice, however, that division by R, i.e. $m_i^{-1} \bmod m_i$ is impossible to compute in the current basis, since $R \equiv 0 \pmod{m_i}$. Thus,

one needs to add a second basis, at least as large as the first, convert the current result to that basis, perform the division and finally convert back to the original basis. This conversion process between bases is often called *base extension*. There are many published base extension methods in the literature; the best ones appear to cost $k^2 + 2k$ modular multiplications to compute a base extension, for bases of k moduli [45]. Algorithm 19 describes RNS modular multiplication using this method.

Algorithm 19 RNS Montgomery Multiplication

Require: $a_i^m = A \bmod m_i, a_i^p = A \bmod p_i, b_i^m = B \bmod m_i, b_i^p = B \bmod p_i, m_i, p_i, N$

Ensure: $z_i^m = ABP^{-1} \bmod m_i$

1: Precompute $\mu_i = -N^{-1} \bmod p_i, n_i = N \bmod m_i, \lambda_i = P^{-1} \bmod m_i$
2: $s_i^m \leftarrow a_i^m b_i^m \bmod m_i$
3: $s_i^p \leftarrow a_i^p b_i^p \bmod p_i$
4: $t_i^p \leftarrow s_i^p \mu_i \bmod p_i$
5: $t_i^m \leftarrow BaseExtend(t_i^p, m_i, p_i)$ ▷ Convert t_i^p into t_i^m
6: $u_i^m \leftarrow t_i^m n_i \bmod a_i^m$
7: $v_i^m \leftarrow (s_i^m + u_i^m) \bmod m_i$
8: $z_i^m \leftarrow v_i^m \lambda_i \bmod m_i$
9: $z_i^p \leftarrow BaseExtend(z_i^m, m_i, p_i)$ ▷ Convert z_i^m into z_i^p

Step 9 of Algorithm 19 is only necessary when multiple modular multiplications are performed, i.e. when the output of a multiplication is the input of another. For single modular multiplications this step can be avoided. The number of modular multiplications of the whole algorithm is $2k^2+9k$. Bajard et al. improved this operation count to $2k^2 + 8k$ [10].

Explicit CRT

The Explicit Chinese Remainder Theorem, like Theorem 4, allows the recovery of an integer n given its residues modulo a set of coprime moduli m_i. Whereas in Theorem 4 the product sum must be reduced modulo P to obtain the final result, the Explicit CRT theorem does not have such requirement, by means of approximation of the quotient.

Theorem 5. *Let $m_1, \ldots, m_r$ be positive, pairwise coprime moduli, whose product is $M = \prod_{i=1}^{r} m_i$. Let r residues n_i also be given, representing the integer $n < M/2$. Let $round(x)$ be the unique integer r such that $|x-r| < 1/2$, where $x - 1/2 \notin \mathbb{Z}$. Let $M_i = M/m_i$, $v_i = M_i^{-1} \pmod{m_i}$ and $x_i = n_i v_i$.*

Then,

$$n = \sum_{i=1}^{r} x_i M_i - M \cdot r \tag{2.12}$$

where $r = round(\alpha), \alpha = \sum_{i=1}^{r} x_i/m_i$.

Several different approaches have been proposed for the calculation of the α coefficient. Montgomery and Silverman propose using floating-point arithmetic to perform a low-precision approximation of α [59]. Bernstein proposes using fixed-point arithmetic and provides precision bounds for which the α approximation can be made without error [14].

Theorem 5 can be extended to work in the underlying ring of the numbers modulo p, given the same initial assumptions and the added restriction that $n_i v_i$ must be reduced modulo m_i:

$$n = \sum_{i=1}^{r} x_i (M_i \bmod p) - (M \bmod p) r \tag{2.13}$$

Since $n_i v_i \bmod m_i < m_i$ and Theorem 5 is correct, n is congruent to $n \pmod p$ and cannot be larger than $p \sum_{i=1}^{r} m_i$. Furthermore, this identity also holds for $p \bmod m_j$, rendering this a highly parallelizable reduction method:

$$n \equiv \sum_{i=1}^{r} x_i (M_i \bmod p \bmod m_j) - (M \bmod p \bmod m_j) r \pmod{m_j} \tag{2.14}$$

Algorithm 20 describes a modular multiplication using the identity in Equation 2.14.

Algorithm 20 Explicit CRT Modular Multiplication

Require: $a_i = A \bmod m_i, b_i = B \bmod m_i, m_i, N$
Ensure: $z_i \equiv AB \pmod{m_i}$
1: Precompute $q_i = (M/m_i)^{-1} \bmod m_i, c_i = M \bmod p \bmod m_i, d_{ij} = M/m_i \bmod p \bmod m_j$
2: $t_i \leftarrow a_i b_i q_i$
3: $\alpha \leftarrow 0$
4: **for** i in 1 to r **do**
5: $\quad \alpha \leftarrow \alpha + t_i/m_i$
6: **end for**
7: **for** i in 1 to r **do**
8: $\quad sum \leftarrow 0$
9: $\quad$ **for** j in 1 to r **do**
10: $\quad\quad sum \leftarrow (sum + t_j d_{ij}) \bmod m_i$
11: $\quad$ **end for**
12: $\quad prod \leftarrow \alpha c_i \bmod m_i$
13: $\quad z_i \leftarrow (sum - prod) \bmod m_i$
14: **end for**

Chapter 3

CUDA

Graphical Processing Units, also known as GPUs, are highly parallel specialized processors typically used in the real-time rendering of 3D content. However, the continued increase of their processing power and computational flexibility has drawn attention to their use outside the realm of 3D graphics, for other computational purposes. The main reason for this substantial increase in computation power comes from the GPU's special-purpose design — highly data-parallel operations on vertexes and textures. This comes at a significant cost: flow control and fast memory access is much less optimized than in a general-purpose CPU.

General purpose computing on the GPU was originally done by treating the input data as a texture and processing it using pixel and vertex shaders. However, doing so directly is not only inconvenient, but also requires expertise in the inner workings of the modern 3D rendering pipeline. The tools available for shader development are cumbersome for general purpose use and little is known about the underlying architecture on which the code will run. This makes it particularly hard to engage and take advantage of the huge computational power these devices offer.

In late 2006, NVIDIA introduced CUDA, a programming environment designed to give applications access to the GPU's computing power. With CUDA, the GPU is viewed as a highly multithreaded processor, operating the same program (in each thread) independently on different data. A program that runs on the GPU is called a *kernel*. When a *kernel* function is called using CUDA, it is necessary to specify how many threads will run the function. Threads are organized in *thread blocks*: groups of threads that have a common *shared memory*. Such memory can be used to communicate between threads in order to achieve cooperation. Each *kernel* can have several thread blocks; the set of all threads in all thread blocks of a kernel is called a *grid*. Thus, a *grid* is the instantiation of a *kernel* in the GPU. There can

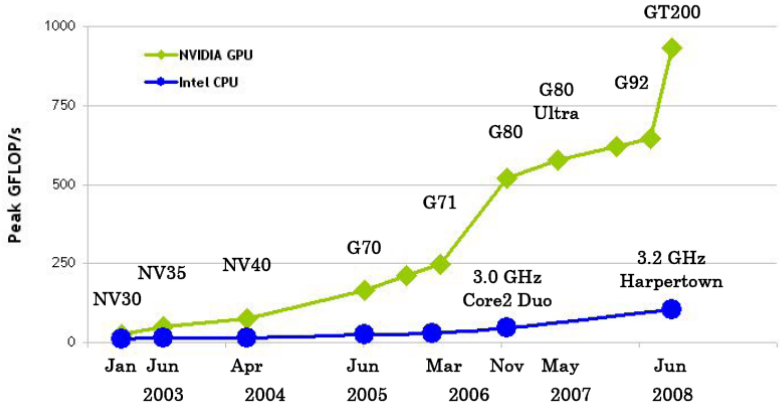

Figure 3.1: Evolution of computing power in FLOPS for recent CPUs and GPUs.

be more than one grid running in the GPU at any given time.

CUDA extends the C programming language in 4 main ways:

- *Function type qualifiers* to specify where a function shall be executed, and how it can be called.
- *Variable type qualifiers* to indicate where a variable shall be stored.
- A directive that allows the programmer to indicate *how a kernel will be executed* on the device.
- *Built-in variables* that identify each thread and block, plus dimensions of block, grid and warp.

In the following sections each of these extensions will be detailed and explained.

3.1 Function Types

There are 3 main types of functions in CUDA: *host functions*, *device functions* and *global functions*.

Host functions, defined by prefixing them with the *__host__* directive, are executed exclusively on the CPU. This is the default type for a function, in case the prefix is omitted.

Figure 3.2: Thread organization within a CUDA kernel.

A *global function*, also known as *kernel*, is a function executed on the GPU but accessible (callable) from the CPU. One should notice that this is precisely the kernel comprised of multiple threads we defined before. A global function is created by using the prefix *__global__* in its declaration.

Device functions are executed on the GPU, but also only callable from the GPU. This means only kernels or other device functions are allowed to call them. They are defined by the prefix *__device__*.

3.2 Variable Types

As with functions, CUDA allows the user to define where variables are stored, and how should they be accessed. There are 3 directives available for this purpose: *__device__*, *__constant__* *and* *__shared__*.

The *__device__* directive informs the compiler that the variable is to be stored in the GPU's global memory. If no other directive is used to specify the location where the variable is to be stored, it will be accessible by all threads in a grid and has the lifetime of the application.

If the variable is known to remain constant throughout the lifetime of the

application, one might use the *__constant__* directive to offload the variable to the GPU's constant memory. This in turn makes memory accesses faster than in global memory.

If there is a need to share information between threads within a block, the *__shared__* directive must be used. It makes the data visible by all threads in the same block and resides in the memory space of the thread block, making accesses faster than global memory. Writes to a shared variable will only be visible by the other threads in the block once they are synchronized (using the __syncthreads() special function).

3.3 Calling a Kernel

Global functions, when called, require several key parameters to be specified:

- The size of the grid, i.e. the number of thread blocks to be run in the kernel. This is a bidimensional value. Refer to 3 where we have a 2×3 grid.
- The dimension of each thread block in number of threads. This is a tridimensional value.
- An optional size of the shared memory to be dynamically allocated for each thread block on top of the statically allocated. The default value is 0.
- An optional stream to be associated to the kernel. Useful when keeping several computations active simultaneously.

The syntax for calling a global function from the host is *FuncName<<<Dg, Db, Ns, S>>>(Arguments)*, where *Dg* is the grid dimension, *Db* is the block dimension, *Ns* the dynamically allocated shared memory and *S* the associated stream.

3.4 Built-In Variables

To enable individual threads' programming without having to write the code for each of them individually, CUDA provides built-in variables that can be accessed *only* inside kernels. They allow to identify and locate each thread so that it can play its correct role in the computation. They are:

- gridDim — Contains the dimensions of the grid, in thread blocks; up to 2 dimensions are possible.

- blockIdx — This variable, composed of 3 dimensions (x, y, z) contains the block index within the current grid.
- blockDim — This variable, also tridimensional, contains the size of the current thread block.
- threadIdx — This variable contains the thread index within the block.

