Noname manuscript No.
(will be inserted by the editor)

Low-Latency Elliptic Curve Scalar Multiplication

Joppe W. Bos

Received: date / Accepted: date

Abstract This paper presents a low-latency algorithm designed for parallel computer architectures to compute the scalar multiplication of elliptic curve points based on approaches from cryptographic side-channel analysis. A graphics processing unit implementation using a standardized elliptic curve over a 224-bit prime field, complying with the new 112-bit security level, computes the scalar multiplication in 1.9 milliseconds on the NVIDIA GTX 500 architecture family. The presented methods and implementation considerations can be applied to any parallel 32-bit architecture.

Keywords Elliptic Curve Cryptography · Elliptic Curve Scalar Multiplication · Parallel Computing · Low-Latency Algorithm

1 Introduction

Elliptic curve cryptography (ECC) [30,37] is an approach to public-key cryptography which enjoys increasing popularity since its invention in the mid 1980s. The attractiveness of small key-sizes [32] has placed this public-key cryptosystem as the preferred alternative to the widely used RSA public-key cryptosystem [48]. This is emphasized by the current migration away from 80-bit to 128-bit security where, for instance, the United States' National Security Agency restricts the use of public key cryptography in "Suite B" [40] to ECC. The National Institute of Standards and Technology (NIST) is more flexible and settles for 112-bit security at the lowest level from the year 2011 on [52].

At the same time there is a shift in architecture design towards many-core processors [47]. Following both these trends, we evaluate the performance of a

Laboratory for Cryptologic Algorithms
École Polytechnique Fédérale de Lausanne (EPFL)
Station 14, CH-1015 Lausanne, Switzerland
E-mail: joppe.bos@epfl.ch

standardized elliptic curve over a 224-bit prime field [55], which complies with the new 112-bit security level, on a number of different graphics processing unit (GPU) architecture families. In order to assess the possibility to use the GPU as a cryptographic accelerator we present algorithms to compute the elliptic curve scalar multiplication (ECSM), the core building block in ECC, which are designed from a low-latency point of view for parallel computer architectures.

Our approach differs from the previous reports implementing ECC schemes on GPUs [5,3,53,2] in that we divide the elliptic curve arithmetic over multiple threads instead of dividing a single finite field operation over the available resources. The presented algorithms are based on methods originating in cryptographic side-channel analysis [31] and are designed for a parallel computer architecture with a 32-bit instruction set. This makes the new generation of NVIDIA GPUs, the GTX 400/500 series known as *Fermi*, an ideal target platform. Despite the fact that our algorithms are not particularly optimized for the older generation GPUs, we show that this approach outperforms, in terms of low-latency, the results reported in the literature on these previous generations GPUs while it at the same time sustains a high throughput. On the newer Fermi architecture the ECSM can be computed in less than 1.9 milliseconds with the additional advantage that the implementation can be made to run in constant time; i.e. resistant against timing attacks.

There is an active area in the cryptologic community which reports how to adopt frequently used cryptographic and cryptanalytic algorithms efficiently to the GPU architecture. These papers report performance benchmark results to demonstrate that the GPU architecture can already be used as a cryptologic workhorse. Previous published GPU implementations cover asymmetric cryptography, such as RSA [39,53,24], and ECC [53,2], and symmetric cryptography [35,22,56,23,45,12]. The GPU has also been considered to enhance the performance of cryptanalytic computations in the setting of finding hash collisions [8], integer factorization [5,3] and computing elliptic curve discrete logarithms [4].

Parts of this work were pre-published as [10] where the Cell broadband engine architecture is considered. This paper is organized as follows. Section 2 recalls the required information related to elliptic curves and presents the standard elliptic curve used throughout the paper. Section 3 introduces the GPU platform and explains which programming paradigm we are using. Section 4 explains the rationale behind the design choices and presents the parallel algorithm to compute the ECSM. Section 5 shows and compares our benchmarks results with other GPU implementations in the literature. Section 6 concludes the paper.

2 Elliptic Curves

Let $p > 3$ be a prime, then any $a, b \in \mathbf{F}_p$ such that $4a^3 + 27b^2 \neq 0$ define an elliptic curve $E_{a,b}$ over the finite field $\mathbf{F}_p$. The zero point $\mathfrak{o}$, the so-called point

Algorithm 1 The radix-r schoolbook multiplication method.

Input: $\left\{ A = \sum_{i=0}^{n-1} a_i r^i, B = \sum_{i=0}^{n-1} b_i r^i \text{ with } 0 \leq a_i, b_i < r \right.$

Output: $\left\{ C = A \cdot B = \sum_{i=0}^{2n-1} c_i r^i \text{ with } 0 \leq c_i < r \right.$

1. $C \leftarrow A \cdot b_0$
2. **for** $i = 1$ to $n - 1$ **do**
3. $\quad C \leftarrow C + r^i(A \cdot b_i)$
4. **end for**
5. **return** C

at infinity, together with the set of points $(x, y) \in \mathbf{F}_p \times \mathbf{F}_p$ which satisfy the short affine Weierstrass equation

$$y^2 = x^3 + ax + b, \tag{1}$$

form an abelian group $E_{a,b}(\mathbf{F}_p)$ [50]. For $\mathfrak{a} \in E_{a,b}(\mathbf{F}_p)$, the additively written group law is defined as follows. Define $\mathfrak{a} + \mathfrak{o} = \mathfrak{o} + \mathfrak{a} = \mathfrak{a}$. For non-zero $\mathfrak{a} = (x_1, y_1), \mathfrak{b} = (x_2, y_2) \in E_{a,b}(\mathbf{F}_p)$ define $\mathfrak{a} + \mathfrak{b} = \mathfrak{o}$ iff $x_1 = x_2$ and $y_1 = -y_2$. Otherwise $\mathfrak{a} + \mathfrak{b} = (x_3, y_3)$ with

$$\begin{array}{l} x_3 = \lambda^2 - x_1 - x_2 \\ y_3 = \lambda(x_1 - x_3) - y_1 \end{array}, \quad \lambda = \begin{cases} \dfrac{3x_1^2 + a}{2y_1} \text{ if } x_1 = x_2 \\ \dfrac{y_1 - y_2}{x_1 - x_2} \text{ otherwise.} \end{cases} \tag{2}$$

