Exploiting the Power of GPUs for Asymmetric Cryptography

Robert Szerwinski and Tim Güneysu

Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
{szerwinski,gueneysu}@crypto.rub.de

Abstract. Modern Graphics Processing Units (GPU) have reached a dimension with respect to performance and gate count exceeding conventional Central Processing Units (CPU) by far. Many modern computer systems include – beside a CPU – such a powerful GPU which runs idle most of the time and might be used as cheap and instantly available co-processor for general purpose applications.

In this contribution, we focus on the efficient realisation of the computationally expensive operations in asymmetric cryptosystems on such off-the-shelf GPUs. More precisely, we present improved and novel implementations employing GPUs as accelerator for RSA and DSA cryptosystems as well as for Elliptic Curve Cryptography (ECC). Using a recent Nvidia 8800GTS graphics card, we are able to compute 813 modular exponentiations per second for RSA or DSA-based systems with 1024 bit integers. Moreover, our design for ECC over the prime field P-224 even achieves the throughput of 1412 point multiplications per second.

Keywords: Asymmetric Cryptosystems, Graphics Processing Unit, RSA, DSA, ECC.

1 Introduction

For the last twenty years graphics hardware manufacturers have focused on producing fast Graphics Processing Units (GPUs), specifically for the gaming community. This has more recently led to devices which outperform general purpose Central Processing Units (CPUs) for specific applications, particularly when comparing the MIPS (million instructions per second) benchmarks. Hence, a research community has been established to use the immense power of GPUs for general purpose computations (GPGPU). In the last two years, prior limitations of the graphics application programming interfaces (API) have been removed by GPU manufacturers by introducing unified processing units in graphics cards. They support a general purpose instruction set by a native driver interface and framework.

In the field of asymmetric cryptography, the security of all practical cryptosystems rely on hard computational problems strongly dependant on the choice of parameters. But with rising parameter sizes (often in the range of 1024–4096

© International Association for Cryptologic Research 2008

bits), however, computations become more and more challenging for the underlying processor. For modern hardware, the computation of a *single* cryptographic operation is not critical, however in a many-to-one communication scenario, like a central server in a company's data processing centre, it may be confronted with hundreds or thousands of simultaneous connections and corresponding cryptographic operations. As a result, the most common current solution are cryptographic accelerator cards. Due to the limited market, their price tags are often in the range of several thousands euros or US dollars. The question at hand is whether commodity GPUs can be used as high-performance public-key accelerators.

In this work, we will present novel implementations of cryptosystems based on modular exponentiations and elliptic curve operations on recent graphics hardware. To the best of our knowledge, this is the first publication making use of the CUDA framework for GPGPU processing of asymmetric cryptosystems. We will start with implementing the extremely wide-spread *Rivest Shamir Adleman* (RSA) cryptosystem [30]. The same implementation based on modular exponentiation for large integers can be used to implement the *Digital Signature Algorithm* (DSA), which has been published by the US National Institute of Standards and Technology (NIST) [25]. Recently, DSA has been adopted to elliptic curve groups in the ANSI X9.62 standard [2]. The implementation of this variant, called ECDSA, is the second major goal of this work.

2 Previous Work

Lately, the research community has started to explore techniques to accelerate cryptographic algorithms using the GPU. For example, various authors looked at the feasibility of the current industry standard for *symmetric* cryptography, the Advanced Encryption Standard (AES) [21,31,18,9]. Only two groups, namely Moss *et al.* and Fleissner, have aimed for the efficient implementation of modular exponentiation on the GPU [24,14]. Their results were not promising, as they were limited by the legacy GPU architecture and interface (cf. the next section). To the best of our knowledge there are neither publications about the implementation of these systems on modern, GPGPU-capable hardware nor on the implementation of elliptic curve based systems.

We aim to fill this gap by implementing the core operations for both systems efficiently on modern graphics hardware, creating the foundation for the use of GPUs as accelerators for public key cryptography. We will use Nvidia's current flagship GPU series, the G80 generation, together with its new GPGPU interface CUDA.

3 Using GPUs for General-Purpose Applications

The following section will give an overview over traditional GPU computing, followed by a more in-depth introduction to Nvidia's general purpose interface CUDA.

3.1 Traditional GPU Computing

Roughly, the graphics pipeline consist of the stages *transform & light*, *assemble primitives*, *rasterise* and *shade*. First GPUs had all functions needed to implement the graphics pipeline hardwired, but over time more and more stages became *programmable* by introducing specialised processors, e.g. vertex and fragment processors that made the transform & light and shading stages, respectively, more flexible.

When processing power increased massively while prices kept falling, the research community thought of ways to use these resources for computationally intense tasks. However, as the processors' capabilities were very limited and the API of the graphics driver was specifically built to implement the graphics pipeline, a lot of overhead needed to be taken into account. For example, all data had to be encoded in textures which are two dimensional arrays of pixels storing colour values for red, green, blue and an additional alpha channel used for transparency. Additionally, textures are *read-only* objects, which forced the programmers to compute one step of an algorithm, store the result in the frame buffer, and start the next step using a texture reference to the newly produced pixels. This technique is known as *ping-ponging.* Most GPUs did only provide instructions to manipulate floating point numbers, forcing GPGPU programmers to map integers onto the available mantissa and find ways to emulate bit-logical functions, e.g., by using look-up tables.

These limitations have been the main motivation for the key GPU manufacturers ATI/AMD and Nvidia to create APIs specifically for the GPGPU community and modify their hardware for better support: ATI's solution is called Close To the Metal (CTM) [1], while Nvidia presented the Compute Unified Device Architecture (CUDA), a radically new design that makes GPU programming and GPGPU switch places: The underlying hardware of the G80 series is an accumulation of *scalar* common purpose processing units ("unified" design) and quite a bit of "glue" hardware to efficiently map the graphics pipeline to this new design. GPGPU applications however directly map to the target hardware and thus graphics hardware can be programmed without any graphics API whatsoever.

3.2 Programming GPUs Using Nvidia's CUDA Framework

In general, the GPU's immense computation power mainly relies on its inherent parallel architecture. For this, the CUDA framework introduces the **thread** as smallest unit of parallelism, i.e., a small piece of concurrent code with associated state. However, when compared to threads on microprocessors, GPU threads have much lower resource usage and lower creation and switching cost. Note that GPUs are only effective when running a *high number* of such threads. A group of threads that is executed *physically* in parallel is called **warp**. All threads in one warp are executed in a *single instruction multiple data* (SIMD) fashion. If one or more thread(s) in the *same* warp need to execute different instructions, e.g., in case of a data-dependent jump, their execution will be serialised and the threads are called *divergent*. As the next level of parallelism, a (thread) **block** is a group of threads that can communicate with each other and synchronise their execution. The maximum number of threads per block is limited by the hardware. Finally, a group of blocks that have same dimensionality and execute the same CUDA program *logically* in parallel is called **grid**.

To allow optimal performance for different access patterns, CUDA implements a hierarchical memory model, contrasting the flat model normally assumed on computers. Host (PC) and device (GPU) have their own memory areas, called *host memory* and *device memory*, respectively. CUDA supplies optimised functions to transfer data between these separate spaces.