3.5 An Example

We now presented an example of a CUDA application that squares an array.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h> // Include CUDA API Functions

// Kernel that executes on the CUDA device - x = x^2
__global__ void multiply_array(float *x, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < N) x[idx] = x[idx] * x[idx];
}

// This is executed on the host
int main(void)
{
  float *x_h, *x_d;  // Pointer to host & device array
  const int N = 10;  // Number of elements in array
  size_t size = N * sizeof(float);

  x_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **)&x_d, size);    // Allocate array on device

  // Initialize host array and copy to CUDA device
  for (int i=0; i<N; i++) x_h[i] = (float)i;
  cudaMemcpy(x_d, x_h, size, cudaMemcpyHostToDevice);

  // Do the actual computation on the device
  int block_size = 4; // One can also use integers, they'll
                      // be overloaded into the correct type
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  multiply_array<<<n_blocks, block_size>>>(x_d, N);

  // Retrieve result from device and store it in host array
  cudaMemcpy(x_h, x_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  // Print results
```

```
  for (int i=0; i<N; i++) printf("%d %f\n", i, x_h[i]);

  // Cleanup
  free(x_h);
  cudaFree(x_d);

  // Exit
  return 0;
}
```

Listing 3.1: Example CUDA application

The structure of a typical CUDA application can be easily derived from the example. First, obtain some data to be processed; in this example the data is artificially generated. Then, copy the data to the GPU, using the `cudaMemcpy` function. The memory used in the device must be previously allocated using the `cudaMalloc` function. After this, the kernel can be called with a configurable number of thread blocks and threads per block. These sizes need not be constant or defined at compile time. After the kernel is done, copy the data back to the host using once again the `cudaMemcpy` function.

3.6 The GT200 Architecture

NVIDIA's current architecture, GT200, is a natural evolution of the previous architectures G80 and G92. [51] gives a thorough coverage of the G80 architecture. The hardware architecture of the G80 matches quite well the CUDA programming model described in Section 3: the computing part of the card is seen as an array of *streaming multiprocessors (SM)*. Early G80 GPUs were composed of 16 SMs; newer GT200 models have up to 30.

Each SM contains its own shared memory and register bank and also its own constant and texture memory cache. Besides these specialized fast memories, the GPU also has access to local and global memory, which reside outside the chip and are not cached. Additionaly, each SM contains a single instruction cache, 8 ALU units and 2 Special Function Units (SFU) — to maximize the ALU area on the chip, each of these ALUs operates in a SIMD fashion, in groups of 32 threads called *warps* controlled by a single instruction sequencer. At each cycle, the SM thread scheduler chooses a warp to be executed. Since each of the 8 ALUs supports up to 128 concurrent thread contexts, i.e. each ALU can be aware of up to 128 concurrent threads operating in it, it is possible to have $8 \times 128 = 1024$ concurrent threads executing on a single SM — in a 30-SM GPU, this amounts to up to 30720 simultaneous threads being executed at any given time.

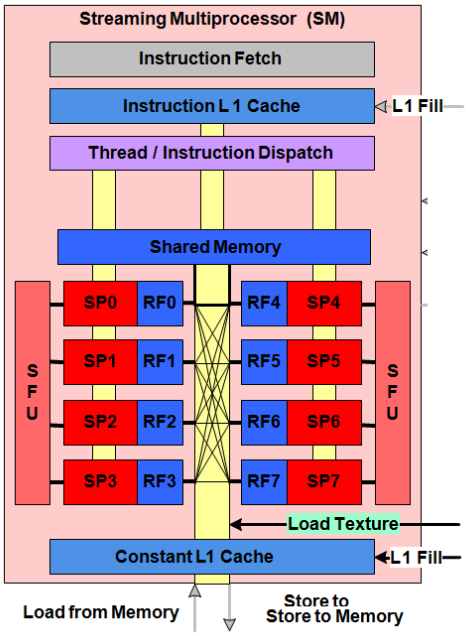

Figure 3.3: The streaming multiprocessor, building block of the GT200 architecture.

Each ALU unit can compute simple arithmetic instructions, be it integer, logical or single precision floating point, per cycle. Moreover, each ALU can compute a MAD (multiply-and-add) operation per cycle. Each SFU unit can compute transcendental functions (e.g. sin, cos), and contains 4 floating point multipliers. Thus, an SM can compute $8 \times 2 + 4 \times 2 = 24$ floating-point operations per cycle — a 30-SM GPU at 1476 MHz can (theoretically) compute $24 \times 30 \times 1476000000 = 1062720000000$ floating-point operations per second, a little over 1 Tflop/s.

The GT200 architecture introduced one double precision floating point unit per SM, doubled the register number (16384 32-bit registers as opposed to 8192 in the G80 architecture) and increased the memory bandwidth and SM amount. The double precision performance, however, is far from being on par with the single precision counterpart — in the same 30-SM GPU at 1476 MHz, the peak throughput is $2 \times 30 \times 1476000000 = 88560000000$, a mere 88 Gflop/s. In comparison, the PowerXCell 8i processor has a double-precision performance of 102.4 Gflop/s with 8 cores [2]; on the other hand, the Intel Core i7-965 has 51.2 Gflop/s with 4 cores [3].

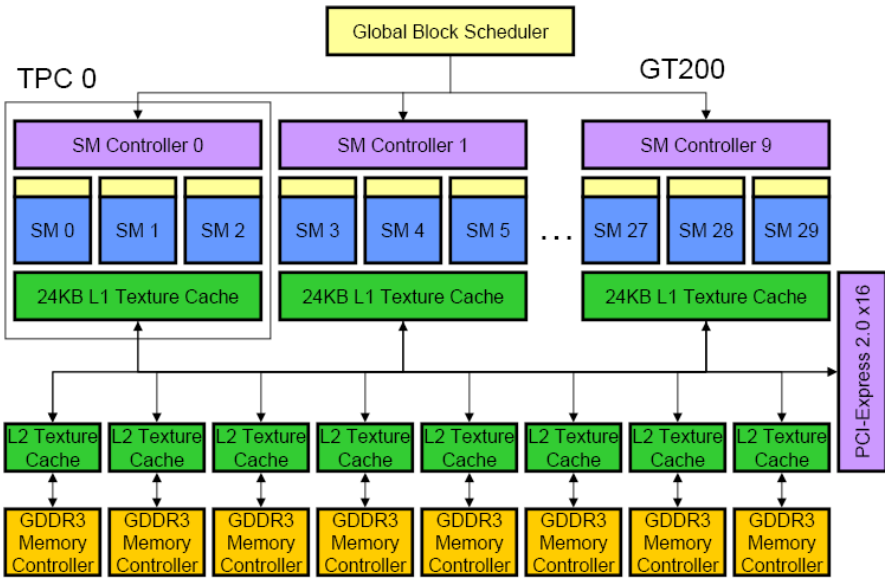

Figure 3.4: The GT200 architecture — a scalable array of SMs

Chapter 4

Library

4.1 Objectives

One of the objectives of this project, as stated in Section 1.2, was to develop a library that made use of the algorithms studied and developed. More precisely, we intend to enable developers to use the GPU's computing power to perform cryptographic tasks. Furthermore, the library must be easy to use and integrate with other applications that already use cryptography.

4.2 Requirements

The library will be used by programmers who already deal with cryptography in their software, be it secure servers, file encryption applications, certificate handling, etc. To be easily integrated in already existent software, it should be easy to replace current cryptographic libraries with ours. The most widespread of these cryptographic libraries is OpenSSL — this makes it a good candidate to start with.

Some use cases of this library are:

- Fast encryption of long streams of data, e.g. files, backups, video, etc.
- Multiple small message encryption and decryption, e.g. an SSL web server.
- Batch public key operations, e.g. SSL handshake handling.

4.3 Design

Since ease of use and integration was one of the design goals, it would seem a good idea to integrate the developed cryptographic code into an OpenSSL 'ENGINE', which is a mechanism the library uses to extend its support to external cryptographic hardware or alternative implementations [78]. However, OpenSSL does not provide any type of batching or even asynchronous functions; this would render the throughput advantages of GPU computing moot. Thus, we opted to create an external library that provided easy batching to users.

However, to retain some compatibility and to avoid 'reinventing the wheel', we employ the data structures (this includes the BIGNUM, RSA, DH, DSA and AES_KEY structures) already existent in OpenSSL. Also, the bookkeeping and other miscellaneous arithmetic operations, such as the final CRT multiplication and addition in RSA decryption, are performed by calls to OpenSSL.

The chosen programming language was ANSI C [7]. This relates to the fact that OpenSSL is developed in ANSI C and that the CUDA compiler is also (officially) ANSI C. However, the CUDA compiler's frontend is in reality C++. Thus, to avoid redundancy in some kernels and enable the compiler to perform some optimizations that would be hard to do otherwise, we employed one particular feature of C++ — function templates. These allow to define different functions for different parameter sets during compile-time, which can be very useful in a resource-scarce architecture like GT200. Nevertheless, all exported functions are compatible with ANSI C by the usage of the `extern "C"` directive.

Error handling is performed by return codes — it would be unwise to kill a whole server process when a request cannot be processed. Errors related to the library have negative return codes; errors related to the CUDA runtime have positive return codes. Successful functions return 0.

Temporary memory used to communicate between the GPU and the host is allocated as *non-pageable memory*. This enables DMA transfers do be performed between the GPU's global memory and the host's RAM, speeding up such transfers considerably [63]. However, allocating too much non-pageable memory can be harmful to a system; by default, no more than 16 MB are allocated.

4.4 Functionality

This section will list and succinctly explain the various functions exported by the library. These are just the functions exported to the user — the inner workings of each are detailed in Chapter 5.

4.4.1 Symmetric encryption primitives

The following functions perform symmetric-key encryption by the AES or Salsa20 ciphers described in Section 2.2.

```
int cudaAES_set_key(AES_KEY *aeskey, u8 *key, int bits, int enc);
```

This function takes as input an array of bytes, *key*, and depending on the value of the input *bits* derives 10, 12 or 14 round keys for 128, 192 and 256-bit keys respectively. Any other values of *bits* are not accepted.

If the input value *enc* is different from 0, the round keys generated are for the encryption process. If *enc* is set to 0, decryption round keys are generated.

The resulting round keys are stored in *aeskey*, a structure defined in OpenSSL's `aes.h` header file.

```
int cudaAES_ecb_encrypt(u8 *in, u8 *out, AES_KEY *aeskey, u32 len,
int enc);
```

The above functions respectively encrypt and decrypt a stream *in* of *len* bytes, where *len* is a multiple of `AES_BLOCK_SIZE` bytes. It is required that *aeskey* be initialized with `cudaAES_set_key` before calling either function.

The output is stored in *out*; *in* and *out* can be the same.

```
int cudaAES_ctr128_encrypt(u8 *in, u8 *out, u32 length, const
AES_KEY *key, u8 *iv, u8 *rem, u32 *num);
int cudaAES_ctr128_decrypt(u8 *in, u8 *out, u32 length, const
AES_KEY *key, u8 *iv, u8 *rem, u32 *num);
```

The `cudaAES_ctr_encrypt` function encrypts a long stream of *length* bytes, pointed to by *in*, in the CTR mode. The *rem* array might contain a partially used block from a previous run; the index of the last used byte is pointed to by *num*. It is required that *aeskey* be initialized with `cudaAES_set_key`.

When the function terminates successfully, *rem* contains the last generated block and *num* contains $16 - (length \bmod 16)$, i.e. the remaining usable bytes for a subsequent encryption. The output is stored in *out*; *in* and *out* can be the same.

Given the symmetry of the CTR mode, the encryption and decryption process is the same. Thus, `cudaAES_ctr_decrypt` is simply an alias for `cudaAES_ctr_encrypt`.

```
int cudaAES_cbc_decrypt(u8 *in, u8 *out, u8 *iv, AES_KEY *aeskey,
u32 len);
```

The `cudaAES_cbc_decrypt` function performs a decryption of a long CBC-encrypted stream. The absence of an encryption counterpart is not a mistake — CBC encryption cannot be parallelized.

The input is a stream *in* of *len* bytes, where *len* is a multiple of 16. The output is stored in *out*, which can overlap with *in*.

```
int cudaAES128_ecb_encrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
const u32 length, const u32 nmsg);
int cudaAES192_ecb_encrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
const u32 length, const u32 nmsg);
int cudaAES256_ecb_encrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
const u32 length, const u32 nmsg);
int cudaAES128_ecb_decrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
const u32 length, const u32 nmsg);
int cudaAES192_ecb_decrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
const u32 length, const u32 nmsg);
int cudaAES256_ecb_decrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
const u32 length, const u32 nmsg);
int cudaAES128_ctr128_encrypt_batch(u8 **in, u8 **out, AES_KEY
*keys, u8 **iv, const u32 length, const u32 nmsg);
int cudaAES192_ctr128_encrypt_batch(u8 **in, u8 **out, AES_KEY
*keys, u8 **iv, const u32 length, const u32 nmsg);
int cudaAES256_ctr128_encrypt_batch(u8 **in, u8 **out, AES_KEY
*keys, u8 **iv, const u32 length, const u32 nmsg);
int cudaAES128_ctr128_decrypt_batch(u8 **in, u8 **out, AES_KEY
*keys, u8 **iv, const u32 length, const u32 nmsg);
int cudaAES192_cbc_encrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
u8 **iv, const u32 length, const u32 nmsg);
int cudaAES256_cbc_decrypt_batch(u8 **in, u8 **out, AES_KEY *keys,
u8 **iv, const u32 length, const u32 nmsg);
```

The above functions are functional equivalent to the long stream versions. However, CTR mode no longer keeps track of usable bytes, i.e. it assumes each message is independent. The added parameter, *nmsg*, defines the amount of messages to encrypt/decrypt at once. Thus, *in* and *out* and pointers to *nmsg* small streams of *length* bytes, while *keys* is a pointer to an array of *nmsg* `AES_KEY` structures.