Currently, the fastest known elliptic curves, in terms of multiplications in $\mathbf{F}_p$, are the family of curves originating from a normal form for elliptic curves introduced by Edwards in 2007 [16] and generalized and proposed for usage in cryptology by Bernstein and Lange [6]. The most efficient curve arithmetic on these type of curves is due to Hisil et al. [25].

2.1 Standard Elliptic Curves

One way to speed-up elliptic curve arithmetic is to enhance the performance of the finite field arithmetic by using a prime of a special form. The structure of such a prime is exploited by constructing a fast reduction method, applicable to this prime only. Typically, multiplication and reduction are performed in two sequential phases. For the multiplication phase we consider the so-called schoolbook, or textbook, multiplication technique (see Algorithm 1 for a high-level description). Other asymptotically faster approaches, such as Karatsuba multiplication [28], are not considered because they are not expected to give a speed-up for the size of integers we target.

In the FIPS 186-3 standard [55], NIST recommends the use of five prime fields when using the elliptic curve digital signature algorithm. The sizes of

Algorithm 2 Fast reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$.

Input: Integer $c = (c_{13}, \ldots, c_1, c_0)$, each c_i is a 32-bit word, and $0 \leq c < p_{224}^2$.
Output: Integer $d \equiv c \bmod p_{224}$.
Define 224-bit integers:
$s_1 \leftarrow (\ c_6,\ c_5,\ c_4,\ c_3,\ c_2,\ c_1,\ c_0)$,
$s_2 \leftarrow (\ c_{10},\ c_9,\ c_8,\ c_7,\ 0,\ 0,\ 0)$,
$s_3 \leftarrow (\ 0,\ c_{13},\ c_{12},\ c_{11},\ 0,\ 0,\ 0)$,
$s_4 \leftarrow (\ c_{13},\ c_{12},\ c_{11},\ c_{10},\ c_9,\ c_8,\ c_7)$,
$s_5 \leftarrow (\ 0,\ 0,\ 0,\ 0,\ c_{13},\ c_{12},\ c_{11})$
return $(\mathrm{d} = s_1 + s_2 + s_3 - s_4 - s_5)$;

the five recommended primes by NIST are 192, 224, 256, 384 and 521 bits. In this work we consider the 224-bit prime

$$p_{224} = 2^{224} - 2^{96} + 1.$$

The prime p_{224}, together with the provided curve parameters from the FIPS 186-3, allows one to use 224-bit ECC which provides a 112-bit security level. This is the lowest strength for asymmetric cryptographic systems allowed by NIST's "SP 800-57 (1)" [52] standard from the year 2011 on (cf. [11] for a discussion about the migration to these new standards).

Reduction modulo p_{224} can be done efficiently: for $x \in \mathbf{Z}$ with $0 \leq x < (2^{224})^2$ and $x = x_L + 2^{224} x_H$ for $x_L, x_H \in \mathbf{Z}, 0 \leq x_L, x_H < 2^{224}$, define

$$\mathfrak{R}(x) = x_L + x_H(2^{96} - 1).$$

It follows that $\mathfrak{R}(x) \equiv x \bmod p_{224}$ and $\mathfrak{R}(x) \leq 2^{320} - 2^{96}$. Algorithm 2 shows the application of $\mathfrak{R}(\mathfrak{R}(x))$ for a machine word (limb) size of 32 bits, based on the work by Solinas [51]. Note that the resulting value $\mathfrak{R}(\mathfrak{R}(x)) \equiv x \bmod p_{224}$ with $-(2^{224} + 2^{96}) < \mathfrak{R}(\mathfrak{R}(x)) < 2^{225} + 2^{192}$.

The NIST curves over prime fields all have prime order. In order to translate a Weierstrass curve into a suitable Edwards curve over the same prime field, to use the faster elliptic curve arithmetic, the curve needs to have a group element of order four [6] (which is not the case with the prime order NIST curves). To comply with the NIST standard we chose not to use Edwards but Weierstrass curves.

An extensive study of a software implementation of the NIST-recommended elliptic curves over prime fields on the x86 architecture is given by Brown et al. [14].

2.2 Elliptic Curve Scalar Multiplication

Repeated elliptic curve point addition is called elliptic curve scalar multiplication:

$$nP := \underbrace{P + P + \ldots + P}_{n \text{ times}}$$

Algorithm 3 The double-and-add algorithm (left) and the Montgomery ladder (right).