Each thread possesses its own **register file**, which can be read and written. Additionally, it can access its own copy of so-called **local memory**. All threads in the same *grid* can access the same on-chip read- and writable **shared memory** region. To prevent hazards resulting from concurrent execution of threads synchronisation mechanisms must be used. Shared memory is organised in groups called banks that can be accessed in parallel. All threads can access a read- and writable memory space called **global memory** and read-only regions called **constant memory** and **texture memory**. The second last is optimised for one-dimensional locality of accesses, while the last is most effective when being used with two-dimensional arrays (matrices). Note that the texture and constant memories are the only regions that are cached. Thus, all accesses to the off-chip regions global and local memory have a high access latency, resulting in penalties when being used too frequently.

The hardware consists of a number of so-called *multiprocessors* that are build from SIMD processors, on-chip memory and caches. Clearly, one processor executes a particular thread, the same warp being run on the multiprocessor at the same time. One or more blocks are mapped to each multiprocessor, sharing its resources (registers and shared memory) and get executed on a time-sliced basis. When a particular block has finished its execution, the scheduler starts the next block of the grid until all blocks have been run.

Design Criteria for GPU Implementations. To achieve optimal performance using CUDA, algorithms must be designed to run in a multitude of parallel threads and take advantage of the presented hierarchical memory model. In the following, we enumerate the key criteria necessary for gaining the most out of the GPU by loosely following the CUDA programming guide [27] and a talk given by Mark Harris of Nvidia [17].

A. ***Maximise use of available processing power***
   - A1. **Maximise independent parallelism** in the algorithm to enable easy partitioning in threads and blocks.
   - A2. **Keep resource usage low** to allow concurrent execution of as many threads as possible, i.e., use only a small number of registers per thread and shared memory per block.
   - A3. **Maximise arithmetic intensity**, i.e., match the arithmetic to bandwidth ratio to the GPU design philosophy: GPUs spend their transistors

on ALUs, not caches. Bearing this in mind allows to hide memory access latency by the use of independent computations (*latency hiding*). Examples include using arithmetic instructions with high throughput as well as re-computing values instead of saving them for later use.

A4. **Avoid divergent threads** in the *same* warp.

B. ***Maximise use of available memory bandwidth***

B1. **Avoid memory transfers between host and device** by shifting more computations from the host to the GPU.

B2. **Use shared memory** instead of global memory **for variables**.

B3. **Use constant or texture memory** instead of global memory **for constants**.

B4. **Coalesce global memory accesses**, i.e., choose access patterns that allow to combine several accesses in the same warp to one, wider access.

B5. **Avoid bank conflicts** when utilising shared memory, i.e., choose patterns that result in the access of *different* banks per warp.

B6. **Match access patterns** for constant and texture memory **to** the **cache design**.

CUDA Limitations. Although CUDA programs are written in the C language together with extensions to support the memory model, allow synchronisation and special intrinsics to access faster assembler instructions, it also contains a number of limitations that negatively affect efficient implementation of public key cryptography primitives. Examples are the lack for additions/subtractions with carry as well as the missing support for inline assembler instructions[1].

4 Modular Arithmetic on GPUs

In the following section we will give different ways do realise *modular arithmetic* on a GPU efficiently, keeping the aforementioned criteria in mind. For the RSA cryptosystem we need to implement arithmetic modulo N, where N is the product of two large primes p and q: $N = p \cdot q$. The arithmetic of both DSA systems, however, is based on the prime field $GF(p)$ as the lowest-level building block. Note that the DSA systems both use a *fixed* – in terms of sessions or key generations – prime p, thus allowing to choose special primes at build time that have advantageous properties when reducing modulo p. For example, the US National Institute of Standards and Technology (NIST) proposes a set of *generalised Mersenne primes* in the *Digital Signature Standard* (DSS) [25, Appendix 6]. As the RSA modulus N is the product of the two secret primes p and q that will be chosen secretly *for each* new key pair, we cannot optimise for the modulus in this case.

[1] Nvidia published their own (abstract) assembler language PTX [28], however as of CUDA version 1.0 one kernel *cannot* contain code both generated from the C language and PTX.

Modular Addition and Subtraction. In general, addition $s \equiv a + b \bmod m$ of two operands a and b, where $0 \leq a, b < m$, is straightforward, as the result of the plain addition operation $a + b$ always satisfies $0 \leq a + b < 2m$ and therefore needs at maximum one subtraction of m to fulfil $0 \leq s < m$. Due to the SIMD design, we require algorithms that have a uniform control flow in all cases and compute both $a + b$ and $a + b - m$ and decide afterwards which is the correctly reduced result, cf. Criterion A4. Subtraction $d \equiv a - b \bmod m$ can be treated similarly: we compute both $a - b$ and $a - b + m$ and use a sign test at the end to derive the correctly reduced result.

Modular Multiplication. Multi-precision modular multiplication $r \equiv a \cdot b \bmod m$ is usually the most critical operation in common asymmetric cryptosystems. In a straightforward approach to compute r, we derive a double-sized product $r' = ab$ first and reduce afterwards by multi-precision division. Besides the quadratic complexity of standard multiplication, division is known to be very costly and should be avoided whenever possible. Thus, we will discuss several multiplication strategies to identify an optimal method for implementation on GPUs.

4.1 Modular Multiplication Using Montgomery's Technique

In 1985 Peter L. Montgomery proposed an algorithm [23] to remove the costly division operation from the modular reduction. Koç et al. [6] give a survey of different implementation options. As all multi-precision Montgomery multiplication algorithms feature no inherent parallelism except the possibility to pipeline, we do *not* consider them optimal for our platform and implement the method with the lowest temporary space requirement of $n+2$ words, coarsely integrated operand scanning (CIOS), as a reference solution only (cf. to Algorithm 1).

4.2 Modular Multiplication in Residue Number Systems (RNS)

As an alternative approach to conventional base-2^w arithmetic, we can represent integers based on the idea of the Chinese Remainder Theorem, by encoding an integer x as a tuple formed from its residues x_i modulo n relatively prime w-bit moduli m_i, where $|x|_{m_i}$ denotes $x \bmod m_i$:

$$\langle x \rangle_{\mathcal{A}} = \langle x_0, x_1, \ldots, x_{n-1} \rangle_{\mathcal{A}} = \langle |x|_{m_0}, |x|_{m_1}, \ldots, |x|_{m_{n-1}} \rangle_{\mathcal{A}} \tag{1}$$

Here, the ordered set of relatively prime moduli $(m_0, m_1, \ldots, m_{n-1})$, gcd $(m_i, m_j) = 1$ for all $i \neq j$, is called *base* and denoted by $\mathcal{A}$. The product of all moduli, $A = \prod_{i=0}^{n-1} m_i$ is called *dynamic range* of $\mathcal{A}$, i.e., the number of values that can be *uniquely* represented in $\mathcal{A}$. In other words, all numbers in $\mathcal{A}$ get implicitly reduced modulo A. Such a representation in RNS has the advantage that addition, subtraction and multiplication can be computed *independently* for all residues:

$$\langle x \rangle_{\mathcal{A}} \circ \langle y \rangle_{\mathcal{A}} = \langle |x_0 \circ y_0|_{m_0}, |x_1 \circ y_1|_{m_1}, \ldots, |x_{n-1} \circ y_{n-1}|_{m_{n-1}} \rangle_{\mathcal{A}}, \circ \in \{+, -, \cdot\} \tag{2}$$