```
int cudaSalsa20_set_key(SALSA_KEY *key, u8 *key_bytes, u32 bits);
int cudaSalsa20_set_iv(SALSA_KEY *key, u8 *iv);
```

These functions respectively set the key and IV in a `SALSA_KEY` structure. The parameter *bits* defines the key length — accepted values are 128 and 256. The IV is constant, set at 8 bytes long.

```
int cudaSalsa20_encrypt(u8 *in, u8 *out, const SALSA_KEY *skey,
u32 len);
int cudaSalsa20_decrypt(u8 *in, u8 *out, const SALSA_KEY *skey,
u32 len);
```

The `cudaSalsa20_encrypt` function encrypts a stream *in*, with *len* bytes, into an encrypted stream *out* using the key *skey*. *skey* must be previously initialized with the functions `cudaSalsa20_set_key` and `cudaSalsa20_set_iv`.

Much like the CTR mode of operation in block cipher, the encryption process is the same as the decryption; once again, `cudaSalsa20_decrypt` is simply an alias for `cudaSalsa20_encrypt`.

4.4.2 Asymmetric cryptographic primitives

This section describes the public-key functions implemented, described in Section 2.3.

```
int RSA_generate_key(RSA *rsa, int bits);
```

`RSA_generate_key` generates an RSA keypair and stores it in *rsa*. The public modulus' size, i.e. *bits*, can be 1024 or 2048 bits. The public exponent used is 65537.

```
int cudaRSA1024_public_encrypt(u8 **from, u8 **to, RSA **rsa,
int batch);
```

```
   int cudaRSA1024_private_decrypt(u8 **from, u8 **to, RSA **rsa,
int f4, int batch);
   int cudaRSA2048_public_encrypt(u8 **from, u8 **to, RSA **rsa,
int batch);
   int cudaRSA2048_private_decrypt(u8 **from, u8 **to, RSA **rsa,
int f4, int batch);
```

The above functions perform public-key encryption and decryption. The input is a list of *batch* arrays of bytes, *in*. If the 1024-bit variants are used, the arrays in *in* and *out* are assumed to contain 128 bytes; the 2048-bit variants assume 256 byte inputs and outputs. The public and/or private key, stored in *rsa*, must match the appropriate size. As an example, using a 2048-bit key with `RSA1024_public_encrypt_batch` will cause an error. The, *f4* indicates if the public exponent is equal to 65537, in which case an optimized exponentiation method will be used instead.

The same functions can also be used to verify RSA signatures — simply put, if the inputs are message digests, one can perform an 'encryption' and verify whether the signature matches the digest.

```
   int cudaDH1024_generate_key(DH **dh, int batch);
   int cudaDH1024_compute_key(u8 **key, BIGNUM **pub_key, DH **dh,
int batch);
   int cudaDH2048_generate_key(DH **dh, int batch);
   int cudaDH2048_compute_key(u8 **key, BIGNUM **pub_key, DH **dh,
int batch);
```

`DH1024_generate_key_batch` and `DH2048_generate_key_batch` generate a new ephemeral key for the Diffie-Hellman key exchange. This consists of generating a secret x, and computing $y = g^x \pmod{p}$. The `DH` structures are assumed to already have been initialized with suitable primes and generators, such as the ones from [39].

`DH1024_compute_key_batch` and `DH2048_compute_key_batch` take as input a public-key from the third party performing the key exchange, *pub_key*, and computes a shared secret using both *pub_key* and *dh*, as described in Section 2.3.1. The resulting shared key is stored in *key*.

```
   int cudaDSA1024_sign(const u8 *dgst, u8 **sig, DSA **dsa,
int batch);
   int cudaDSA1024_verify(int *status, const u8 **dgst, u8 **sig,
DSA **dsa, int batch);
```