Input: $\begin{cases} G \in E_{a,b}(\mathbf{F}_p), \\ s \in \mathbf{Z}_{>0},\ 2^{k-1} \leq s < 2^k, \\ s = \sum_{i=0}^{k-1} s_i 2^i,\ \text{with } 0 \leq s_i < 2 \end{cases}$
Output: $P = sG \in E_{a,b}(\mathbf{F}_p)$
1. $P \leftarrow G$
2. **for** $i = k-2$ down to 0 **do**
3. $\quad P \leftarrow 2P$
4. $\quad$ **if** $s_i = 1$ **then**
5. $\quad\quad P \leftarrow P + G$
6. $\quad$ **end if**
7. **end for**

Input: $\begin{cases} G \in E_{a,b}(\mathbf{F}_p), \\ n = \sum_{i=0}^{k-1} n_i 2^i, n \in \mathbf{Z}_{>0}, \\ 2^{k-1} \leq n < 2^k \end{cases}$
Output: $P = nG \in E_{a,b}(\mathbf{F}_p)$
1. $P \leftarrow G, Q \leftarrow G$
2. **for** $i = k-2$ down to 0 **do**
3. $\quad$ **if** $n_i = 1$ **then**
4. $\quad\quad (P, Q) \leftarrow (P + Q, 2Q)$
5. $\quad$ **else**
6. $\quad\quad (P, Q) \leftarrow (2P, P + Q)$
7. $\quad$ **end if**
8. **end for**

for $n \in \mathbf{Z}_{>0}$ and is the main operation in many elliptic curve based cryptographic systems. Prime examples are the elliptic curve digital signature algorithm as specified in [55] and elliptic curve Diffie-Hellman for key agreement (see [54]). Improving the performance of the ECSM translates directly in faster ECC schemes in practice. A survey of the different techniques and approaches together with the cost to compute the ECSM is given by Cohen, Miyaji and Ono [15]. An updated overview from 2008, incorporating the latest curve representations and algorithms, is given by Bernstein and Lange [7].

Let us recall one particular approach, known as the *Montgomery ladder*: this technique was introduced by Montgomery in [38] in the setting of integer factorization using the elliptic curve method (ECM) [33]. We give the higher level description from [27]. Let $L_0 = s = \sum_{i=0}^{t-1} k_i 2^i$, define $L_j = \sum_{i=j}^{t-1} k_i 2^{i-j}$ and $H_j = L_j + 1$. Then,

$$L_j = 2L_{j+1} + k_j = L_{j+1} + H_{j+1} + k_j - 1 = 2H_{j+1} + k_j - 2.$$

One can update these two values using

$$(L_j, H_j) = \begin{cases} (2L_{j+1}, L_{j+1} + H_{j+1}) \text{ if } k_j = 0, \\ (L_{j+1} + H_{j+1}, 2H_{j+1}) \text{ if } k_j = 1. \end{cases}$$

An overview of this approach is given in the right part of Algorithm 3. This approach is slower compared to, for instance, the double-and-add technique (the left part of Algorithm 3) since a duplication and addition are always performed per bit. This disadvantage is actually used as a feature in environments which are exposed to side-channel attacks and where the algorithms need to process the exact same sequence of steps independent of the input parameters. It is not difficult to alter Algorithm 3 such that it becomes branch-free (using the bit n_i to select which point to double). In ECM the elliptic curve scalar multiplication is calculated using the Montgomery form which avoids computing on the y-coordinate. This is achieved as follows: given the x- and

z-coordinate of the points P, Q and $P-Q$ one can compute the x- and z-coordinates of $P+Q$ (and similarly $2P$ or $2Q$). Avoiding computations on one of the coordinates results in a speedup in practice (see [38] for all the details).

3 Compute Unified Device Architecture

Graphics Processing Units (GPUs) have mainly been game- and video-centric devices. Due to the increasing computational requirements of graphics-processing applications, GPUs have become very powerful parallel processors and this, moreover, incited research interest in computing outside the graphics-community. Until recently, programming GPUs was limited to graphics libraries such as OpenGL [49] and Direct3D [9], and for many applications, especially those based on integer-arithmetic, the performance improvements over CPUs was minimal, sometimes even degrading. The release of NVIDIA's G80 series and ATI's HD2000 series GPUs (which implemented the unified shader architecture), along with the companies' release of higher-level language support with Compute Unified Device Architecture (CUDA), Close to Metal (CTM) [46] and the more recent Open Computing Language (OpenCL) [21] facilitate the development of massively-parallel general purpose applications for GPUs [43, 1]. These general purpose GPUs have become a common target for numerically-intensive applications given their ease of programming (relative to previous generation GPUs), and ability to outperform CPUs in data-parallel applications, commonly by orders of magnitude.

We focus on NVIDIA's GPU architecture with CUDA, more specifically the third generation GPU family known under the code name *Fermi* [42]. After the first generation G80 architecture, the first GPU to support the C-programming language, and the second generation GT200 architecture, the Fermi architecture was released in 2010. One of the main features for our setting is the support of $32 \times 32 \rightarrow 32$-bit multiplication instructions, for both the least- and most-significant 32-bit of the multiplication result. The previous NVIDIA architecture families have native $24 \times 24 \rightarrow 32$-bit multiplication instructions (for the least significant 32-bit of the result).

We briefly recall some of the basic components of NVIDIA GPUs. More detailed information about the specification of CUDA as well as experiences using this parallel computer architecture can be found in [42, 43, 41, 18, 34]. Each GPU contains a number of streaming multiprocessors (SMs) and each SM consists of multiple scalar processor cores (SP); these number vary per graphics card. Typically, on the Fermi architecture, there are 32 SPs per SM and around 16 SMs per GPU. C for CUDA is an extension to the C language that employs the massively parallel programming model called *single-instruction multiple-thread*. The programmer defines *kernel functions*, which are compiled for and executed on the SPs of each SM, in parallel: each light-weight thread executes the same code, operating on different data. A number of threads are grouped into a *thread block* which is scheduled on a single SM, the threads of which time-share the SPs. Hence, typically the number of launched threads is a multiple

of the available SPs (cores) of the GPU. This hierarchy provides for threads within the same block to communicate using the on-chip shared memory and to synchronize their execution using barriers (a synchronization method which causes threads to wait until all threads reach a certain point).