Algorithm 1. Montgomery Multiplication for Multi-Precision Integers (CIOS Method) [6]

```
Require: Modulus $M$ and radix $R = 2^{wn}$ s.t. $R > M$ and $\gcd(R, M) = 1$; $M'_0 = (-M^{-1} \bmod R) \bmod 2^w$, two unsigned integers $0 \le A, B < M$ in Montgomery form, i.e. $X = (X_{n-1}X_{n-2} \dots X_0)_{2^w}$ for $X \in \{A, B, M\}$.
Ensure: The product $C = ABR^{-1} \pmod{M}$, $0 \le C < M$, in Montgomery form.
 1: $T \leftarrow 0$
 2: for $i$ from $0$ to $n-1$ do
 3:     $c \leftarrow 0$
 4:     for $j$ from $0$ to $n-1$ do {Multiplication}
 5:         $(c, T_j) \leftarrow A_j \cdot B_i + T_j + c$
 6:     end for
 7:     $(T_{n+1}, T_n) \leftarrow T_n + c$

 8:     $m \leftarrow T_0 \cdot M'_0 \bmod 2^w$ {Reduction}
 9:     $(c, T_0) \leftarrow m \cdot M_0 + T_0$
10:     for $j$ from $1$ to $n-1$ do
11:         $(c, T_{j-1}) \leftarrow m \cdot M_j + T_j + c$
12:     end for
13:     $T_{n-1} \leftarrow T_n + c$
14:     $T_n \leftarrow T_{n+1} + c$
15: end for
16: return $(T_{n-1}T_{n-2} \dots T_0)_{2^w}$
```

which allows carry-free computations[2] and multiplication without partial products. However, some information involving the whole number x cannot be easily computed. For instance, sign and overflow detection and comparison of magnitude are hard, resulting from the fact that residue number systems are no weighted representation. Furthermore, division and as a result reduction modulo an arbitrary modulus $M \neq A$ is *not* as easy as in other representations.

But similar to the basic idea of Montgomery multiplication, one can create a modular multiplication method for input values in RNS representation as shown in Algorithm 2, which involves a second base $\mathcal{B} = (\widetilde{m}_0, \widetilde{m}_1, \dots, \widetilde{m}_{n-1})$ with corresponding dynamic range B. It computes a value $v = XY + fM$ that is equivalent to $0 \bmod A$ and $XY \bmod M$. Thus, we can safely divide by A, i.e., multiply by its inverse modulo B, to compute the output $XYA^{-1} \pmod{M}$. Note that the needed reduction modulo A to compute f is *free* in $\mathcal{A}$.

All steps of the algorithm can be efficiently computed in parallel. However, a method to convert between both bases, a *base extension* mechanism, is needed. We take three different options into account: the method based on a Mixed Radix System (MRS) according to Szabó and Tanaka [37], as well as CRT-based methods due to Shenoy and Kumaresan [33], Kawamura *et al.* [20] and Bajard *et al.* [3]. We present a brief introduction of these methods, but for more detailed information about base extensions, please see the recent survey at [5].

[2] Inner-RNS operations still contain carries.

Algorithm 2. Modular Multiplication Algorithm for Residue Number Systems [20]

```
Require: Modulus $M$, two RNS bases $\mathcal{A}$ and $\mathcal{B}$ composed of $n$ distinct moduli $m_i$
    each, $\gcd(A, B) = \gcd(A, M) = 1$ and $B > A > 4M$.
    Two factors $X$ and $Y$, $0 \leq X, Y < 2M$, encoded in both bases and in Montgomery
    form, i.e. $\langle X\rangle_{\mathcal{A}\cup\mathcal{B}}$ and $\langle Y\rangle_{\mathcal{A}\cup\mathcal{B}}$, $X = xA \pmod M$ and $Y = yA \pmod M$.
Ensure: The product $C = XYA^{-1} \pmod M$, $0 \leq C < 2M$, in both bases and
    Montgomery form.
1: $\langle u\rangle_{\mathcal{A}\cup\mathcal{B}} \leftarrow \langle X\rangle_{\mathcal{A}\cup\mathcal{B}} \cdot \langle Y\rangle_{\mathcal{A}\cup\mathcal{B}}$
2: $\langle f\rangle_{\mathcal{A}} \leftarrow \langle u\rangle_{\mathcal{A}} \cdot \langle -M^{-1}\rangle_{\mathcal{A}}$
3: $\langle f\rangle_{\mathcal{A}\cup\mathcal{B}} \leftarrow \text{BaseExtend}(\langle f\rangle_{\mathcal{A}})$
4: $\langle v\rangle_{\mathcal{B}} \leftarrow \langle u\rangle_{\mathcal{B}} + \langle f\rangle_{\mathcal{B}} \cdot \langle M\rangle_{\mathcal{B}}$ $\{\langle v\rangle_{\mathcal{A}} = 0$ by construction$\}$
5: $\langle w\rangle_{\mathcal{B}} \leftarrow \langle v\rangle_{\mathcal{B}} \cdot \langle A^{-1}\rangle_{\mathcal{B}}$
6: $\langle w\rangle_{\mathcal{A}\cup\mathcal{B}} \leftarrow \text{BaseExtend}(\langle w\rangle_{\mathcal{B}})$
7: return $\langle w\rangle_{\mathcal{A}\cup\mathcal{B}}$
```

4.3 Base Extension Using a Mixed Radix System (MRS)

The classical way to compute base extensions is due to Szabó and Tanaka [37]. Let $(m_0, \ldots, m_{n-1})$ be the MRS base *associated* to $\mathcal{A}$. Then, each integer x can be represented in a *mixed radix system* as

$$x = x'_0 + x'_1 m_0 + x'_2 m_0 m_1 + \cdots + x'_{n-1} m_0 \ldots m_{n-2}. \tag{3}$$

The MRS digits x'_i can be derived from the residues x_i by a recursive strategy: where $m^{-1}_{(i,j)}$ are the pre-computed inverses of m_j modulo m_i. To convert x from

$$\begin{aligned} x'_0 &= x_0 \pmod{m_0} \qquad (4)\\ x'_1 &= (x_1 - x'_0)m^{-1}_{(1,0)} \pmod{m_1}\\ &\vdots\\ x'_{n-1} &= (\cdots((x_n - x'_0)m^{-1}_{(n-1,0)} - x'_1)m^{-1}_{(n-1,1)} - \cdots - x'_{n-2})m^{-1}_{(n-1,n-2)} \pmod{m_{n-1}} \end{aligned}$$

this representation to a target RNS base, we could reduce Equation (3) by each target modulus $\widetilde{m}_k$, involving pre-computed constants $\widetilde{c}_{(k,i)} = \left|\prod_{l=0}^{i-1} m_l\right|_{\widetilde{m}_k}$. But instead of creating a table for all $\widetilde{c}_k$, a recursive approach is more efficient in our situation, eliminating the need for table-lookups [4], and allowing to compute all residues in the target base in parallel:

$$|x|_{\widetilde{m}_k} = \left|(\ldots((x'_{n-1}m_{n-2} + x'_{n-2})m_{n-3} + x'_{n-3})m_{n-4} + \cdots + x'_1)m_0 + x_0\right|_{\widetilde{m}_k} \tag{5}$$