```
   int cudaDSA2048_sign(const u8 *dgst, u8 **sig, DSA **dsa,
int batch);
   int cudaDSA2048_verify(int *status, const u8 **dgst, u8 **sig,
DSA **dsa, int batch);
```

The above functions, as the name would imply, perform *batch* signatures and verifications simultaneously. The messages' digests, computed using SHA-1, is input in *dgst*; the signature in *sig*. If a signature is being performed, *dsa* must contain the secret exponent x. In the verification functions, the results of each signature are stored in the array *status* — 0 for invalid signature, 1 for valid signature.

4.5 Testing

In security, having fast functions is simply not enough: they have to be correct. In order to ensure the correctness of the functions described in Section 4.4, several small applications were created to verify that the outputs do indeed match the expected.

Each of these applications starts with a pseudorandom seed, harvested from an adequate randomness source, e.g. `/dev/urandom`. From this seed, various random parameters are generated, such as encryption keys, IVs, exponents, etc. Then the library functions are called with these inputs, and compared against OpenSSL's output, which is assumed to be correct. If an error is detected, the problem can be easily reproduced and debugged by reusing the seed which caused it.

While in symmetric cryptography testing the procedure is fairly straightforward, in public-key it is not as simple. In multiple precision and modular arithmetic there exist many corner-cases that are difficult to catch by simply performing arithmetic in random numbers. Thus, to test the modular arithmetic correctness, we performed exponentiations with numbers in the highest and lowest ranges for each of the numbers. Results were then verified step by step, and were compared against the correct results obtained using the MAGMA algebra system [20]. This helped uncover some arithmetic flaws, which were promptly corrected.

Chapter 5

Implementation Details

During the course of this project, research was done on cryptographic primitives and their implementation. In this chapter, we present the techniques and results on the GPU implementation of the algorithms introduced in Section 2.

Optimization of massively parallel GPUs is quite different from usual performance guidelines for CPUs. Whereas in CPUs most care goes into avoiding pipeline stalls and keeping memory accesses cached, GPU programs have other properties that can be exploited to increase the overall throughput of a kernel. [73] defines 4 main guidelines for the optimization of CUDA kernels:

1. *Hide memory latency by raising occupancy.* The G80 and GT200 GPU architectures allow up to 768 and 1024 simultaneous threads per execution unit, respectively. Global memory accesses are often very slow, ranging from 200 to 600 cycles. Thus, by having many concurrent threads active, one can avoid execution stalls due to slow memory accesses.

2. *Make use of on-chip fast memory.* Current NVIDIA GPUs contain fast on-chip memory, either in the form of registers or shared-memory. Using this memory instead of global memory not only speeds up accesses but reduces the bandwidth needs of the kernel.

3. *Avoid thread divergence.* The G80 and GT200 hardware groups threads in groups of 32 and executes them in a SIMD-like fashion. Whenever one of the threads diverges from the execution path of the others, the hardware serializes the execution of the threads until the divergence is over. This, naturally, creates a large performance penalty that should be avoided.

4. *Avoid inter-block communication.* CUDA provides intra-block synchronization and communication through shared-memory and the __syncthreads function. Whenever inter-block communication is required, this can either be done by atomic functions or by decomposing the kernel into multiple kernel calls with different parameters; either way, this can slow down a kernel considerably.

Throughout this chapter we'll be referring to these principles whenever a decision regarding efficiency has to be made. This is particularly important for bandwidth-bound kernels, as the ones in the following section.

5.1 Symmetric Cryptography

There are two different use cases under the symmetric cryptography category: long message encryption and multiple short packet encryption. The former is used e.g. to encrypt files, among other common uses. The latter is widely used in network communications, such as IPSec, SSL, or other encrypted protocols in use.

In the case of block ciphers, such as the AES, long message encryption in the GPU is only practical when the mode of operation of the cipher allows its parallelization. Such modes are e.g. ECB and CTR; the popular CBC mode is not parallelizable (in encryption), rendering long message encryption in this mode in the GPU not practical [19].

5.1.1 AES

The implementation of the AES cipher in this project was based on the one found in the OpenSSL library [78]. One common optimization done is to combine the mixing steps of each round and transform them into table lookups, yielding four 1024 byte tables (4KB). This way, the encryption process is transformed into a series of XORs and table lookups, simplifying and speeding up significantly the cipher in most architectures [28] [17].

In our parallel implementation, each GPU thread is responsible for the encryption of a 128-bit block. Alternatively, it would be possible to divide each block encryption into 4 threads, as described by Manavski in [52]. However, this is not of much practical advantage, since 128-bit blocks can too be loaded with a single memory access per thread, resulting in coalesced memory accesses across threads [63]. Thus, as recommended by guideline 1 of Section 5 we opted to perform more computations per thread, allowing more blocks to be simultaneously processed.

Type of Memory	Peak Throughput (MB/s)
Global	246.174
Constant	519.887
Shared	2872.080
Texture	1152.804

Table 5.1: Peak throughput of AES in CTR mode and a 128-bit key for each memory type available to the GPU.

The main challenge when implementing fast AES in the GPU is the lookup table storage. In order to store the lookup tables, we could choose between the GPU's global memory, constant memory, texture memory and shared memory. Given that the encryption process is mostly bounded by table access speed, it was critical that the fastest option be chosen here. Since the table lookup indexes are pretty much random, we can't rely on contiguous access speed. This rules out global memory (latency of around 400-600 cycles, can be amortized if several threads access contiguous memory addresses), constant memory (cached, as fast as reading a register if all threads in a half-warp read the same address) and texture memory (cached, optimized for addresses that are close together). The best option seems to be shared memory, since it has a relatively high number of 16 ports (or as NVIDIA calls them, banks) and is as fast as accessing registers as long as there are no bank conflicts between groups of 16 threads. Experimental measurements of conflicts showed that each shared memory access had in average 5.4 conflicts; constant memory had almost always 16 conflicts.

To conclude which table storage approach was the best, we performed several encryptions with AES in the CTR mode and a 128-bit key on a 256MB dataset. Table 5.1 presents the results that corroborated the above hypothesis.

Storing lookup tables in shared memory has a disadvantage, though — each thread block must be large enough so as to amortize the overhead of copying the tables to shared memory. Thus, we used a block size of 256, which effectively turns the table creation into 4 extra instructions per thread, a rather small overhead. Also, to avoid repeating this process multiple times, the same threads are reused to perform the whole encryption (up to the GPU's global memory size). When a thread finishes encrypting a block, it encrypts the block corresponding to the thread index plus the total number of threads running, which is constant. This total number is chosen in a way so as to maximize the GPU's execution unit occupancy, i.e. prevent that execution units stop waiting for a memory access or a pipeline stall. The

chosen number was the number of SMs in the GPU — a lower number would leave SMs unused, a higher number would not improve performance, as all SMs would be already filled with threads.

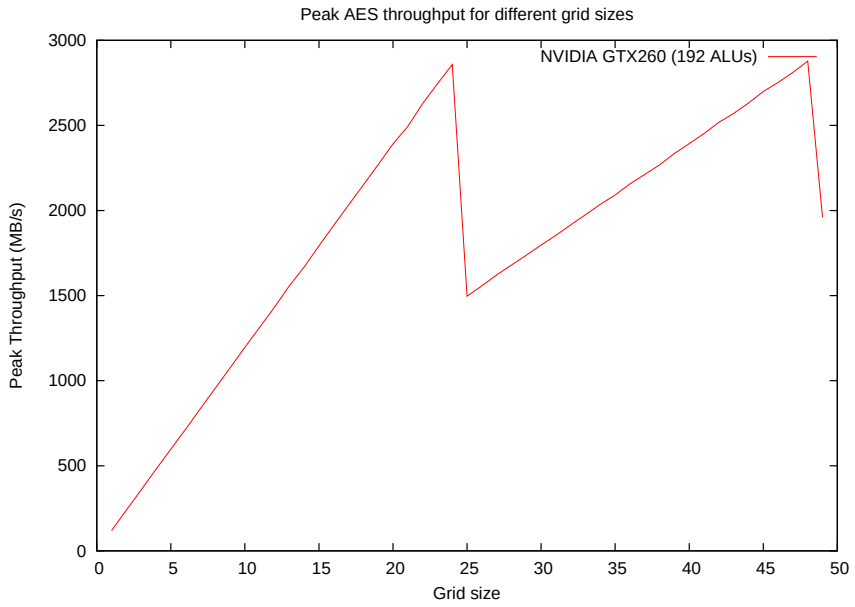

Figure 5.1: AES performance for various grid sizes.

In order to confirm this hypothesis, we performed a series of throughput measures with grid sizes ranging between 1 and 50, using the NVIDIA GTX260 with 24 SMs (refer to Section 3.6). Figure 5.1 shows the results. One can observe that the best throughputs are obtained when the grid size is equal or a multiple of the number of SMs in the GPU. Since there seems to be no advantage of having a multiple of this number as grid size, we choose to simply set the grid size to the number of SMs.

5.1.2 Salsa20

Our implementation of this stream cipher was based on the specifications and reference implementation provided by its author [15]. The implemented variant was the 12-round Salsa20 recommended by the eSTREAM committee, also called Salsa20/12 [8]. Given that as described in Section 2.2.2 this cipher works on 64-byte blocks, we can use the same techniques used in AES to parallelize Salsa20. While it would still be possible, not unlike AES, to separate a single block's execution into 4 threads, there are no advantages in

doing so: it would require too many synchronizations. This cipher simplifies our task, though, since it does not have lookup tables. Thus, our main objective is to keep the GPU's execution units busy during encryption. Once again, one thread per 512-bit block was the method employed. Reusing the same threads for further encryption was also implemented. The thread block size used was the same as AES's, 256.

5.2 Asymmetric Cryptography

5.2.1 Modular Exponentiation

Modular exponentiation, as remarked in Section 2.3, is the most important operation in the vast majority of public-key algorithms. Thus, our attention on our implementation focused on speeding this operation up.

The serial case

Our starting point was Algorithm 17 and 12 for multiplication and modular reduction respectively. These cost $2n^2+n$ single precision multiplications and $2n$ space, for n digit numbers. This approach was called *Separated Operand Scanning* (SOS) in [50]. The same paper describes some other approaches for computing Montgomery multiplications. We chose the *Coarsely Integrated Operand Scanning* method instead, since it takes $n + 2$ space and the same number of single-precision multiplications. The same method also seems to perform better in all tests, since it also has the least number of additions and memory accesses.

Barrett modular reduction was considered, but rejected. While it requires the same type of precomputation as Montgomery's algorithm, it is non-trivial to perform this type of reduction with less than $2n$ space and not clear whether it would present any performance advantage at all. We chose, then, the Montgomery method instead.

Algorithm 21 still needs a final subtraction, analogous to Step 4 in Algorithm 17. Results by Colin Walter and later by Hachez and Quisquater show that as long as $m < \beta^{n-1}$, for $\beta \geq 4$, such step is not necessary [80] [37]. The reported performance hit (due to the extra digit needed) indicates a slowdown ranging from 2.1% to 8.5% for cryptographically significant sizes of m. In light of the hardware being employed to perform these multiplications, it is of advantage to remove the unpredictable final subtractions, since they not only add complexity to the implementation but also cause divergence across threads, possibly limiting the throughput.