On a lower level, threads inside each thread block are executed in groups of 32 called *warps*. On the Fermi architecture each SM has two warp schedulers and two instruction dispatch units. This means that two instructions, from separate warps, can be scheduled and dispatched at the same time. By switching between the different warps, trying to fill the pipeline as much as possible, a high throughput rate can be sustained. When the code executed on the SP contains a conditional data-dependent branch all possibilities, taken by the threads inside this warp, are serially executed (threads which do not follow a certain branch are disabled). After executing these possibilities the threads within this warp continue with the same code execution. For optimal performance it is recommended to avoid multiple execution paths within a single warp.

The GPU has a large amount of global memory. Global memory is shared among all threads running on the GPU (on all SMs). Communication between threads inside a single thread block can be performed using the faster shared memory. If a warp requests data from global memory, the request is split into two separate memory requests, one for each half-warp (16 threads), each of which is issued independently. If the word size of the memory requested is 4, 8, or 16 bytes, the data requested by all threads lie in the same segment and are accessed in sequence (the k^{th} thread in the half-warp fetches the k^{th} word) then the global memory request is *coalesced*. In practice this means that a 64-byte memory transaction, a 128-byte memory transaction, or two 128-byte memory transactions are issued if the size of the words accessed by the threads is 4, 8, or 16 bytes, respectively. When this transfer is not coalesced, 16 separate 32-byte memory transactions are performed. More advanced rules might apply to decide if a global memory request can be coalesced or not depending on the architecture used, see [43] for the specific details.

4 Parallel Algorithms

In this section we present low-latency algorithms to compute the ECSM designed for architectures capable of running multiple computational streams in parallel together with implementation considerations to speed up the modular arithmetic. Our algorithms are especially designed for platforms which have a 32-bit instruction set, e.g. addition, subtraction and multiplication instructions with 32-bit in- and output. The (modular) multiplication operations in this chapter are designed to operate on relatively small (224-bit) integers. On the widely available x86 and x86-64 architectures the threshold for switching from schoolbook multiplication to methods with a lower asymptotic run-time complexity (e.g. Karatsuba multiplication) is > 800 bits [20] (but this threshold depends on the word-size of the architecture and the various latencies and

throughput of the addition and multiplication instructions). In our 224-bit setting we have investigated schoolbook multiplication only.

A common approach to obtain low-latency arithmetic is to compute the modular multiplications with multiple threads using a residue number system (RNS) [19,36]. This might be one of the few available options to lower the latency for schemes which perform a sequence of data-dependent modular multiplications, such as in RSA where the main operation is modular exponentiation, but different approaches can be tried in the setting of elliptic curve arithmetic. We follow the ideas from [27,25] and choose, in contrast to for instance [2], to let a single thread compute a single modular multiplication. The parallelism is exploited at the elliptic curve arithmetic level where multiple instances of the finite field arithmetic are computed in parallel to implement the elliptic curve group operation. On a highly parallel architecture, like the GPU, multiple instances of our parallel ECSM algorithm are dispatched in order to take full advantage of all the available cores.

4.1 Finite Field Arithmetic

In order to speed up the modular calculations we represent the integers using a redundant representation. Instead of fully reducing to the range $[0, p_{224}\rangle$ we use the slightly larger interval $[0, 2^{224}\rangle$. This redundant representation saves a multi-limb comparison to detect if we need to perform an additional subtraction after a number has been reduced to $[0, 2^{224}\rangle$. Using this representation, reduction can be done more efficiently, as outlined in this section, while it does not require more 32-bit limbs (or registers) to store the integers. Various operations need to be adapted in order to handle the boundary cases, this is outlined in this section.

4.1.1 Modular Addition and Subtraction

After an addition of a and b, with $0 \leq a, b < 2^{224}$, resulting in $a + b = c = c_H 2^{224} + c_L$, with $0 \leq c_L < 2^{224}$ and $0 \leq c_H \leq 1$, there are different strategies to compute the modular reduction of c. One can subtract p_{224} once ($c_H = 1$) or the result is already in the proper interval ($c_H = 0$) and subsequently continue using the 224 least significant bits of the outcome. In order to prevent divergent code on parallel computer architectures the value $c_H p_{224}$ (either 0 or p_{224}) could be pre-computed and subtracted after the addition of a and b. Note that an additional subtraction might be required in the unlikely event that $c_H = 1$ and $a + b - p_{224} \geq 2^{224}$.

A faster approach is to use the special form of the prime p_{224}. Using

$$c = c_H 2^{224} + c_L \equiv c_L + c_H(2^{96} - 1) \bmod p_{224} \tag{3}$$

requires the computation of one addition of a and b and one addition with the pre-computed constant $c_H(2^{96} - 1)$, for $c_H \in \{0, 1\}$. Again, in the unlikely event that $c_H = 1$ and $c_L + 2^{96} - 1 \geq 2^{224}$ an additional subtraction is

Algorithm 4 Radix-2^r schoolbook multiplication algorithm for architectures which have a multiply-and-add instruction. We use $r = 32$ for the GPU architecture.

Input: Integers $a = \sum_{i=0}^{n-1} a_i 2^{ri},\ b = \sum_{i=0}^{n-1} b_i 2^{ri},$ with $0 \le a_i, b_i < 2^r$.

Output: Integer $c = a \cdot b = \sum_{i=0}^{2n-1} c_i 2^{ri},$ with $0 \le c_i < 2^r$.