4.4 Base Extension Using the Chinese Remainder Theorem (CRT)

Recall the definition of the CRT and adopt it to the source base $\mathcal{A}$ with dynamic range A:

$$x = \sum_{k=0}^{n-1} \hat{A}_k \left|\frac{x_k}{\hat{A}_k}\right|_{m_k} - \alpha A, \qquad \alpha < n \tag{6}$$

where $\hat{A}_k = A/m_k$ and α is an integer s.t. $0 \leq x < A$. Note that α is strictly upper-bounded by n. When reducing this equation with an arbitrary target modulus, say $\widetilde{m}_i$, we yield

$$|x|_{\widetilde{m}_i} = \left| \sum_{k=0}^{n-1} \left| \hat{A}_k \right|_{\widetilde{m}_i} \delta_k \; - |\alpha A|_{\widetilde{m}_i} \right|_{\widetilde{m}_i}, \qquad \delta_k = \left| x_k \cdot \hat{A}_k^{\;-1} \right|_{m_k} \tag{7}$$

where $\left|\hat{A}_k\right|_{\widetilde{m}_i}$, $\left|\hat{A}_k^{\;-1}\right|_{m_k}$ and $|A|_{\widetilde{m}_i}$ are pre-computed constants. Note that the δ_k do *not* depend on the target modulus and can thus be reused in the computation of a different target residue.

This is an efficient way to compute all residues modulo the target base, provided we know the value of α. While involving a couple of look-ups for the constants as well, the instruction flow is highly uniform (cf. Criterion A4) and fits to our SIMD architecture, i.e., we can use n threads to compute the n residues of x in the target base in parallel (cf. Criterion A1).

The first technique to compute such an α is due to Shenoy and Kumaresan [33] and requires a *redundant modulus* $m_r \geq n$ that is relatively prime to all other moduli m_j and $\widetilde{m}_i$, i.e., $\gcd(A, m_r) = \gcd(B, m_r) = 1$. Consider Equation 7, set $\widetilde{m}_i = m_r$ and rearrange it to the following:

$$|\alpha|_{m_r} = \left| |A^{-1}|_{m_r} \cdot \left(\sum_{k=0}^{n-1} \left| \hat{A}_k \right|_{m_r} \delta_k - |x|_{m_r} \right) \right|_{m_r}. \tag{8}$$

Since $\alpha < n \leq m_r$ it holds that $\alpha = |\alpha|_{m_r}$ and thus Equation 8 computes the exact value of α, involving the additional constant $|A^{-1}|_{m_r}$.

Kawamura *et al.* propose a different technique that approximates α using fixed-point computations [20]. Consider Equation 7, rearrange it and divide by A:

$$\alpha \quad = \quad \sum_{k=0}^{n-1} \frac{\delta_k}{m_k} - \frac{|x|_{\widetilde{m}_i}}{A} \quad = \quad \left\lfloor \sum_{k=0}^{n-1} \frac{\delta_k}{m_k} \right\rfloor. \tag{9}$$

Next, they approximate α by using $\mathrm{trunc}_r(\delta_k)$ as numerator and 2^w as denominator and adding a properly chosen offset σ, where $\mathrm{trunc}_r(\delta_k)$ sets the last $w-r$ bits of δ_k to zero:

$$\alpha' = \left\lfloor \sum_{k=0}^{n-1} \frac{\mathrm{trunc}_r(\delta_k)}{2^w} \; + \; \sigma \right\rfloor = \left\lfloor \frac{1}{2^r} \sum_{k=0}^{n-1} \left\lfloor \delta_k / 2^{w-r} \right\rfloor \; + \; \sigma \right\rfloor, \tag{10}$$

Thus, the approximate value α' can be computed in fixed-point arithmetic as integer part of the sum of the r most-significant bits of all δ_k. Provided σ is chosen correctly, Equation 10 will compute $\alpha' = \alpha$, and the resulting base extension will be exact.

Finally, Bajard *et al.* follow the most radical approach possible [3]: they allow an offset of $\alpha A \leq (n-1)A$ to occur in Equation 7 and thus do not need to compute α at all. After the first base extension we have $f' = f + \alpha A$ and thus

$w' = w + \alpha M$, i.e., the result w' will contain a maximum offset of $(n-1)M$, and thus be equivalent to $w \mod M$. However, this technique needs additional measures of precaution in the multiplication algorithm, which predominantly condense in the higher dynamic ranges needed.

4.5 Multiplication Modulo Generalised Mersenne Primes

For some cryptosystems like DSA, arithmetic in an underlying prime field is required. Taking advantage of the special structure of Mersenne primes, the reduction modulo p after a multiplication can be carried out very efficiently. Using such a method, we can compute r' using a standard multi-precision multiplication method first, followed by a reduction algorithm that is specific for the given prime. In this work, we will use an algorithm to efficiently compute multiplications modulo P-224, where P-224 is the 224 bit prime proposed by NIST [25]. Algorithm 3 performs the complete reduction for this prime with only two additions and two subtractions of 224 bit integers and a subsequent correction step to determine the correct value of $r \equiv r' \mod p$, since $-2p \leq r' < 3p$ must be considered. Note that this final correction step additionally needs the same amount of computations, as we have to avoid data-dependant branches (cf. Criterion A4).

Algorithm 3. NIST Reduction for P-224 $= 2^{224} - 2^{96} + 1$

```
Require: Double-sized integer $r' = (r'_{13}, \ldots, r'_2, r'_1, r'_0)$ in base $2^{32}$ and $0 \leq r' < \text{P-224}^2$
Ensure: Single-sized integer $r \equiv r' \mod \text{P-224}$, $0 \leq r < \text{P-224}$.
1: Concatenate $r'_i$ to following 224-bit integers $t_j$:
     $t_1 = (r'_6, r'_5, r'_4, r'_3, r'_2, r'_1, r'_0)$, $t_2 = (r'_{10}, r'_9, r'_8, r'_7, 0, 0, 0)$, $t_3 = (0, r'_{13}, r'_{12}, r'_{11}, 0, 0, 0)$
     $t_4 = (0, 0, 0, 0, r'_{13}, r'_{12}, r'_{11})$, $t_5 = (r'_{13}, r'_{12}, r'_{11}, r'_{10}, r'_9, r'_8, t_7)$
2: Compute $r'' = t_1 + t_2 + t_3 - t_4 - t_5$
3: return $r = r'' \mod \text{P-224}$
```

5 Implementation

In this section we will describe the implementation of two primitive operations for a variety of cryptosystems: first, we realise modular exponentiation on the GPU for use with RSA, DSA and similar systems. Second, for ECC-based cryptosystems we present an efficient point multiplication method which is the fundamental operation, e.g., for ECDSA or ECDH [16].