Algorithm 21 Montgomery Multiplication — Coarsely Integrated Operand Scanning

Require: $x, y, m, m' = m^{-1} \bmod \beta$
Ensure: $z = xy \bmod m$
1: $t \leftarrow 0$
2: **for** $i = 0$ to $i = n - 1$ **do**
3: $\quad t \leftarrow t + xy_i$
4: $\quad \mu \leftarrow m't_0 \bmod \beta$
5: $\quad t \leftarrow t + \mu m$
6: $\quad t \leftarrow t/\beta$
7: **end for**
8: $z \leftarrow t$
9: **return** z

Another significant improvement that can be made is to notice that, in an exponentiation, at least half of the multiplications are actual squarings (cf. Algorithm 14). In a squaring, due to the symmetry of the operands, many of the partial products are equal. Consider a 2-digit integer a to be squared. The classic method to square a is presented as follows:

$$\begin{array}{cccc} & & a_1 & a_0 \\ & \times & a_1 & a_0 \\ \hline & & a_0a_1 & a_0a_0 \\ & a_1a_1 & a_1a_0 & \\ \hline z_3 & z_2 & z_1 & z_0 \end{array}$$

One can observe that $a_0a_1 = a_1a_0$, rendering one of the 4 total multiplications unnecessary. One can also also notice that the diagonal partial products a_ia_i are always unique and cannot be reduced. All the other subproducts, however, can be reduced in half. Thus, we can compute a squaring as described in Equation 5.1, in about $(n^2 + n)/2$ operations.

$$(\sum_{i=0}^{n-1} x_i\beta^i)^2 = \sum_{i=0}^{n-1} \beta^{2i}x_i^2 + 2\sum_{j=i+1}^{n-1} \beta^{i+j}x_ix_j \tag{5.1}$$

Practical tests showed a reduction of at least 12% in time of exponentiation when using the optimized squaring code.

In order to minimize the amount of multiplications during exponentiation, we employed Algorithm 15 with a window size of 4. Measurements showed a reduction of 20-25% in time of exponentiation compared to Algorithm 14.

Another critical aspect of our implementation pertains to the location of the various numbers involved. The temporary variable t (cf. Algorithm 21), where temporary multiplications are performed is stored in shared memory, due to its fast access. Temporary variables used in the exponentiation are stored in local memory, where all accesses are always coalesced [63]. When the same modulus is used for a large number of computations, it is stored in constant memory. The exponent can be different for each thread; although it may inflict a small performance loss, the cost of having divergent threads is less than the cost of computing every multiplication and subsequently deciding whether to store the value or not through, e.g., logical operations. When every exponent is the same across a warp, no penalty exists. Using local memory for often used variables also allows the compiler to map such accesses to registers (mostly by loop unrolling), considerably speeding up the exponentiations.

The parallel case

The implementation described in the previous section is meant to be used for large load scenarios, where a large number of public-key operations has to be performed simultaneously. For smaller loads a faster approach (per operation) would be preferred. One of the approaches we used was to parallelize the underlying arithmetic, i.e. multiplication and addition/subtraction. The crux of this method lies in the observation that addition can be made in essentially constant time, given a large enough base β.

Consider the following numbers $a = 9169766787456036090883820023\text{-}0623439402$ and $b = 338298265791763513998413517820620220116$. Using the classic representation of Section 2.5.1 and a base $\beta = 2^{32}$, we have:

$$a = 4238731818 + 2^{32}796305440 + 2^{64}2486365495 + 2^{96}1157387284$$
$$b = 4157908692 + 2^{32}1366941397 + 2^{64}4273596754 + 2^{96}4269924418$$

Adding a to b, using Algorithm 10, yields:

$$a + b = 4101673214 + 2^{32}2163246837 + 2^{64}2464994953 + 2^{96}1132344406$$
$$c = 0 + 2^{32}1 + 2^{64}0 + 2^{96}1 + 2^{128}1$$

Now it remains to add the carries, c, to the partial sum $a + b$. Notice, however, that the digits of c are composed of only either 1 or 0; further carries will only be produced if one of the digits of $a + b$ is *equal* to $\beta - 1$. Thus, for arbitrary inputs this will only happen with very low probability.

From the above observation one can devise a parallel method to add numbers — add each digit individually in parallel, add the previous digit's carry and finally (with very low probability of happening) add whatever carries are left. Algorithm 22 describes this method.

Algorithm 22 Parallel addition

Require: x, y
Ensure: $z = x + y$
1: $j \leftarrow 2$
2: **for** each node i in parallel **do**
3: $z_i, c \leftarrow x_i + y_i$
4: $z_{i+1}, c \leftarrow z_i + c$
5: **while** $c \neq 0$ for any node i **do**
6: $z_{i+j}, c \leftarrow z_{i+j-1} + c$
7: $j \leftarrow j + 1$
8: **end while**
9: **end for**
10: **return** z

One can observe that, for a base β and after Step 4, the likelihood of a digit having again a carry out is less than $n\beta^{-1}$. For commonly used bases, such as 2^{32} or 2^{64}, this gives us a constant-time addition for most inputs. Furthermore, with constant-time addition, one can reduce the average complexity of Algorithm 12 to $O(n)$, by computing all partial products of the inner loop in parallel and adding the resulting carries in $O(1)$. This method, along with Algorithm 21 seems to be very similar to the one described in [31] for the MasPar parallel computer. The drawback of this method is the many synchronizations needed; we addressed this issue by employing the vote functions provided by the NVIDIA GT200 architecture, which perform the testing for carries in an entire warp in a very efficient manner. This proved to be particularly advantageous for 512 and 1024 bit integers, which fit entirely inside a warp, not needing explicit __syncthreads() calls.

Another drawback of this method is that it is fairly simple to mount a denial of service attack against a server using this implementation. By ensuring that most, or all, additions take $O(\log n)$ time instead of $O(1)$, the throughput would be severely hit. There is no good solution for this, aside from just using another method for parallel arithmetic.

Operation	Kawamura	Bajard	ECRT	Improved ECRT
Multiplications	$2k^2 + 9k$	$2k^2 + 8k$	$k^2 + 3k$	$k^2 + 3k$
Divisions	0	0	k	0

Table 5.2: Operation counts for various RNS modular multiplication algorithms.

Residue Number Systems

As detailed in Section 2.5.5, numbers can be represented by their residues modulo a set of coprimes. Despite the limitations of classical residue number systems, modulo reduction can be performed by either the Explicit Chinese Remainder Theorem or by the analogous of Montgomery multiplication in these number systems. Table 5.2 summarizes the operation counts of the several approaches described in Section 2.5.5 for modular multiplication in such systems.

Provided that the divisions in the Explicit CRT approach can be eliminated or replaced by less costly multiplications, this method has the best operation count. Plus, it is considerably simpler to implement than the Montgomery approaches. Hence we chose to study and efficiently implement this method, instead of Montgomery's.

Two main approaches were considered to implement the single-precision modular arithmetic. The first was to use the native floating-point arithmetic of the GPU. It has the advantage of having the largest theoretical throughput in FLOPS, given the architecture described in Section 3.6. With this approach, the moduli are 12 bits long, so as to take up a full 24-bit product, the limit of single-precision floating point precision. Modulo reduction is performed by the simple expression described in Equation 2.8.

In this approach, Algorithm 20 is implemented directly, with 1 digit being handled per thread. Division, where applicable, is replaced by multiplication by the reciprocal, $1/p$, which is precomputed at the beginning of the kernel. This is the most suited approach to the GPU. However, with 12 bit moduli many primes are needed to represent cryptographically useful numbers — for a 1024 bit modulus at least 175 primes are needed, and 192 to have a warp-aligned kernel.

The second approach consists of using 32-bit moduli and integer arithmetic. In this scenario, the integer modular reductions and divisions are the bottleneck — NVIDIA simply says they are 'particularly costly and should be avoided' [63]. Thus, we use the observations of Section 2.5.4 to produce faster single-precision modular arithmetic.

The basis employed is, then, the set of primes from 2^{32} downwards, until there are enough primes to represent the desired number(s). Theorem 2 shows that there are enough such primes close to 2^{32} for most cryptographic uses. For instance, representing a 4096-bit integer requires 129 32-bit primes. By Theorem 2, and assuming $\log 2^{32} \approx \log(2^{32} - c)$, we have $c/\log 2^{32} = 129 \equiv c \approx 2861$[1]. Division can also be sped up in a similar way. However, we'll show that with the above basis, one can avoid divisions altogether.

Let u be a positive integer and P the product of k moduli p_i. [18, Theorem 2.1] states that $\alpha - round(\alpha) = u/P$, i.e. u/P must be less than $1/2$. One can, by adding a few bits of dynamic range, improve the guarantee that α is very close to a positive integer; in fact, it is possible to choose how close! Forcing $P > ut$, for some integer constant t, will guarantee that $\alpha - round(\alpha) < 1/t$.

Suppose that one chooses the moduli as described above, of the form $2^{32} - c_i$. The divisions are performed by fixed-point arithmetic — t_i/p_i is computed as $\lfloor 2^{32} t_i/p_i \rfloor$. Observe, now, that one can approximate $\lfloor 2^{32} t_i/p_i \rfloor$ as $\lfloor 2^{32} t_i/2^{32} \rfloor$ — since $2^{32} t_i < 2^{64}$, the maximum error of this approximation is $\lfloor (2^{64}-1)/(2^{32}-c_i) \rfloor - \lfloor (2^{64}-1)/2^{32} \rfloor = c_i + 1$. This means we can approximate α without performing any division with an error of at most $2^{-32} \sum_i^k c_i + 1$. Thus, we can compute α without divisions (or multiplications!) at all, as long as $u/P < 2^{-32} \sum_i^k c_i + 1$.

As a practical example, consider an arbitrary modulus m of 512 bits. [18] shows that ECRT modular multiplication requires that $P > 2(m \sum_i^k p_i)^2$. To represent this quantity, we can take the first 35 primes below 2^{32} — $\langle 2^{32} - 5, \ldots, 2^{32} - 959 \rangle$. The maximum error bound is $2^{-32} \sum_i^k c_i + 1 \approx 3.99 \times 10^{-6}$. The maximum possible m renders $u/P \approx 5.70 \times 10^{-7}$. Since $u/P < 2^{-32} \sum_i^k c_i + 1$, no changes to the basis are needed and divisions are avoided altogether.

5.2.2 RSA

Key Generation

As stated in Section 2.1.3, it is easy to make a compositeness test from a number of simple exponentiations. Thus, one can perform a primality test by exponentiating and comparing the result with 1. This turns out to be very useful in RSA key generation, where the bottleneck is searching for two primes that make up the modulus N.

Primes can be generated by selecting random numbers p and checking that $2^{p-1} \equiv 3^{p-1} \equiv 5^{p-1} \equiv 7^{p-1} \equiv 1 \pmod{p}$ — this is the Fermat primality test. By Theorem 2, a 1024-bit RSA key needs about $2 \times 4 \times \log 2^{512} \approx$

[1] The actual precise value is 2759.

Algorithm 23 Improved Explicit CRT Modular Multiplication