1. $d_i \leftarrow 0, \quad i \in [0, n-1]$
2. **for** $j = 0$ to $n-1$ **do**
3. $\quad (e, D_j) \leftarrow \texttt{split}(a_0 \cdot b_j + d_0)$
4. $\quad$ **for** $i = 1$ to $n-1$ **do**
5. $\quad\quad (e, d_{i-1}) \leftarrow \texttt{split}(a_i \cdot b_j + e + d_i)$
6. $\quad$ **end for**
7. $\quad d_{n-1} \leftarrow e$
8. **end for**
9. **return** $(c \leftarrow (d_{n-1}, d_{n-2}, \ldots, d_0, D_{n-1}, D_{n-2}, \ldots, D_0))$

required. The probability to obtain a carry after adding the fourth 32-bit limb of $2^{96} - 1$ to c_L is so small that an early-abort strategy can be applied; i.e. all concurrent threads within the same warp assume that no carry is produced and continue executing the subsequent code. In the unlikely event that one or more of the threads produce a carry this results in divergent code and the other threads in this warp remain idle until the computation has been completed. This strategy decreases the number of instructions required to implement the modular addition and makes this approach preferable in practice.

For modular subtraction the same two approaches can be applied. In the first approach the modulus p_{224} is added to $c = a - b$ if there is a borrow: i.e. $b > a$. An additional addition of p_{224} might be required since $0 > p_{224} - 2^{224} < a - b + p_{224}$. In the second approach $2p_{224} + a$ is computed before subtracting b, to ensure that the result is positive. Next, we proceed as in the addition scenario with the only difference that $c_H \in \{0, 1, 2\}$.

4.1.2 Multiplication

Algorithm 4 depicts schoolbook multiplication designed to run on parallel architectures and is optimized for architectures with a native multiply-and-add instruction. After trivially unrolling the for-loops the algorithm is branch-free. Algorithm 4 splits the operands in r-bit words and takes advantage of the r-bit multiplier assumed to be available on the target platform. We use $r = 32$ for the GPU architecture but this can be modified to work with any other word size on different architectures. After the multiply-and-add, and a possible extra addition of one r-bit word, the $2r$-bit result z is split into the r most and r least significant bits. This is denoted by $(\lfloor \frac{z}{2^r} \rfloor, z \bmod 2^r) \leftarrow \texttt{split}(z)$. Note that the computation $a_i \cdot b_j + e + d_i$ fits in two 32-bit limbs since $(2^{32} - 1)^2 + 2(2^{32} - 1) < 2^{64}$.

Algorithm 4 requires the computation of $\texttt{split}(a_0 \cdot b_j + d_0)$ for n values of j and $\texttt{split}(a_i \cdot b_j + e + d_i)$ for $n(n-1)$ pairs (i, j), where $n = 7$ for the 224-bit multiplication. On the GTX 400/500 family of GPUs, where there are $32 \times 32 \rightarrow 32$-bit multiplication instructions to get the lower and higher 32-bits and 32-bit additions with carry in and out, the former can be implemented using four and the latter using six instructions. A direct implementation of the schoolbook algorithm as presented in Algorithm 1 might result in a slightly lower instruction count, using the addition with carry in- and out, but has the disadvantage that it requires more storage (registers) compared to Algorithm 4. We implemented both methods (Algorithm 1 and 4) and found that, when benchmarking the performance on different GPU families, the more memory efficient method from Algorithm 4 is faster in practice. This is the approach we use in our final implementation.

4.1.3 Fast Reduction

The output from the fast reduction routine as outlined in Algorithm 2, and denoted by *Red*, is not in the preferred range $[0, p_{224}\rangle$ nor in the range for the redundant representation $[0, 2^{224}\rangle$; instead, $-(2^{224} + 2^{96}) < Red(a \cdot b) < 2^{225} + 2^{192}$. In order to avoid working with negative (signed) numbers we modify the algorithm slightly such that it returns $d = s_1 + s_2 + s_3 - s_4 - s_5 + 2p_{224} \equiv c = a \cdot b \bmod p_{224}$ where $2^{224} - 2^{96} < d < 2^{226} + 2^{192}$.

In order to fully reduce multiple integers simultaneously using SIMT instructions, several approaches can be applied. Obviously the reduction algorithm can be applied again. A most likely faster approach is to subtract p_{224} repeatedly until the result is in the desired range $[0, p_{224}\rangle$. Since the arithmetic is executed on parallel architectures, the repeated subtraction is calculated by masking the value appropriately before subtracting. This process needs to be performed four times, when avoiding branches (i.e. divergent code), since the largest positive integer t which results in a positive $(2^{226} + 2^{192} - 1) - t \cdot p_{224}$ is $t = 4$.

An additional performance gain is possible at the expense of some storage. Select the desired multiple of the modulus p_{224} which needs to be subtracted from a look-up table, and perform a single subtraction. Using a redundant representation in $[0, 2^{224}\rangle$ the most significant word, containing the possible carry, has to be inspected only to determine the multiple of p_{224} to subtract. Note that an extra single subtraction might be needed in the unlikely situation that the result after the subtraction is $> 2^{224}$. The partially reduced numbers can be used as input to the same modular multiplication routines and if reduction to $[0, p_{224}\rangle$ is required this can be achieved at the cost of a single conditional multi-limb subtraction.