5.1 Modular Exponentiation Using the CIOS Method

We implemented the CIOS Method as introduced in Algorithm 1 for sequential execution since it does *not* include any inherent parallelism. Fan *et al.* describe efficient ways to pipeline such an algorithm for the use on multi-core systems [13].

This would however need fairly complex coordination and memory techniques and thus will not be considered further for our implementation, cf. Criteria A4 and B4-B6.

As all modular exponentiations are independent, we let each thread compute exactly one modular exponentiation in parallel with all others. Resulting from that, this solution only profits from coarse-grained parallelism. We assume the computation of distinct exponentiations, each having the *same* exponent t – for example RSA signatures using the same key – and thus need to transfer only the messages P_i for each exponentiation to the device and the result $P_i^t \pmod N$ back to the host. As a result, every thread executes the same control flow, fulfilling Criterion A4. To accelerate memory transfers between host and device, we use page-locked host memory and pad each message to a fixed length that forces the starting address of each message to values that are eligible for global memory coalescing (cf. Criteria B1 and B4).

For modular exponentiation based on Algorithm 1, we applied the straightforward binary right-to-left method [35]. During exponentiation, each *thread* needs three temporary values of $(n+2)$ words each that get used as input and output of Algorithm 1 in a round-robin fashion by pointer arithmetic. Thus, $3(n+2)$ words are required. This leads to 408 bytes and 792 bytes for 1024 bits and 2048 bit parameters, respectively. Each multiprocessor features 16384 bytes of shared memory, resulting in a maximum number of $\lfloor 16386/408 \rfloor = 40$ and $\lfloor 16386/792 \rfloor = 20$ threads per multiprocessor for 1024 and 2048 bits, respectively, if we use shared memory for temporary values. Clearly, both solutions are inefficient when considering that each multiprocessor is able to execute 768 threads per block in principle (i.e., we favour Criterion A2 over B2).

Thus, we chose to store the temporary values in *global memory*. We have to store the values *interleaved* so that memory accesses of one word by all threads in a warp can be combined to *one* global memory access. Hence, for a given set of values $(A, B, C, \ldots)$ consisting each of $n+2$ words $X = (x_0, x_1, \ldots, x_{n+1})$, we store all first words $(a_0, b_0, c_0, \ldots)$ for all threads in the same block, then all second words $(a_1, b_1, c_1, \ldots)$, and so on (cf. Criterion B4).

Moreover, we have to use *nailing* techniques, as CUDA does not yet include add-with-carry instructions. Roughly speaking, nailing reserves one or more of the high-order bits of each word for the carry that can occur when adding two numbers. To save register and memory space, however, we store the full word of w bits per register and use bit shifts and `and`-masking to extract two nibbles, each providing sufficient bits for the carry (cf. Criterion A3). This can be thought of decomposing a 32 bit addition in two 16 bit additions plus the overhead for carry handling.

5.2 Modular Exponentiation Using Residue Number Systems

Computations in residue number systems yield the advantage of being inherently parallel. According to Algorithm 2 all steps are computed in *one* base only, except for the first multiplication. Thus, the optimal mapping of computations to threads is as follows: each thread determines values for one modulus in the two bases. As a result, we have coarse-grained (different exponentiations) and fine-grained parallelism (base size), fulfilling Criterion A1. We call n' the number of residues that can be computed in parallel, i.e., the number of threads per encryption. The base extension by Shenoy *et al.* needs a *redundant* residue starting from the first base extension to be able to compute the second base extension. To reflect this fact, we use two RNS bases $\mathcal{A}$ and $\mathcal{B}$, having n moduli each, and an additional residue m_r resulting in $n' = n + 1$. For all other cases, it holds that $n' = n$.

Considering the optimal number of bits per modulus, we are faced with $w = 32$ bit integer registers on the target hardware. Thus, to avoid multi-precision techniques, we can use moduli that are smaller than 2^w. The hardware can compute 24 bit multiplications faster than full 32 bit multiplications. However, CUDA does *not* expose an intrinsic to compute the most-significant 16 bits of the result. Using 16 bit moduli would waste registers and memory and increase the *number of memory accesses* as well. Thus, we prefer *full* 32 bit moduli to save storage resources at the expense of higher computational cost (cf. Criteria A2 and A3).

For Algorithm 1 to work, the dynamic ranges A and B and the modulus M have to be related according to $B > A > 2^2M$, or $B > A > (2+n)^2M$ when using Bajard's method. For performance reasons, we consider *full warps* of 32 threads only, resulting in a slightly reduced size of M. The figures for all possible combinations can be found in Table 6 in the Appendix. For input and output values, we assume that all initial values will have been already converted to both bases (and possibly the redundant modulus m_r) and that output values will be returned in the same encoding. Note that it would be sufficient to transfer values in *one* base only and do a base extension for all input values (cf. Criterion B1, transferring values in both bases results in a more compact kernel together with a slightly higher latency). Different from the CIOS method, temporary values can be kept local for each thread, i.e., every thread stores its assigned residues in registers. Principally all operations can be performed *in parallel* on different residues and – as a result – the plain multiplication algorithm does *not* need any synchronisations. However, both properties do *not* hold for the base extension algorithms.

Mixed Radix Conversion. Recall that the mixed radix conversion computes the mixed radix representation from all residues in the source base first and uses this value to compute the target residues. The second step involves the computation of n' residues and can be executed in parallel, i.e., each thread computes the residue for 'its' modulus. As a result, we have to store the n MRS digits in shared memory to make them accessible to all threads (cf. Criteria A1 and B2). The first step however is the main caveat of this algorithm due to its highly divergent nature as each MRS digit is derived from the residue of a temporary variable in a *different* modulus (and thus thread) and depends on all previously computed digits, clearly breaking Criterion A4 and resulting in serialisation of executions. Additionally, note that threads having already computed an MRS digit do not generate any useful output anymore.

CRT-Based Conversion. The first step for all CRT-based techniques is to compute the δ_k for each source modulus and can be carried out by one thread for each value. Second, all n' threads compute a weighted sum involving δ_k and a modulus-dependent constant. Note that all threads need to access *all* δ_k and thus δ_k have to be stored in shared memory (cf. Criterion B2). Third, α has to be derived, whose computation is the main difference in the distinguished techniques. α is needed by *all* threads later and thus needs to be stored in shared memory as well. After computing α all threads can proceed with their independent computations.

Bajard's method does not compute α and consequently needs no further operations. For Shenoy's method, the second step above is needed for the redundant modulus m_r as well, which can be done in parallel with all other moduli. Then, a *single* thread computes α and writes it to shared memory. The redundant residue m_r comes at the price of an additional thread, however the divergent part needed to compute α does only contain one addition and one multiplication modulo m_r. Kawamura's method needs to compute the sum of the r most significant bits of all δ_k. While the right-shift of each δ_k can be done using *all* threads, the sum over all shifted values and the offset has to be computed using a *single* thread. A final right-shift results in the integer part of the sum, namely α.

Comparison and Selection. Clearly, Bajard's method is the fastest since it involves no computation of α. Shenoy's method only involves a small divergent part. However, we pay the price of an additional thread for the redundant modulus, or equivalently decrease the size of M. Kawamura's technique consists of a slightly larger divergent part, however it does neither include look-ups nor further reduces the size of M.