Require: $a_i = A \bmod m_i, b_i = B \bmod m_i, m_i, N$
Ensure: $z_i \equiv AB \pmod{m_i}$
1: Precompute $q_i = (M/m_i)^{-1} \bmod m_i, c_i = M \bmod p \bmod m_i, d_{ij} = M/m_i \bmod p \bmod m_j$
2: $t_i \leftarrow a_i b_i q_i$
3: $\alpha \leftarrow 0$
4: **for** i in 1 to r **do**
5: $\quad \alpha \leftarrow \alpha + t_i$ ▷ $\alpha = \alpha + t_i/2^{32}$
6: **end for**
7: $\alpha \leftarrow (\alpha + 2^{31})/2^{32}$ ▷ $\lfloor \alpha + 0.5 \rfloor$
8: **for** i in 1 to r **do**
9: $\quad sum \leftarrow 0$
10: $\quad$ **for** j in 1 to r **do**
11: $\quad\quad sum \leftarrow (sum + t_j d_{ij}) \bmod m_i$
12: $\quad$ **end for**
13: $\quad prod \leftarrow \alpha c_i \bmod m_i$
14: $\quad z_i \leftarrow (sum - prod) \bmod m_i$
15: **end for**

2800 512-bit exponentiations; a 2048-bit RSA key needs about 5600 1024-bit exponentiations. Thus, one can see that the results obtained in Section 5.2.1 directly apply to the key generation performance of RSA.

A stronger test, keeping the same complexity, is devised by noticing that the only square roots of 1 in a prime field are 1 and -1 [49, Chapter 5]. Thus, the square root of $a^{p-1}, a^{(p-1)/2}$ is either 1 or -1 $\pmod p$. Repeatedly checking whether this identity holds true for several different a was first proposed as a compositeness test by Solovay and Strassen in [75]. In this test, there does not exist a class of numbers that can 'fool' every base a, like the Carmichael numbers described in Section 2.1.3. Each iteration has a 1/2 chance of being erroneous — for k iterations of the Solovay-Strassen test, there's a $(1/2)^k$ chance of error [54].

By imposing certain restrictions on the structure of the prime being generated, one can devise an even stronger test for compositeness. First, define the number to be tested to be congruent to 3 (mod 4). This is not a very important restriction — asymptotically, half the primes are of this form[2]. To test for compositeness, simply check if $a^{p-1} \equiv \pm 1 \pmod p$, for arbitrary a. Koblitz shows that the probability of error of this test is $(1/4)^k$, for k trials

[2]It is easy to see that numbers of the form 0 (mod 4) are divisible by 4, and numbers of the form 2 mod 4 are divisible by 2.

Bits	Number of trials
512	6
1024	3
2048	2

Table 5.3: Minimum amount of Rabin-Miller trials to reduce probability of error to at least 2^{-80}.

[49, Chapter 5]. This test is a particular case of the more general *Miller-Rabin* primality test; it has the added advantage of requiring only 1 exponentiation per trial for the same error probability [57] [68].

The $(1/4)^k$ probability is worst-case; the average probability is much more favorable, as shown in [29]. Table 5.3 presents the number of iterations k needed to achieve an average probability of error of at least 2^{-80} for several relevant cryptographic ranges.

Algorithm 24 Prime Generation

Require: $n = \log_2 p$
Ensure: Prime p
1: $p \leftarrow randombits(n)$
2: $p \leftarrow p - (p \bmod 4)$
3: $p \leftarrow p + 3$ ▷ Ensure p is congruent to 3 (mod 4)
4: **repeat**
5: **repeat**
6: $p \leftarrow p + 4$
7: **until** $p \bmod \{2, 3, 5, \ldots, 257\} \neq 0$ ▷ Weed out small factors
8: $flag \leftarrow 0$
9: **for** $i = 1$ to k **do** ▷ k trials
10: $a = randombits(n)$
11: **if** $a^{(p-1)/2} \neq \pm 1 \pmod p$ **then**
12: $flag \leftarrow flag + 1$
13: **end if**
14: **end for**
15: **until** $flag = 0$
16: **return** p

Encryption and Signature Verification

Although the public exponent in RSA is selectable by a user generating keys, it often is selected with performance in mind. One very common exponent is the prime $2^{16} + 1 = 65537$; in fact, it is the default exponent used in OpenSSL's RSA key generation [78]. Computing this power is very efficient, given its low Hamming weight — it requires only 16 squarings and 1 multiplication.

Other common exponents, often used in low-power devices such as smartcards, are 3 and 17. Their low size and Hamming weight also allow for efficient exponentiations. However, there are some attacks that are possible for very low exponents that do not apply for larger ones [24]. NIST's guidelines regarding key sizes and exponents do not allow exponents smaller than 65537 [66]. Thus we decided not to support those smaller exponents explicitly. Still, all exponents different than 65537 are handled with the algorithms from Section 5.2.1.

Algorithm 25 RSA Encryption — Public Exponent $2^{16} + 1$

Require: $x, e = 2^{16} + 1, N$
Ensure: $z = x^e \bmod N$
1: $z \leftarrow x$
2: **for** $i = 0$ to 16 **do**
3: $\quad z \leftarrow x^2 \bmod N$
4: **end for**
5: **return** $zx \bmod N$

Decryption and Signature Generation

One particular case of RSA is decryption. Since decryption (or signing) needs the owner's private key, one can also store the original primes p and q that make up the public key. Based on this observation, Quisquater and Couvreur in [67] introduced a faster way to perform RSA decryptions than the one presented in Algorithm 4 — by performing the exponentiation modulo the two factors of the public modulus N and 'pasting' them together using the Theorem 4, one is replacing 1 n-bit exponentiation by 2 $n/2$-bit exponentiations and a couple of multiplications. Since the complexity of exponentiation using simple multiplication algorithms is $O(n^3)$, one can expect this CRT approach to be around 4 times faster. It also turns one exponentiation into two smaller parallel ones, which suits the GPU paradigm very well.

Algorithm 26 RSA CRT Decryption

Require: $x, e, p, q, dP = e^{-1} \bmod (p-1), dQ = e^{-1} \bmod (q-1), qInv = q^{-1} \bmod p$

Ensure: $z = x^e \bmod pq$

1: $z1 \leftarrow x^{dP} \bmod p$
2: $z2 \leftarrow x^{dQ} \bmod q$
3: $h \leftarrow qInv(z1 - z2) \bmod p$
4: $z = z2 + hq$
5: **return** z

5.2.3 Diffie-Hellman

Diffie-Hellman, in its classic form, relies on modular exponentiation alone as the sole operation; since the modulus is prime, we cannot use the Chinese Remainder Theorem to parallelize operations. Thus, we use the algorithms of Section 5.2.1 directly to implement it.

5.2.4 DSA

DSA, unlike both RSA and Diffie-Hellman, requires more operations than simple modular exponentiation. Its bottleneck, however, is still the modular exponentiation. For the signing process, described in Algorithm 8, we use the modular exponentiations from Section 5.2.1 and the remaining arithmetic is performed in the CPU.

The verification step presents a different challenge — computing the product of 2 modular exponentiations. There exist several ways to perform this computation. The solution chosen is the simplest: compute every exponent independently and subsequently multiply each pair together. This allows us to take advantage of the GPU's parallelism, not unlike the RSA decryption described in Section 5.2.2.

Chapter 6

Results

The chapter presents the results obtained when benchmarking the methods described in the previous chapter. Unless otherwise stated, the GPU employed was an NVIDIA GTX260 running at 1242 MHz, featuring 192 execution units distributed across 24 SMs. The CPU used for comparisons was an Intel Core 2 Duo E8400, running at 3.0 GHz and using a single core.

6.1 Symmetric Primitives

6.1.1 AES

Long message encryption

To determine the relative performance against a common CPU as well as the overhead of encrypting data on a GPU, we performed encryptions on data sets ranging from 1 KB to 256 MB. The algorithm was AES with a 128 bit key, each block encrypted independently (also known as ECB mode). Figure 6.1 shows the results obtained. We can see that 64 KB is roughly the threshold where the GPU starts to outperform the CPU in bulk AES encryption. Furthermore, these figures clearly show the bottleneck in GPU symmetrical encryption: PCIe bandwidth. Ignoring transfers between the host and GPU, we obtained peak throughputs of over 35 Gbit/s, matching the best published performances of dedicated AES VLSI circuits and outperforming the best current software implementations [41] [43]. Current PCIe bandwidth on the testing hardware is allegedly 4 GB/s; measurements revealed it was slightly lower than that, at 3.1 GB/s. Taking into account that 2 transfers are done to encrypt a block of data, one to send and one to receive, this figure is cut in half, explaining the results shown on Figure 6.1. Upcoming

PCIe revisions[1] will double the available bandwidth between the host and the GPU.

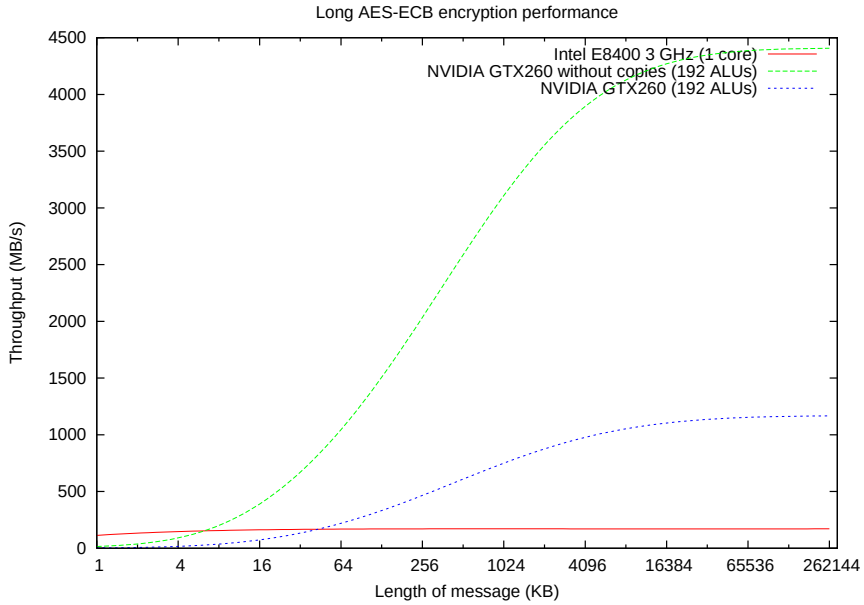

Figure 6.1: AES-ECB performance for various message sizes in a CPU and GPU implementation.

Multiple message encryption

In order to test the GPU's behavior when encrypting batches of small messages, we performed encryptions on large numbers of small messages with lengths ranging between 1 and 16 KB. Each message had its own allocated memory block, to simulate a real world scenario (i.e. L2 cache misses on the CPU side).

Figure 6.4 shows the performance obtained for a typical number of messages of varying sizes[2], with and without memory copies. Once again, copying between the host and the GPU is the bottleneck. In this case the penalty is heightened, since the copies are not done in bulk but individually, as each message has its own memory address. As such, it becomes hard to surpass the CPU's performance, unless an irrealistic number of messages is processed at once.

[1] `http://www.pcisig.com/news_room/08_08_07/`

[2] This number was chosen as the approximation of the total size of 1 second of requests in a busy SSL server.

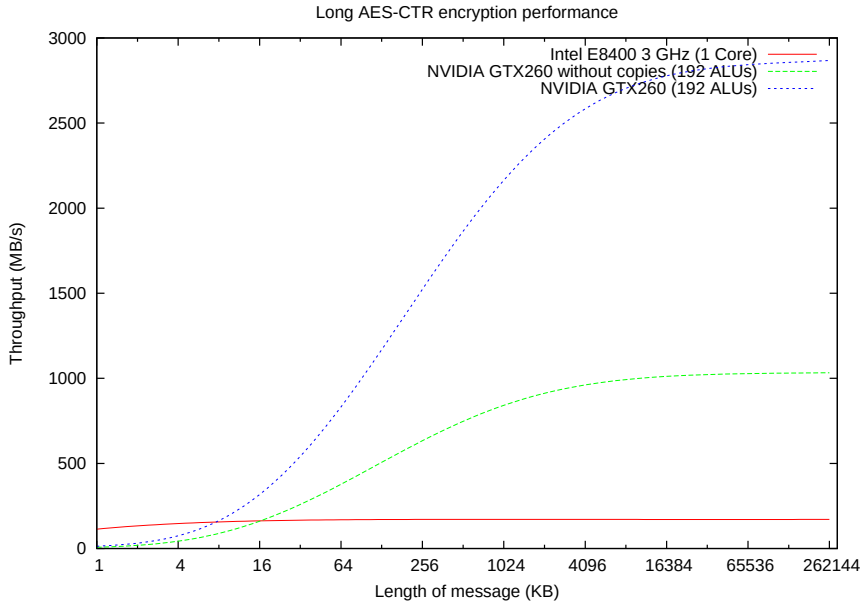

Figure 6.2: AES-CTR performance for various message sizes in a CPU and GPU implementation.

It is also interesting to notice the performance drop in higher message sizes, when the individual message length raises. One can see that starting at 1024 byte messages performance starts to drop dramatically. This seems to be related to the memory access patterns — memory reads across threads can be coalesced into a single memory transaction if they are close enough together across threads. As message size increases, this will no longer happen and a memory transaction per thread is issued, severely harming throughput.

6.1.2 Salsa20

For this cipher, we ran the same tests as the ones described in Section 6.1.1. Figure 6.5 shows the results. Again, the bottleneck here is the PCIe bandwidth: 1300 MB/s peak throughput including memory transfers against over 9000 MB/s without, by far the fastest reported speed of Salsa20/12 — the best recorded timings of 2.57 clocks per byte give a throughput of roughly 1187 MB/s on a 3.2 GHz CPU [16].

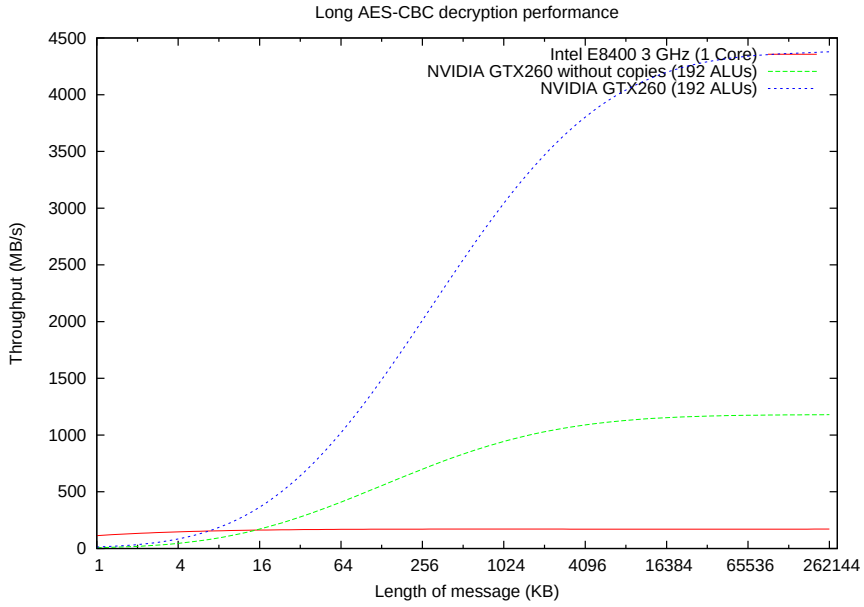

Figure 6.3: AES-CBC performance for various message sizes in a CPU and GPU implementation.

6.2 Asymmetric Primitives

6.2.1 Modular Exponentiation

The relative performance between the various methods mentioned in Section 5.2.1 with 512-bit integers and exponents is presented in Figure 6.8. One can observe that the serial implementation offers the best throughput when dealing with large numbers of messages, starting to outperform the CPU at approximately 1500 simultaneous 512-bit modular exponentiations. Below 1500 concurrent exponentiations we have the parallel approach competing with the CPU quite closely, whereas the residue number system approach seems to be slower than all the other ones. At about 4600 exponentiations, we see a drop in performance — this is the point where the GPU is saturated with threads, having to wait for a whole block to finish before starting a new one.

At the lower end of the spectrum, when very few exponentiations are needed, the GPU performance worsens. Figure 6.7 shows the times for 1 to 32 exponentiations done concurrently.