A refinement, in terms of storage, of the previous approaches to reduce the resulting value is by generating the desired values efficiently on-the-fly. We distinguish two cases (just as when doing the modular addition in Section 4.1.1); either subtract multiples of p_{224} or $2^{96} - 1$. Selecting the correct multiple of p_{224} is illustrated in Table 1. The 32-bit unsigned limbs c_i of $t \cdot p_{224} = \sum_{i=0}^{7} c_i 2^{32i}$

Table 1 The values of the 32-bit unsigned limbs c_i of $t \cdot p_{224} = \sum_{i=0}^{7} c_i 2^{32i}$

t	$t \cdot p_{224} = \{c_7, \ldots, c_0\}$							
	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
0	0	0	0	0	0	0	0	0
1	0	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	0	0	1
2	1	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-2$	0	0	2
3	2	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-3$	0	0	3
4	3	$2^{32}-1$	$2^{32}-1$	$2^{32}-1$	$2^{32}-4$	0	0	4

for $0 \leq t < 5$ can be computed as $c_0 = t$, $c_1 = c_2 = 0$, $c_3 = 0 - t$. The values for c_4, c_5, c_6, c_7 can be efficiently constructed using masks depending on $t = 0$ or $t > 0$. When subtracting multiples $t(2^{96} - 1) = \sum_{i=0}^{3} c_i 2^{32i}$ of $2^{96} - 1$ for $0 \leq t < 5$, the constants can be computed as

$$\begin{aligned} c_0 &= 0 - t \\ c_1 = c_2 &= \begin{cases} 0, & \text{if } t = 0, \\ 2^{32} - 1, & \text{if } t > 0. \end{cases} \\ c_3 &= \begin{cases} 0, & \text{if } t = 0, \\ t - 1, & \text{if } t > 0. \end{cases} \end{aligned}$$

The conditional statements can be converted to straight line (non-divergent) code to make the algorithms more suitable for parallel computer architectures.

4.2 Elliptic curve arithmetic

A common approach to avoid the relatively costly modular inversion operation in the elliptic curve group operation (see equation (2)) is to use projective coordinates. The projective equation of the short affine Weierstrass form from equation (1) is

$$Y^2 Z = X^3 + aXZ^2 + bZ^3. \tag{4}$$

A point (x, y) is represented as $(X : Y : Z)$ such that $(x, y) = (X/Z, Y/Z)$ for non-zero Z and the neutral element is represented as $(0 : 1 : 0)$.

In our setting we are interested, given a parallel computer architecture capable of launching a number of threads $\mathfrak{T}_i$, to lower the latency of the longest running thread $T_{max} = \max_i T_i$ as opposed to the total time of all resources combined $T_{sum} = \sum_i T_i$ where T_i is the time corresponding to thread $\mathfrak{T}_i$. Since high-throughput and low-latency may be conflicting goals, one may not be able to achieve both at the same time. Our approach is designed at the elliptic curve arithmetic level for low-latency while not sacrificing the throughput too much. To accomplish this we chose to aim for a high-throughput (and longer latency) design at the finite field arithmetic level: a single thread computes a single multiplication. If one is willing to sacrifice the throughput even more one could compute the finite field arithmetic using multiple threads as well, reducing the throughput even further. The elliptic curve point addition and

Table 2 Instruction overview to compute $(P+Q, 2Q) = (\tilde{P}, \tilde{Q}) = ((\tilde{P}_x, \tilde{P}_z), (\tilde{Q}_x, \tilde{Q}_z))$ using seven threads. The bold entries are pre-computed 224-bit integers, G_x is the x-coordinate of the input point to the Montgomery ladder.

Operation		Thread 1	Thread 2	Thread 3	Thread 4	Thread 5	Thread 6	Thread 7
(1)	**mul**	$t_0 = P_x Q_z$	$t_1 = Q_x P_z$	$t_2 = P_x Q_x$	$t_3 = P_z Q_z$	$t_4 = Q_x^2$	$t_5 = Q_z^2$	$t_6 = Q_x Q_z$
(2)	**triple**	$t_7 = 3t_3$	$t_8 = 3t_5$					
(3)	**add**	$t_9 = t_0 + t_1$	$t_{10} = t_4 + t_8$					
(4)	**sub**	$t_2 = t_2 - t_7$	$t_0 = t_0 - t_1$	$t_4 = t_4 - t_8$				
(5)	**mul**	$t_9 = t_9 t_2$	$t_3 = t_3^2$	$\tilde{P}_z = t_0^2$	$t_{10} = t_{10}^2$	$t_{11} = t_6 t_5$	$t_6 = t_6 t_4$	$t_5 = t_5^2$
(6)	**mul**	$t_9 = 2t_9$	$t_3 = \mathbf{4b} t_3$	$t_0 = G_x \tilde{P}_Z$	$t_{11} = \mathbf{8b} t_{11}$	$t_5 = \mathbf{4b} t_5$	$t_6 = 4t_6$	
(7)	**add**	$t_9 = t_9 + t_3$	$\tilde{Q}_z = t_5 + t_6$					
(8)	**sub**	$\tilde{P}_x = t_9 - t_0$	$\tilde{Q}_x = t_{10} - t_{11}$					

Table 3 Performance comparison of 224-bit elliptic curve scalar multiplication on different GPU platforms. A bold platform name indicates that this platform has compute capability 2.0 (Fermi) or higher (the latest (third) GPU-architecture family). Results when utilizing the entire GPU are expressed in operations (224-bit elliptic curve scalar multiplications) per second (op/s).