Not all base extension mechanisms can be used for both directions required for Algorithm 2. For Bajard's method, consider the consequence of an offset in the second base extension: we would compute some w'' in base $\mathcal{A}$ that is *not equal* to the w' in $\mathcal{B}$. As a result, neither $\langle w' \rangle_{\mathcal{A}}$ nor $\langle w'' \rangle_{\mathcal{B}}$ could be computed leading to an invalid input for a subsequent execution of Algorithm 2. Thus, their method is only available for $\mathcal{A} \rightarrow \mathcal{B}$ conversions. Shenoy's method can only be used for the *second* base extension as there is no efficient way to carry the redundant residue through the computation of f modulo A. The technique by Kawamura *et al.* would in principle be available for both conversions. However, the sizes of both bases would be different to allow proper reduction in the $\mathcal{A} \rightarrow \mathcal{B}$ case, thus we exclude this option from our consideration. Table 1 shows the available and the practical combinations.

Table 1. Base Extension Algorithm Combinations

		$\mathcal{A} \rightarrow \mathcal{B}$			
		MRC (M)	**Shenoy (S)**	**Kawamura (K)**	**Bajard (B)**
$\mathcal{B} \rightarrow \mathcal{A}$	**MRC (M)**	●	○	○	●
	Shenoy (S)	●	○	○	●
	Kawamura (K)	●	○	○	●
	Bajard (B)	○	○	○	○

5.3 Point Multiplication Using Generalised Mersenne Primes

For realising the elliptic curve group operation, we chose mixed affine-Jacobian coordinates [8] to avoid costly inversions in the underlying field and thus concentrated on efficient implementation of modular multiplication, the remaining time critical operation. For this, we used a straightforward schoolbook-type multiplication combined with the efficient reduction technique for the generalised Mersenne prime presented in Algorithm 3.

As for the CIOS method, there is no intrinsic parallelism except pipelining in this approach (cf. Criterion A1). Thus, we use one thread per point multiplication. We assume the use of the same base point P per point multiplication kP and *varying* scalars k. Thus, the only input that has to be transferred are the scalars. Secondly, we transfer the result in projective Jacobian coordinates back to the host. For efficiency reasons, we encode all coordinates interleaved for each threads in a block again.

We used shared memory to store all temporary values, nailed to 28 bits to allow schoolbook multiplication without carry propagation. Thus, we need 8 words per coordinate. Point addition and doubling algorithms were inspired by `libseccure` [29]. With this approach shared memory turns out to be the limiting factor. Precisely, we require 111 words per point multiplication to store 7 temporary coordinates for point addition and modulo arithmetic, *two* points and each scalar. This results in 444 bytes of shared memory and a maximum of $\lfloor 16384/444 \rfloor = 36$ threads per multiprocessor. This leaves still room for improvements as Criterion A1 is *not* fulfilled. However, due to internal errors in the toolchain, we were not (yet) able to compile a solution that uses global memory for temporary values instead. Note that the left-to-right binary method for point multiplication demands only one temporary point. However, for the sake of a homogeneous flow of instructions we compute both possible solutions per scalar bit and use a small divergent section to decide which of them is the desired result (cf. Criterion A4).

6 Conclusion

With the previously discussed implementations on GPUs at hand, we finally need to identify the candidate providing the best performance for modular exponentiation.

6.1 Results and Applications

Before presenting the benchmarking results of the best algorithm combinations we show our results regarding the different base extension options for the RNS method. The benchmarking scheme was the following: first, we did an exhaustive search for the number of registers per thread that can principally be generated by the toolchain. Then, we benchmarked all available execution configurations for these numbers of registers. To make the base extension algorithms comparable, we would have to repeat this for all possible combinations, as shown in

Table 2. Results for different Base Extension Techniques (RNS Method)

Base Ext. $\mathcal{A} \rightarrow \mathcal{B}$	Base Ext. $\mathcal{B} \rightarrow \mathcal{A}$	**Throughput (1024 bits) [Enc/s] (rel.)**	**Throughput (2048 bits) [Enc/s] (rel.)**
M	**M**	194 (46%)	28 (50%)
B	**M**	267 (63%)	38 (67%)
B	**K**	408 (97%)	55 (98%)
B	**S**	419 (100%)	56 (100%)

Table 1. However to reduce the complexity of benchmarking, it suffices to measure all possible combinations in the first row and all possible combinations in the second column to gain figures for all available combinations. The results for the particular best configuration can be found in Table 2.

Clearly, the mixed radix based approach also used in [24] cannot compete with CRT-based solutions. Kawamura *et al.* is slower than the method of Shenoy *et al.* , but performs only slightly worse for the 2048 bit range. Figure 1 shows the time over the number of encryptions for the four cases and the 1024 bit and 2048 bit ranges, respectively.

Both graphs show the characteristic behaviour: Depending on the number of blocks that are started on the GPU and the respective execution configuration we get stair-like graphs. Only multiples of the number of warps per multiprocessor and the number of multiprocessors result in optimal configurations that fully utilise the GPU. However, depending on the number of registers per thread and the amount of shared memory used other configurations are possible and lead to smaller steps in between.

Optimised Implementations. Beside the reference implementation based on the CIOS algorithm, we selected as best choice the CRT-RNS method based on a combination of Bajard's and Shenoy's methods to compute the first and second base extension of Algorithm 2, respectively.

The selection of the implementation was primarily motivated to achieve high throughput rather than a small latency. Hence, due to the latency, not all implementations might be suitable for all practical applications. To reflect this, we present figures for data throughput as well as the initial latency t_{min} required at the beginning of a computation. Note that our results consider optimal configurations of warps per block and blocks per grid only. Table 3 shows the figures for modular exponentiation with 1024 and 2048 bit moduli and elliptic curve point multiplication using NIST's P-224 curve.

The throughput is determined from the number of encryptions divided by the elapsed time. Note that this *includes* the initial latency t_{min} at the beginning of the computations. The corresponding graphs are depicted in Figure 2. Note the relatively long plateau when using the CIOS technique. It is a direct result from having coarse-grained parallelism only: the smallest number of encryptions that can be processed is 128 times higher than for the RNS method. Its high offset is due to storing temporary values in global memory: memory access latency is

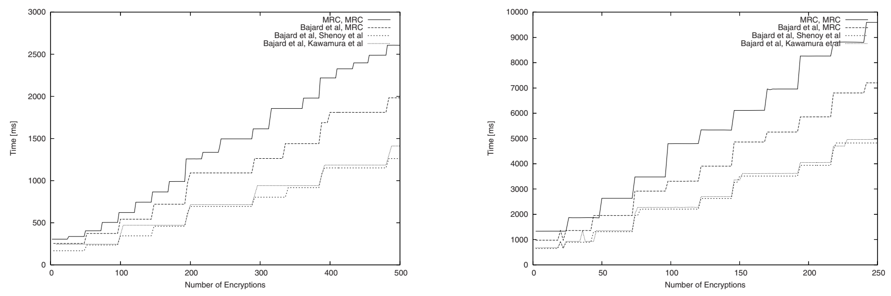

Fig. 1. Results For Modular Exponentiation with about 1024 (left) and 2048 bit (right) Moduli For Different Base Extension Methods, based on a Nvidia 8800 GTS Graphics Card

Table 3. Results for Throughput and Minimum Latency t_{min} on a Nvidia 8800 GTS Graphics Card

Technique	Throughput [OPs/s]	[ms/OP]	Latency t_{min} [ms]	OPs at t_{min}
ModExp-1024 CIOS	813.0	1.2	6930	1024
ModExp-1024 RNS	439.8	2.3	144	4
ModExp-2048 CIOS	104.3	9.6	55184	1536
ModExp-2048 RNS	57.9	17.3	849	4
ECC PointMul-224	1412.6	0.7	305	36

hidden by scheduling independent computations, however the time needed to fetch/store the first value in each group cannot be hidden.