As one can observe, modular exponentiation at this range doesn't seem to be faster using any method on the GPU. Still, the pattern remains: the

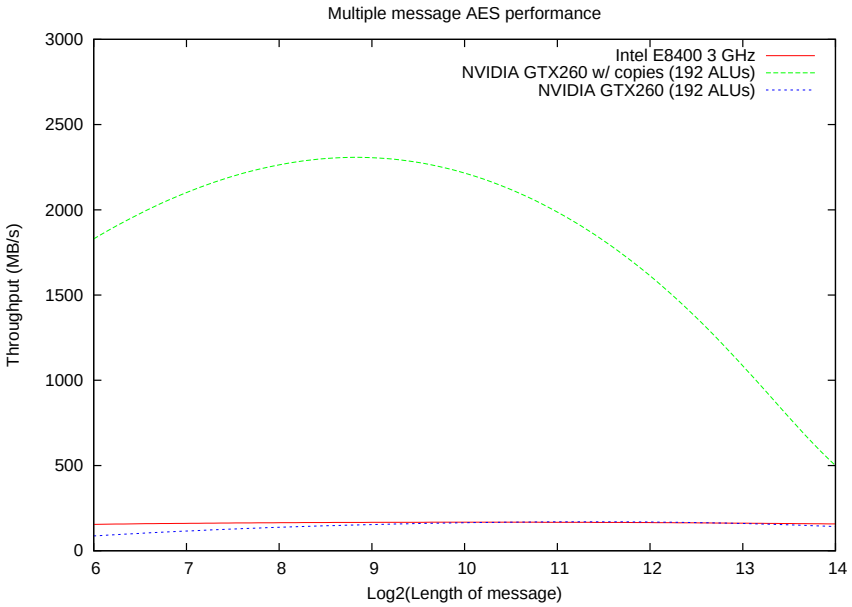

Figure 6.4: AES multiple message encryption performance for various message sizes in a CPU and GPU implementation.

parallel Montgomery arithmetic is faster than the residue number system counterpart. The point where the parallel approach starts to match the CPU appears to be at about 128 exponentiations.

At 1024-bit exponentiations, the situation is slightly different. While residue number systems maintain a poor relative performance against both CPU and the other methods, the parallel Montgomery arithmetic is clearly the winner, being consistently superior to both CPU and the GPU serial approach starting at batches of about 48 numbers to exponentiate. The superiority of the parallel approach is easily explained by the larger size of numbers, where a larger speedup can be had in each multiplication. The RNS approach is slower due to a larger required number of threads — 1024-bit digits require 96 threads per exponentiation, whereas the parallel Montgomery approach requires 32. As the number size grows, this difference should become close to 0; for cryptographically useful numbers, however, it appears RNS systems bring no significant advantages.

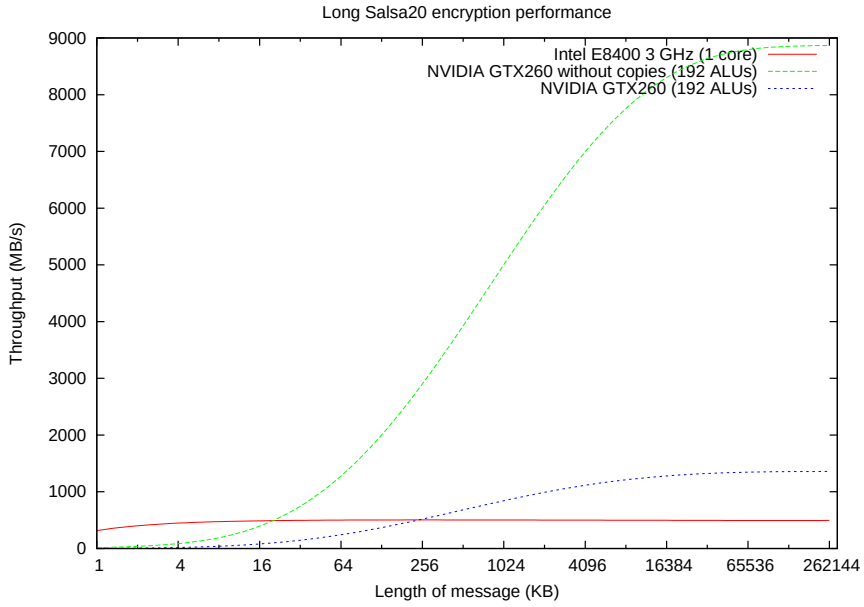

Figure 6.5: Salsa20 performance for various message sizes in a CPU and GPU implementation.

6.3 Discussion

6.3.1 Symmetric-key primitives

Symmetric ciphers seem to be more well-suited to the GPU instruction set than their public-key counterpart. However, they still suffer from a significant bottleneck — bandwidth. The numbers in Table 6.1 clearly show that there is potential for substantial speedups if more bandwidth is provided; this could be done by either a new PCIe revision or by integrating the GPU in the CPU, avoiding the transfers altogether.

Cipher	Throughput (with copies)	Throughput (without copies)
AES-ECB	1166	4406
AES-CTR	1032	2867
AES-CBC	1179	4378
Salsa20/12	1358	8868

Table 6.1: Peak throughputs, in MB/s, for the various parallel ciphers and modes of operation implemented in the NVIDIA GTX260.

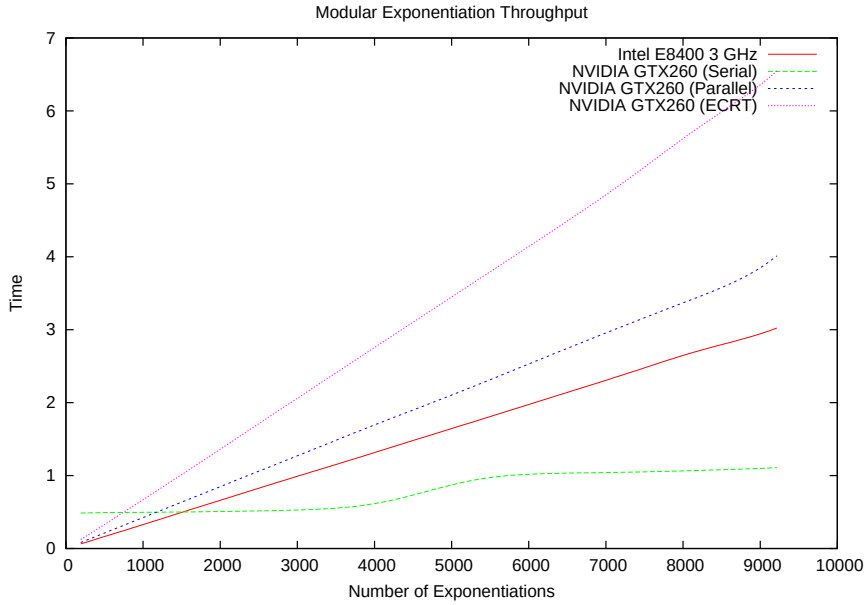

Figure 6.6: 512-bit modular exponentiation performance on the GPU and CPU.

With regard to the performance obtained we are very close, or have surpassed, the state of the art. The best results in the literature for AES-CTR encryption in a GPU are due to Harrison and Waldron, which obtain peaks of 864.25 MB/s with memory transfers and 1927.875 MB/s without [40]. The best results of AES-CTR in CPUs are due to Käsper and Schwabe, reaching 6.9 cycles per byte in the most recent Intel Core i7 CPUs [43]. This translates to a peak of 414 MB/s (per core) at 3.0 GHz. Our results of 1032 MB/s for AES-CTR with transfers and 2877 MB/s without are, as far as we know, the best to date.

6.3.2 Public-key primitives

The public-key results, particularly modular exponentiation, are harder to compare. Szerwinski et al. only measured performance for 1024 and 2048 bit exponentiation — but one of the most common public-key operations, RSA signing/decryption, can be performed modulo 512-bit integers! Assuming the theoretical speedup of 4, they have achieved 3252 512-bit modular exponentiations per second, against our 11686. At 1024-bit, Szerwinski achieved 813 modular exponentiations per second, against our result of 1215 [77]. The considerable slowdown from 512-bit to 1024-bit is to be expected — the the-

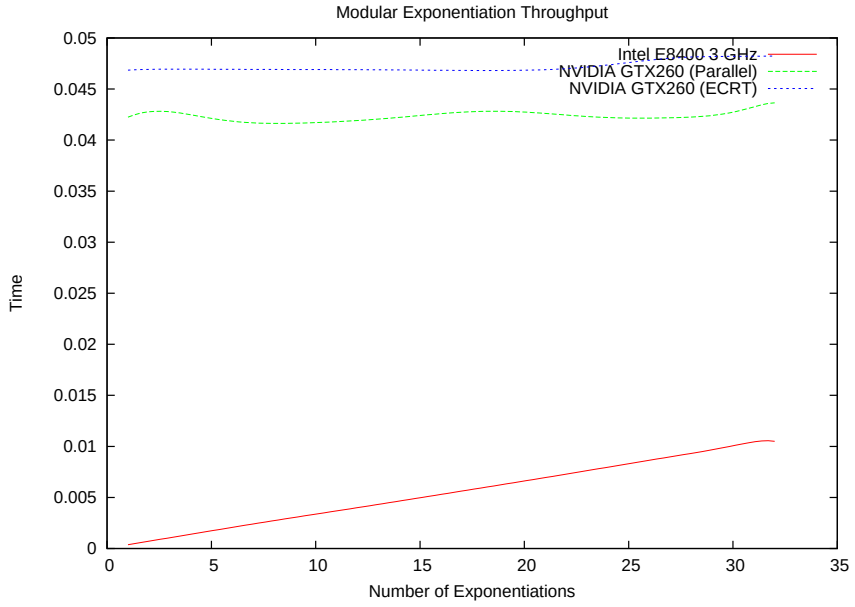

Figure 6.7: 512-bit modular exponentiation performance on the GPU and CPU, at lower message rates.

oretical slowdown is of about 8, as shown in Section 2.5.1.

The implementation of the the public-key algorithms considered in this work heavily relies on the modular exponentiation, described in Section 5.2.1. Thus, the results in throughput of these algorithms are extremely correlated with the results shown in Section 6.2.1 — they present the very same behavior, except for a small constant factor. Table 6.2 presents the throughputs achieved for the various algorithms, using several different keys sizes, compared against an Intel E8400 CPU, using the OpenSSL library.

Another aspect to consider is power efficiency. The NVIDIA GTX260 GPU has a thermal design power (TDP) of 182W; the Intel E8400 CPU has a TDP of 65W. At 512-bit exponentiations, the GPU can perform 64 exponentiations per watt, whereas the CPU reaches 47 exponentiations per watt. At larger key sizes the power efficiency decreases: 6 exponentiations per watt against 8 from the CPU. With symmetric ciphers, ignoring bandwidth constraints, we obtain 16 MB/s per watt AES-CTR encryption in the GPU, against 13 MB/s per watt in the CPU. The difference in Salsa20/12 is the most striking — 50 MB/s per watt in the GPU, versus 15 MB/s per watt in the CPU.

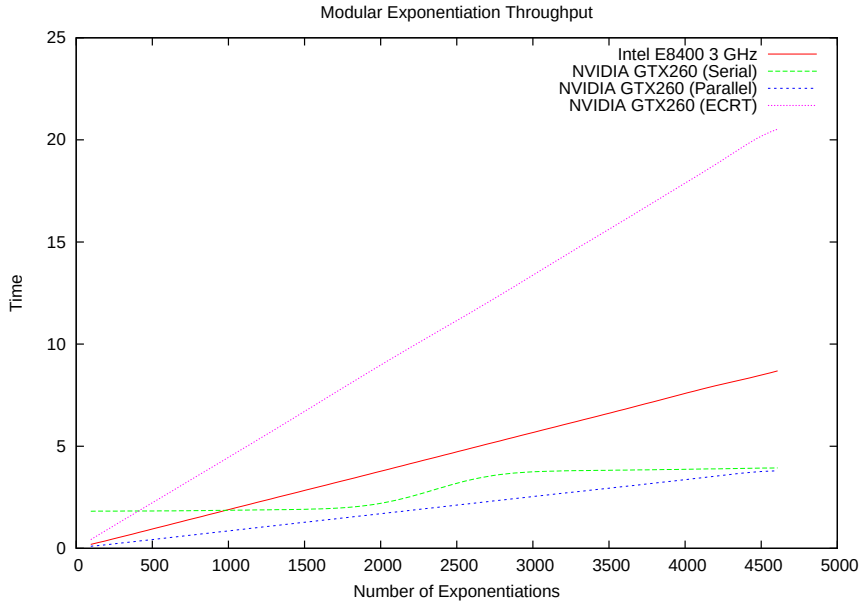

Figure 6.8: 1024-bit modular exponentiation performance on the GPU and CPU.

Algorithm		1024-bit		2048-bit	
		Encrypt	Decrypt	Encrypt	Decrypt
RSA	GPU	43457	6022	8947	590
	CPU	27980	1524	8796	265
DH	GPU	1215	1215	71	71
	CPU	530	530	80	80
DSA	GPU	5653	3256	813	418
	CPU	2878	2468	906	768

Table 6.2: Peak throughputs for several public-key algorithms and key sizes using the NVIDIA GTX260 GPU and Intel E8400 CPU.

Chapter 7

Conclusions and Future Work

Over the course of the last few years, GPUs have risen in relevance in many computational areas. Their impressive computing power in floating-point allowed GPUs to provide significant speedups in computational finance, chemistry, etc. In fact, a GPU-based cluster recently reached the Top 500 Supercomputer list, at 170 TFlops [53].

With this in mind, we started this project to harvest the computing power to perform encryptions, decryptions and other cryptographic operations in the GPU. The objective of this work was, then, the implementation in an efficient manner of symmetric and public-key algorithms: AES, Salsa20, RSA, Diffie-Hellman, DSA.

One of the main obstacles, especially in symmetric cryptography, is the overhead incurred in copying data to the GPU and back. From this results that using the GPU for cryptography is most fruitful when aiming for high-throughput, e.g. batching many operations together or encrypting long streams. Another obstacle to good performance in modular exponentiation was the poor integer multiplication in the GPU — only 24-bit multiplications are supported.

We believe that we were successful: the algorithms were indeed implemented. Not only were they implemented, the performance figures beat the state of the art implementations in the literature. This was accomplished by a careful study and development of the algorithms, particularly in multi-precision arithmetic: we've improved the Explicit Residue Number System approach, developed an innovative parallel classic arithmetic method and compared them with the classic serial approach. By using the best method for each key and batch size, we obtained some of the best throughputs to date in GPUs. Our results of 1032 MB/s AES-CTR encryption and 1358 MB/s Salsa20/12 encryption are the best known on software; the RSA-1024 encryption and decryption rates are the best on GPU to date, and outperform

the CPU by a large margin.

Finally, we bundled the aforementioned algorithms together in a function library, depending on OpenSSL, that enables developers to perform symmetric and public-key encryptions in a GPU more easily.

7.1 Future work

A whole class of primitives have been willingly ignored from this work — hash functions. While GPUs have been successfully used in computing hash functions for small strings, e.g. passwords, less success has been had in using GPUs to speed up hash computations of large streams. This relates to the structure of the most common hash functions — MD5, SHA-1, SHA-2 all share the same design, the *Merkle-Damgård construction*, which is non-parallelizable [56]. There are, however, alternative modes of operation that allow a hash computation to be parallelized. The most famous of these is the *Merkle tree*, where the original message is seen as the leaves of a tree and hash computations are performed in a divide and conquer fashion until the final result is reached [55]. Not only hash computations would benefit from this mode of operation; hash-based message authenticators, such as HMAC, would also benefit from this mode of operation [12].

Elliptic curve groups were first proposed by Koblitz for cryptographic uses in 1987 [48]. Since then, they've continuously grown in popularity, due to their best security per bit of key length than classic systems: elliptic curve key sizes of 160 bits are roughly equivalent in security to 1024-bit RSA [70]. The smaller number arithmetic could favor GPU implementations as well.

Shacham and Boneh improved the RSA handshake performance by grouping computations together in batches [74]. However, their technique is rather limited — only works for RSA decryption and very low exponents. Nevertheless, they proposed an architecture for a batching web server, where a server process receives requests and groups them together in batches. We can extend this idea for a GPU server process that receives handshake requests, packet encryption/decryption requests, etc, and processes them in batches as described in [74]. This could effectively turn a GPU into an off-the-shelf cryptographic accelerator.

NVIDIA is not the only GPU manufacturer — CUDA, however, is NVIDIA-specific. A new, vendor-neutral, open standard for heterogeneous computing named OpenCL has been released by the Khronos Group, also responsible for the OpenGL standard [61]. The SDK is not available for any major GPU vendor as of yet. Nonetheless, using OpenCL in the future for GPU computation appears to be a more compatible and easier to maintain choice than

the more restrictive CUDA.

Bibliography

[1] `http://3.14.by/en/read/md5_benchmark` (accessed 3 July 2009).

[2] `http://www-03.ibm.com/technology/resources/technology_cell_pdf_PowerXCell_PB_7May2008_pub.pdf` (accessed 3 July 2009).

[3] `http://www.intel.com/support/processors/sb/cs-023143.htm` (accessed 3 July 2009).

[4] *Secure Hash Standard*. National Institute of Standards and Technology, Washington, 1995. Note: Federal Information Processing Standard 180-1.

[5] *Digital signature standard (DSS)*. National Institute of Standards and Technology, Washington, 2000. URL: `http://csrc.nist.gov/publications/fips/`. Note: Federal Information Processing Standard 186-2.

[6] Ager, Bernhard, Holger Dreger, and Anja Feldmann: *Exploring the Overhead of DNSSEC*. 2005.

[7] "American National Standards Institute", "1430 Broadway, New York, NY 10018, USA": *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[8] Babbage, S. *et al.*: *The eSTREAM Portfolio (rev. 1)*. `http://www.ecrypt.eu.org/stream/portfolio_revision1.pdf`, September 2008.

[9] Babbage, Steve, Christophe De Cannière, Anne Canteaut, Carlos Cid, Henri Gilbert, Thomas Johansson, Matthew Parker, Bart Preneel, Vincent Rijmen, , and Matthew Robshaw: *The eSTREAM Portfolio*. April 2008.

[10] Bajard, Jean Claude and Laurent Imbert: *A Full RNS Implementation of RSA*. IEEE Trans. Comput., 53(6):769–774, 2004, ISSN 0018-9340.

[11] Barrett, P.: *Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor.* In Odlyzko, A.M. (editor): *Advances in Cryptology - CRYPTO '86, Santa Barbara, California*, volume 263 of *LNCS*, pages 311–323. Springer, 1987.