Ref	Platform	#GPUs	CUDA cores per GPU	Processor clock (MHz)	Modulus Type	Modulus Bit-size	Minimum latency [ms]	Maximum throughput [op/s]
[5] (scaled)	8800 GTS	1	96	1200	generic	280	-	3 018
	GTX 280	1	240	1296	generic	280	-	11 417
	GTX 295	2	240	1242	generic	280	-	21 103
[3]	GTX 295	2	240	1242	generic	210	-	259 534
[53]	8800 GTS	1	96	1200	special	224	305.0	1 413
[2]	8800 GTS	1	96	1200	special	224	30.3	3 138
	GTX 285	1	240	1476	special	224	24.3	9 990
New	GTX 295	2	240	1242	special	224	10.6	79 198
	GTX 465	1	352	1215	special	224	2.6	152 023
	GTX 480	1	480	1401	special	224	2.3	237 415
	GTX 580	1	512	1544	special	224	1.9	290 535
OpenSSL [44]	Intel Core i7-2600K			3400	special	224	0.27	14 740
OpenSSL using [29]	running on 4 cores				special	224	0.09	46 176

duplication are processed simultaneously, significantly reducing the latency at the expense of potentially lowering the throughput.

Another desirable property of an implementation of a parallel algorithm is that all threads follow the exact same procedure since this reduces the amount of divergent code. An active research area where such algorithms have been studied is in the context of cryptographic side-channel attacks. Side channel attacks [31] are attacks which use information gained from the physical implementation of a certain scheme to break its security; e.g. the elapsed time or power consumption. In order to avoid these types of attacks the algorithms must perform the same actions independent of the input to avoid leaking information. The approach we use is based on the Montgomery ladder applied to projective Weierstrass coordinates [17,26,13] instead of Montgomery coordinates. Even though the y-coordinate can be recovered, this is not necessary in most of the elliptic curve based cryptographic schemes because they only use the x-coordinate to compute the final result.

In particular, we adopt the formulas from [17]. Recall from Algorithm 3 that every iteration processes a single bit of the scalar at the cost of computing an elliptic curve addition and doubling. Computation on the Y-coordinate is omitted as follows [17]

$$
\begin{aligned}
(P+Q, 2Q) = (\tilde{P}, \tilde{Q}) = ((\tilde{P}_x, \tilde{P}_z), (\tilde{Q}_x, \tilde{Q}_z)) = \\
\begin{cases}
\tilde{P}_x = 2(P_xQ_z + Q_xP_z)(P_xQ_x + aP_zQ_z) \\
\qquad +4bP_z^2Q_z^2 - G_x(P_xQ_z - Q_xP_z)^2 \\
\tilde{P}_z = (P_xQ_z - Q_xP_z)^2 \\
\tilde{Q}_x = (Q_x^2 - aQ_z^2)^2 - 8bQ_xQ_z^3 \\
\tilde{Q}_z = 4(Q_xQ_z(Q_x^2 + aQ_z^2) + bQ_z^4).
\end{cases}
\end{aligned}
\tag{5}
$$

Note that G_x is the x-coordinate of the input point to the elliptic curve scalar multiplication algorithm. Using that the NIST standard defines $a = -3$, and slightly rewriting equation (5), results in the set of instructions presented in Table 2. Every row of the table is executed concurrently by the different threads, an empty slot means that the thread either remains idle or works on fake data. The bold entries in Table 2 are pre-computed at the initialization phase of the algorithm. The b value is one of the parameters which define the elliptic curve (together with a and p_{224}) and is provided in the standard. The b is invariant for the different concurrent elliptic curve scalar multiplications. Depending on the thread identifier, the pre-computed value is copied to the correct shared memory position which is used in operation number 6.

Using the instruction flow from Table 2 seven threads can compute a single elliptic curve scalar multiplication (ECSM) using the Montgomery ladder algorithm. The time T_{max} is three multiplications, two additions, two subtractions and a single triple operation in $\mathbf{F}_{p_{224}}$ to compute a single elliptic curve addition *and* duplication. This is in contrast with T_{sum} which consist of 18 multiplications, two triplings, four additions, five subtractions and two multiplications by a power of two in $\mathbf{F}_{p_{224}}$. Although T_{sum} is significantly higher, compared to the cost to process one bit of the scalar using different coordinate

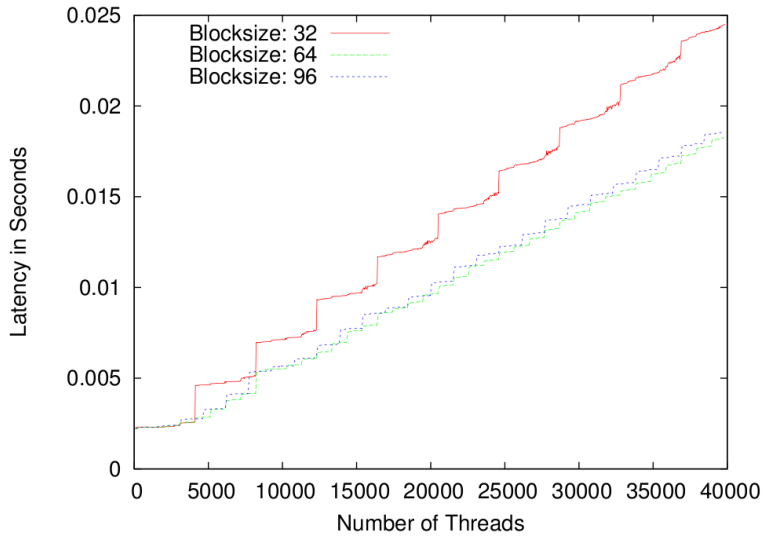

Fig. 1 Latency results when varying the amount of dispatched threads for blocksize equal to 32 (red, top line), 64 (green, bottom line) and 96 (blue, middle line) on the GTX 580 GPU.

representations and different ECSM algorithms, the latency T_{max} is roughly three multiplications to compute both an elliptic curve addition and doubling using seven threads. Running seven threads in parallel is the best, e.g. the highest number of concurrent running threads, we could achieve and it should be noted that this is suboptimal from different perspectives. First of all, a GPU platform using the CUDA paradigm typically dispatches threads in block sizes which are a small multiple of 32. Hence, in each subgroup of eight threads one thread remains inactive: decreasing the overall throughput. Secondly, 20 multiplications are used in Table 2 which results in one idle thread in the third multiplication (thread 7 in operation 6). On different parallel platforms, where i threads are processed concurrently, with $2 \leq i \leq 7$, the approach outlined in Table 2 can be computed using $\lceil \frac{20}{i} \rceil$ multiplications.