Clearly, the CIOS method delivers the highest throughput at the price of a high initial latency. For interactive applications such as online banking using TLS this will be a major obstacle. However, non-interactive applications like a *certificate authority* (CA) might benefit from the raw throughput[3]. Note that both applications will share the *same* secret key for all digital signatures when using RSA. In case of ECC (ECDSA) however, *different* exponents were taken into account.

The residue number system based approach does only feature roughly half of the throughput but provides a more immediate data response. Thus, this method seems to be suitable even in interactive applications. Last but not least elliptic curve cryptography clearly outperforms modular exponentiation based techniques not only due to the much smaller parameters. With respect to other hardware and software implementations compared against our results in the next section, we present an ECC solution which outperforms most hardware devices and comes close the the performance of recent dual-core microprocessors.

[3] Also consider the top model of Nvidia's next series of GPUs, the GeForce 9800GX2, that can be used in a four-card setup.

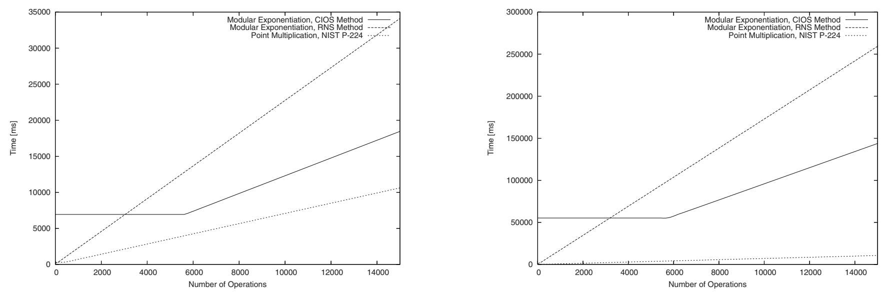

Fig. 2. Results For Modular Exponentiation with about 1024 (left) and 2048 bit (right) Moduli and Elliptic Curve Point Multiplication on NIST's P-224 Curve, based on a Nvidia 8800 GTS Graphics Card

6.2 Comparison with Previous Implementations

Due to the novelty of general purpose computations on GPUs and since directly comparable results are rare, we will take reference to recent hardware and software implementations in literature as well. To give a feeling for the different GPU generations we include Table 4.

Table 4. Comparison of Nvidia GPU platforms

GPU	**Shader clock [MHz]**	**Shaders**	**Fill Rate [GPixels/s]**	**Mem Bandwidth** [GB/s]	CUDA
7800GTX			13.2	54.4	no
8800GTS	1200	92	24.0	64.0	yes
8800GTX	1350	128	36.8	86.4	yes
9800GX2	1500	$2 \cdot 128$	76.8	128.0	future

Moss *et al.* implemented modular exponentiation for 1024 bit moduli on Nvidia's 7800GTX GPU [24], using the same RNS approach but picking different base extension mechanisms. The authors present the *maximum* throughput only that has been achieved at the cost of an unspecified but high latency. Fleissner's recent analysis on modular exponentiation for GPUs is based on 192 bit moduli but relates the GPU performance solely to the CPU of his host system.

Costigan and Scott implemented modular exponentiation on IBM's Cell platform, i.e., a Sony Playstation 3 and an IBM MPM blade server, both running at 3.2 GHz [10]. We only quote the best figures for the Playstation 3 as they call the results for the MPM blade preliminary. The Playstation features one PowerPC core (PPU) and 6 *Synergistic Processing Elements* (SPUs). Software results have been attained from ECRYPT's eBATS project [11]. Here, we picked a recent Intel Core2 Duo with 2.13 GHz clock frequency. Since mostly all figures for software relate to cycles, we assumed that repeated computations can be

Table 5. Comparison of our designs to results from literature. The higher throughput values the better. ModExp-i denotes modular exponentiation using an i-bit modulus. PointMul-i denotes point multiplication on elliptic curves over $\mathbb{F}_p$, where p is a i-bit prime. Results that used the Chinese remainder theorem are marked with "CRT".

Reference	**Platform & Technique**	**Throughput [ModExps/s] and [PointMuls/s]**						
		ModExp-1024	ModExp-1024, CRT	ModExp-2048	ModExp-2048, CRT	ECC PointMul-160	ECC PointMul-224	ECC PointMul-256
Our Design	Nvidia 8800GTS GPU, CIOS algorithm	**813.0**		**104.3**				
	Nvidia 8800GTS GPU, RNS arithmetic	439.8		57.9				
	Nvidia 8800GTS GPU, ECC NIST-224						1412.6	
[24] Moss	Nvidia 7800GTX GPU, RNS arithmetic	175.4						
[10] Costigan	Sony Playstation 3, 1 PPU, 6 SPUs		909.2		**401.4**			
[22] Mentens	Xilinx xc2vp30 FPGA	471.7	1724.1		235.8	1000.0		440.5
[32] Schinianakis	Xilinx xc2vp125 FPGA, RNS arithmetic					413.9		
[36] Suzuki	Xilinx xc4fx12 FPGA, using DSPs	584.8		79.4				
[26] Nozaki	0.25μm CMOS, 80 MHz, 221k GE	238.1		34.2				
[11] eBATS	Intel Core2 2.13 GHz		1447.5		300.4	**2623.4** [a]	**1868.5**[a]	1494.8[a]
[15] Gaudry	Intel Core2 2.66 GHz							**6900**[b]

[a] Performance for ECDSA operation including additional modular inversion and multiplication operation.
[b] Special elliptic curve in Montgomery form, non-compliant to ECC standardised by NIST.

performed *without* interruption on *all* available cores so that no further cycles are spent, e.g., on scheduling or other administrative tasks. Note that this is a very optimistic assumption possibly overrating the performance of microprocessors with respect to actual applications. We also compare our work to the very fast software implementation by [15] on an Intel Core2 system at 2.66 GHz but which uses the special Montgomery and non-standard curve over $\mathbb{F}_{2^{255}-19}$.

To the best of our knowledge, Mentens published the best results for public key cryptography on reconfigurable hardware so far [22]. She used a Field Programmable Gate Array (FPGA) of Xilinx' Virtex-II Pro family, namely the xc2vp30-7FF1152. Schinianakis *et al.* implemented elliptic curve cryptography on the same family of FPGAs but using RNS arithmetic for the underlying field [32]. Suzuki implemented the modular exponentiation on FPGAs taking advantage of the included digital signal processors (DSPs) on a board from Xilinx' Virtex 4 FX family [36].