[12] Bellare, Mihir, Ran Canetti, and Hugo Krawczyk: *Keying hash functions for message authentication*, 1996. URL: `http://www-cse.ucsd.edu/ mihir/papers/hmac.html`.

[13] Bernstein, Daniel J.: *Pippenger's exponentiation algorithm.* URL: `http://cr.yp.to/papers.html`.

[14] Bernstein, Daniel J.: *Multidigit modular multiplication with the explicit Chinese remainder theorem.* 1995. URL: `http://cr.yp.to/papers.html`.

[15] Bernstein, Daniel J.: *The Salsa20 Family of Stream Ciphers.* New Stream Cipher Designs: The eSTREAM Finalists, pages 84–97, 2008.

[16] Bernstein, Daniel J. and Tanja Lange (editors): *eBACS: ECRYPT Benchmarking of Cryptographic Systems.* `http://bench.cr.yp.to`, accessed 17 June 2009.

[17] Bernstein, Daniel J. and Peter Schwabe: *New AES software speed records.* In *Progress in Cryptology - INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer, 2008.

[18] Bernstein, Daniel J. and Jonathan P. Sorenson: *Modular exponentiation via the explicit chinese remainder theorem.* Mathematics of Computation, 76(257):443–454, January 2007.

[19] Bielecki, Wlodzimierz and Dariusz Burak: *Biometrics, Computer Security Systems and Artificial Intelligence Applications*, chapter Parallelization of Standard Modes of Operation for Symmetric Key Block Ciphers, pages 101–110. Springer US, Secaucus, NJ, USA, 1st edition, November 2006.

[20] Bosma, Wieb, John Cannon, and Catherine Playoust: *The magma algebra system i: the user language.* J. Symb. Comput., 24(3-4):235–265, 1997, ISSN 0747-7171.

[21] Brauer, Alfred: *On addition chains.* Bulletin of the American Mathematical Society, 45:736–739, 1939, ISSN 0273–0979.

[22] Brent, Richard P. and Paul Zimmermann: *Modern Computer Arithmetic*. Version 0.3, 2009. `http://www.loria.fr/~zimmerma/mca/pub226.html`.

[23] Cook, D., J. Ioannidis, A. Keromytis, and J. Luck: *CryptoGraphics: Secret Key Cryptography Using Graphics Cards*, 2005. `citeseer.ist.psu.edu/cook05cryptographics.html`.

[24] Coppersmith, Don: *Small solutions to polynomial equations, and low exponent RSA vulnerabilities*. Journal of Cryptology, 10:233–260, 1997, ISSN 0933–2790.

[25] Costigan, Neil and Peter Schwabe: *Fast elliptic-curve cryptography on the Cell Broadband Engine*. In *Progress in Cryptology – AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 368–385. Springer, 2009.

[26] Costigan, Neil and Michael Scott: *Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3*. Cryptology ePrint Archive, Report 2007/061, 2007. `http://eprint.iacr.org/`.

[27] Crandall, Richard and Carl Pomerance: *Prime numbers. A Computational Perspective*. Springer-Verlag, New York, 2005, ISBN 978-0-387-25282-7,0-387-25282-7.

[28] Daemen, Joan and Vincent Rijmen: *Aes proposal: Rijndael*, 1998.

[29] Damgård, I., P. Landrock, and C. Pomerance: *Average case error estimates for the strong probable prime test*. 61(203):177–194, 1993.

[30] Diffie, Whitfield and Martin Hellman: *New directions in cryptography*. IEEE Transactions on Information Theory, 22:644–654, 1976, ISSN 0018–9448.

[31] Dixon, B. and A. K. Lenstra: *Massively parallel elliptic curve factoring*. Lecture Notes in Computer Science, 658:183–193, 1993.

[32] Downley, P., B. Leong, and R. Sethi: *Computing sequences with addition chains*. SIAM J. Comput., 10(3):638–646, 1981.

[33] ECRYPT: *The eSTREAM Project*. `http://www.ecrypt.eu.org/stream/`.

[34] ElGamal, Taher: *A public key cryptosystem and a signature scheme based on discrete logarithms.* IEEE Transactions on Information Theory, 31:469–472, 1985, ISSN 0018–9448.

[35] Fleissner, Sebastian: *GPU-Accelerated Montgomery Exponentiation.* In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I*, pages 213–220, Berlin, Heidelberg, 2007. Springer-Verlag, ISBN 978-3-540-72583-1.

[36] Garner, Harvey L.: *The residue number system.* In *IRE-AIEE-ACM '59 (Western): Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 146–153, New York, NY, USA, 1959. ACM.

[37] Hachez, Gaël and Jean Jacques Quisquater: *Montgomery exponentiation with no final subtraction: Improved results.* In Ç.K. Koç, C. Paar (editor): *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000*, pages 293–301. Springer-Verlag, 2000.

[38] Hardy, Godfrey H. and E. M. Wright: *An introduction to the theory of numbers.* Oxford University Press, 5th edition, 1979, ISBN 0–19–853170–2.

[39] Harkins, D. and D. Carrel: *The Internet Key Exchange (IKE)*, 1998. `http://www.ietf.org/rfc/rfc2409.txt`.

[40] Harrison, Owen and John Waldron: *Practical symmetric key cryptography on modern graphics hardware.* In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 195–209, Berkeley, CA, USA, 2008. USENIX Association.

[41] Hodjat, A. and I. Verbauwhede: *Minimum area cost for a 30 to 70 Gbit/s AES processor.* pages 83–88, Feb. 2004.

[42] Kahn, David: *The Codebreakers: The Story of Secret Writing.* Scribner, New York, NY, USA, revised edition, 1996, ISBN 0-684-83130-9.

[43] Käsper, Emilia and Peter Schwabe: *Faster and Timing-Attack Resistant AES-GCM.* In *Proceedings of CHES 2009*, 2009. to appear.

[44] Kaufman, C.: *Internet Key Exchange (IKEv2) Protocol.* RFC 4306 (Proposed Standard), December 2005. `http://www.ietf.org/rfc/rfc4306.txt`, Updated by RFC 5282.

[45] Kawamura, Shin ichi, Masanobu Koike, Fumihiko Sano, and Atsushi Shimbo: *Cox-rower architecture for fast parallel montgomery multiplication.* In *EUROCRYPT*, pages 523–538, 2000.

[46] Kivinen, T. and M. Kojo: *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE).* RFC 3526 (Proposed Standard), May 2003. `http://www.ietf.org/rfc/rfc3526.txt`.

[47] Knuth, Donald E.: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, 3rd edition, 1997, ISBN 0–201–89684–2.

[48] Koblitz, Neal: *Elliptic curve cryptosystems.* Mathematics of Computation, 48:203–209, 1987, ISSN 0025–5718.

[49] Koblitz, Neal: *A course in number theory and cryptography.* Springer-Verlag, New York, 2nd edition, 1994, ISBN 0–387–94293–9.

[50] Koç, Çetin Kaya, Tolga Acar, and Burton S. Kaliski, Jr.: *Analyzing and Comparing Montgomery Multiplication Algorithms.* IEEE Micro, 16(3):26–33, 1996, ISSN 0272-1732.

[51] Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym: *NVIDIA Tesla: A Unified Graphics and Computing Architecture.* IEEE Micro, 28(2):39–55, 2008, ISSN 0272-1732.

[52] Manavski, Svetlin: *Cuda compatible GPU as an efficient hardware accelerator for AES cryptography.* In *IEEE International Conference on Signal Processing and Communication, ICSPC 2007*, pages 65–68, Dubai, United Arab Emirates, November 2007.

[53] Matsuoka, Satoshi: *The TSUBAME Cluster Experience a Year Later, and onto Petascale TSUBAME 2.0.* In *Proceedings of the 14th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 8–9, Berlin, Heidelberg, 2007. Springer-Verlag, ISBN 978-3-540-75415-2.

[54] Menezes, Alfred J., Paul C. van Oorschot, and Scott A. Vanstone: *Handbook of applied cryptography.* CRC Press, Boca Raton, Florida, 1996, ISBN 0–8493–8523–7. URL: `http://cacr.math.uwaterloo.ca/hac`.

[55] Merkle, R. C.: *A digital signature based on a conventional encryption function.* In Pomerance, Carl (editor): *Advances in Cryptology - Crypto '87*, pages 369–378, Berlin, 1987. Springer-Verlag.

[56] Merkle, Ralph: *Secrecy, authentication, and public key systems*. PhD thesis, 1979.

[57] Miller, Gary L.: *Riemann's hypothesis and tests for primality*. Journal of Computer and System Sciences, 13:300–317, 1976, ISSN 0022–0000.

[58] Montgomery, Peter L.: *Modular multiplication without trial division*. Mathematics of Computation, 44:519–521, 1985, ISSN 0025–5718.

[59] Montgomery, Peter L. and Robert D. Silverman: *An FFT extension to the $P-1$ factoring algorithm*. Mathematics of Computation, 54:839–854, 1990, ISSN 0025–5718.

[60] Moss, Andrew, Dan Page, and Nigel Smart: *Toward Acceleration of RSA Using 3D Graphics Hardware*. In *Cryptography and Coding*, pages 369–388. Springer-Verlag LNCS 4887, December 2007. `http://www.cs.bris.ac.uk/Publications/Papers/2000772.pdf`.

[61] Munshi, Aaftab: *OpenCL 1.0 Specification*. Khronos Group, May 2009. URL: `http://www.khronos.org/registry/cl/`.

[62] National Institute of Standards and Technology: *Federal Information Processing Standard 197, The Advanced Encryption Standard (AES)*. 2001.

[63] NVIDIA: *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2009. URL: `http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf`.

[64] Payne, Bryson R. and Markus A. Hitz: *Implementation of residue number systems on GPUs*. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, page 57, New York, NY, USA, 2006. ACM, ISBN 1-59593-364-6.

[65] Pohlig, Stephen C. and Martin E. Hellman: *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*. IEEE Transactions on Information Theory, 24:106–110, 1978, ISSN 0018–9448.

[66] Polk, W. Timothy, Donna F. Dodson, and William E. Burr: *Cryptographic Algorithms and Key Sizes for Personal Identity Verification (NIST SP 800-78)*. Technical report, NIST, August 2007.

[67] Quisquater, J. and C. Couvreur: *Fast Decipherment Algorithm for RSA Public-key Cryptosystem.* Electronics Letters, 18(21):905–907, October 1982.

[68] Rabin, Michael O.: *Probabilistic algorithm for testing primality.* Journal of Number Theory, 12:128–138, 1980, ISSN 0022–314X.

[69] Rapuano, S. and E. Zimeo: *Measurement of performance impact of SSL on IP data transmissions.* 2007.

[70] Rechberger, Christian and Vincent Rijmen: *ECRYPT Yearly Report on Algorithms and Keysizes (2007-2008).* Technical report, 2008.

[71] Rivest, Ronald L., Adi Shamir, and Leonard M. Adleman: *A method for obtaining digital signatures and public-key cryptosystems.* Communications of the ACM, 21:120–126, 1978, ISSN 0001–0782.

[72] Rosenberg, Urmas: *Using Graphic Processing Unit in Block Cipher Calculations.* Master's thesis, University of Tartu, 2007.

[73] Ryoo, Shane, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu: *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA.* In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM, ISBN 978-1-59593-795-7.

[74] Shacham, Hovav and Dan Boneh: *Improving SSL Handshake Performance via Batching.* In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pages 28–43, London, UK, 2001. Springer-Verlag, ISBN 3-540-41898-9.

[75] Solovay, Robert M. and Volker Strassen: *A fast Monte-Carlo test for primality.* SIAM Journal on Computing, 6:84–85, 1977, ISSN 0097–5397.

[76] Sorenson, Jonathan P.: *A sublinear-time parallel algorithm for integer modular exponentiation.* In Odlyzko, Andrew M., Gary Walsh, and Hugh Williams (editors): *Conference on the mathematics of public key cryptography: the Fields Institute for Research in the Mathematical Sciences, Toronto, Ontario, June 12–17, 1999*, 1999.

[77] Szerwinski, Robert and Tim Güneysu: *Exploiting the Power of GPUs for Asymmetric Cryptography.* In *Cryptographic Hardware and Embedded Systems — CHES 2008*, pages 79–99. 2008.

[78] The OpenSSL Project: *OpenSSL: The open source toolkit for SSL/TLS*. `www.openssl.org`, April 2009.

[79] Vitkar, Sameer and Monish Shah: *Performance report on hardware accelerator with EMC Retrospect 7.5*. Technical report, 2007.

[80] Walter, Colin D.: *Montgomery exponentiation needs no final subtractions*. Electronics Letters, 35(21):1831–1832, October 1999.

[81] Yamanouchi, Takeshi: *AES Encryption and Decryption on the GPU*. In Nguyen, Hubert (editor): *GPU Gems 3*, chapter 36. Addison Wesley Professional, August 2007.

[82] Zhao, Li, R. Iyer, S. Makineni, and L. Bhuyan: *Anatomy and Performance of SSL Processing*. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 197–206, Washington, DC, USA, 2005. IEEE Computer Society, ISBN 0-7803-8965-4.