Nozaki *et al.* designed an RSA circuit in 0.25 μm CMOS technology, that needs 221k gate equivalents (GE) [26] and uses RNS arithmetic with Kawamura's base extension mechanism.

6.3 Further Work

Elliptic curves in Hessian form feature highly homogeneous formulae to compute all three projective coordinates in point additions [19,34]. However, the curves standardised by ANSI and NIST *cannot* be transformed to Hessian form. Furthermore, point doublings can be converted to point additions by simple coordinate rotations. Thus, it is possible to compute point doublings and additions for all three coordinates in parallel. A future study will show the applicability to graphics hardware.

References

1. Advanced Micro Devices, Inc. (AMD), Sunnyvale, CA, USA. ATI CTM Guide, Release 1.01 (2006)
2. American National Standards Institute (ANSI). Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA) (ANSI X9.62:2005) (2005)
3. Bajard, J.-C., Didier, L.-S., Kornerup, P.: Modular multiplication and base extension in residue number systems. In: Burgess, N. (ed.) Proceedings ARITH15, the 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA, pp. 59–65 (June 2001)
4. Bajard, J.-C., Meloni, N., Plantard, T.: Efficient RNS bases for cryptography. In: Proceedings of IMACS 2005 World Congress, Paris, France (July 2005)
5. Bajard, J.-C., Plantard, T.: RNS bases and conversions. Advanced Signal Processing Algorithms, Architectures, and Implementations XIV 5559(1), 60–69 (2004)
6. Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro 16(3), 26–33 (1996)
7. Koç, Ç.K., Naccache, D., Paar, C. (eds.): CHES 2001. LNCS, vol. 2162. Springer, Heidelberg (2001)
8. Cohen, H., Frey, G. (eds.): Handbook of elliptic and hyperelliptic curve cryptography. Chapman & Hall/CRC Press, Boca Raton (2005)
9. Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: CryptoGraphics: Secret key cryptography using graphics cards. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376. Springer, Heidelberg (2005)
10. Costigan, N., Scott, M.: Accelerating SSL using the vector processors in IBM's Cell broadband engine for Sony's Playstation 3. In: SPEED 2007 Workshop Record [12] (2007), `http://www.hyperelliptic.org/SPEED/`
11. ECRYPT. eBATS: ECRYPT benchmarking of asymmetric systems. Technical report (2007), `http://www.ecrypt.eu.org/ebats/`
12. ECRYPT European Network of Excellence in Cryptography. Software Performance Enhancement for Encryption and Decryption (SPEED), 2007 Workshop Record, Amsterdam, The Netherlands (June 2007), `http://www.hyperelliptic.org/SPEED/`
13. Fan, J., Skiyama, K., Verbauwhede, I.: Montgomery modular multiplication algorithm for multi-core systems. In: SPEED 2007 Workshop Record [12] (2007), `http://www.hyperelliptic.org/SPEED/`
14. Fleissner, S.: GPU-accelerated Montgomery exponentiation. In: Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2007. LNCS, vol. 4487, pp. 213–220. Springer, Heidelberg (2007)
15. Gaudry, P., Thomé, E.: The mpF$_q$ library and implementing curve-based key exchanges. In: SPEED 2007 Workshop Record [12], pp. 49–64 (2007), `http://www.hyperelliptic.org/SPEED/`
16. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer, New York (2003)
17. Harris, M.: Optimizing CUDA. In: Supercomputing 2007 Tutorial, Reno, NV, USA (November 2007)

18. Harrison, O., Waldron, J.: AES encryption implementation and analysis on commodity graphics processing unit. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 209–226. Springer, Heidelberg (2007)
19. Hisil, H., Carter, G., Dawson, E.: Faster group operations on special elliptic curves. Cryptology ePrint Archive, Report 2007/441 (2007), `http://eprint.iacr.org/`
20. Kawamura, S., Koike, M., Sano, F., Shimbo, A.: Cox-rower architecture for fast parallel Montgomery multiplication. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 523–538. Springer, Heidelberg (2000)
21. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: Proceedings of IEEE's International Conference on Signal Processing and Communication ICSPC 2007, pp. 65–68 (November 2007)
22. Mentens, N.: Secure and Efficient Coprocessor Design for Cryptographic Applications on FPGAs. PhD thesis, Katholieke Universiteit Leuven, Leuven-Heverlee, Belgium (June 2007)
23. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
24. Moss, A., Page, D., Smart, N.: Toward acceleration of RSA using 3d graphics hardware. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 369–388. Springer, Heidelberg (2007)
25. National Institute of Standards and Technology (NIST). Digital signature standard (DSS) (FIPS 186-2) (January 2000)
26. Nozaki, H., Motoyama, M., Shimbo, A., Kawamura, S.: Implementation of RSA algorithm based on RNS Montgomery multiplication. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 364–376. Springer, Heidelberg (2001)
27. Nvidia Corporation, Santa Clara, CA, USA. Compute Unified Device Architecture (CUDA) Programming Guide, Version 1.0 (June 2007)
28. Nvidia Corporation, Santa Clara, CA, USA. Parallel Thread Execution (PTX) ISA Version 1.0, Release 1.0 (June 2007)
29. Poettering, B.: seccure – SECCURE elliptic curve crypto utility for reliable encryption, version 0.3 (August 2006), `http://point-at-infinity.org/seccure/`
30. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public key cryptosystems. In: Communications of the ACM, vol. 21, pp. 120–126 (February 1978)
31. Rosenberg, U.: Using graphic processing unit in block cipher calculations. Master's thesis, University of Tartu, Tartu, Estonia (2007)
32. Schinianakis, D.M., Kakarountas, A.P., Stouraitis, T.: A new approach to elliptic curve cryptography: an RNS architecture. In: Proceedings of IEEE's 14th Mediterranian Electrotechnical Conference (MELECON 2006), pp. 1241–1245 (May 2006)
33. Shenoy, A.P., Kumaresan, R.: Fast base extension using a redundant modulus in RNS. In: IEEE Transactions on Computers, vol. 38, pp. 292–297 (February 1989)
34. Smart, N.P.: The Hessian form of an elliptic curve. In: Koç, Ç.K., et al. (eds.) [7], pp. 118–125
35. Stinson, D.R.: Cryptography. Theory and Practice, 3rd edn. Taylor & Francis, Abington (2005)
36. Suzuki, D.: How to maximize the potential of FPGA resources for modular exponentiation. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 272–288. Springer, Heidelberg (2007)
37. Szabó, N.S., Tanaka, R.I.: Residue Arithmetic and its Applications to Computer Technology. McGraw-Hill Inc., USA (1967)

A Appendix

Table 6. Modulus Sizes for Modular Multiplication Using RNS

1st Base Ext.	**2nd Base Ext.**	**1024 bit range**	**2048 bit range**
Bajard *et al.*	Shenoy *et al.*	981	2003
	Others	1013	2035
Others	Shenoy *et al.*	990	2014
	Others	1022	2046