This approach is not limited to arithmetic modulo p_{224} but applies to any modulus. Given the (estimated) performance of a single modular multiplication, either using a special or a generic modulus, the approach from Table 2 can be applied such that the overall latency to multiply an elliptic curve point with a k-bit scalar is approximately the time to compute $k \cdot \lceil \frac{20}{i} \rceil$ single thread multiplications.

5 Results

Table 3 states the performance results, using the approach from Table 2, when using our GPU implementation running on a variety of GPUs from both the older and newer generations of CUDA architectures. Our benchmark results include transferring the in- and output and allows to use different multipliers and elliptic curve points in the same batch. To be compatible with as many settings used in practice as possible, it is not assumed that the initial elliptic curve point is given in affine coordinates but instead in projective coordinates. This has a performance disadvantage: the amount of data which needs to be transferred from the host to the GPU is increased. While primarily designed for the GTX 400 (and newer) family, the bold entries GTX 465, 480 and 580 in Table 3, the performance on the older GTX 200 series in terms of latency and throughput are remarkably good.

Our fastest result is obtained on the GTX 580 GPU when computing a single 224-bit elliptic curve scalar multiplication and requires 1.94 milliseconds when dispatching eight threads. This is an order of magnitude faster, in term of response time, compared to the previous fastest low-latency implementation [2]. Figure 1 shows the latencies when varying the amount of threads, eight threads are scheduled to work on a single ECSM, for different block-sizes on the GTX 580. There is a clear trade-off: increasing the block-size allows to hide the various latencies by performing a context switch and calculating on a different group of threads within the same block. On the other hand, every eight threads require their own memory and registers for the intermediate values as outlined in Table 2. Increasing the block size too much results in a performance degradation because the required memory for the entire block does not fit in the shared memory any more. As can be observed from Figure 1 a block-size of 64 results in the optimal practical performance when processing larger batches of ECSM computations.

To illustrate the computational power of the GPU even further let us consider the throughput when fixing the latency to 5 milliseconds. As can be seen from Figure 1, the GTX 580 can compute 916, 1024, or 960 224-bit elliptic curve scalar multiplications within this time limit when using a block-size of 32, 64, or 96 threads respectively. The best of these short runs already achieves a throughput of over 246 000 scalar multiplications per second, when using a blocksize of 64, which is already 0.85 of the maximum observed throughput obtained when processing much larger batches.

5.0.1 Performance Comparison

In [5] and the follow-up work [3] fast elliptic curve scalar multiplication is implemented using Edwards curves in a cryptanalytic setting. The GPU implementations optimize for high-throughput and implement generic modular arithmetic. The setting considered in [5,3] requires to perform an ECSM with a 11 797-bit scalar; in order to compare results we scale their figures by a factor $\frac{11\,797}{224}$. Comparing to these implementations is difficult because both the

finite field and elliptic curve arithmetic differ from the approaches considered in this paper where the faster arithmetic on Edwards curves cannot be used. On the GTX 295 architecture, for which our algorithms are not designed, the throughput reported in [3] is 3.3 times higher. The associated latency times are not reported. Since the approach from [3] is designed for high-throughput, e.g. one thread per ECSM resulting in no communication overhead, we expect that this approach has a higher throughput (and higher latency) on the Fermi cards compared to the new results from this work.

The GPU implementations discussed in [53, 2] target the same special modulus as discussed in this paper. In [53] one thread per multiplication is used (optimizing for high-throughput) and multiplies the same elliptic curve point in all threads. This reduces the amount of data which needs to be transferred from the host machine to the GPU. The authors of [2] implement a low-latency algorithm by parallelizing the finite field arithmetic using RNS (see Section 4.1). Their performance data do not include the transfer time of the input (output) from the host (GPU) to the GPU (host). Both the GTX 285 and 295 belong to the same GPU family, the former is clocked 1.2 faster than the latter while the GTX 295 consists of two of these slower GPUs. Compared to [2] our minimum latency is more than twice lower while the maximum throughput on a *single* slower GPU of the GTX 295 is almost quadrupled.

To put these results in perspective we have ran the latest benchmark suite in OpenSSL [44] of the Elliptic curve Diffie-Hellman protocol [54] which incorporates the optimizations reported in [29]. The target machine has an Intel Core i7-2600K CPU (clocked at 3.40GHz). The OpenSSL benchmark tool measures a throughput of 11 544 Diffie-Hellman operations per second per core (using NIST's p_{224} prime)[1]. Even though a single Diffie-Hellman operation is essentially an elliptic curve scalar multiplication there is some additional overhead in OpenSSL's benchmark suite so these results are only a lowerbound on the performance. Taken into account that this machine has four CPUs the throughput of our GPU implementation is more than six times higher (on the GTX 580). The latency on the CPU is lower: around 0.09 ms compared to 1.9 ms on the CPU and GPU respectively. This is no surprise as it was the main motivation to investigate low-latency implementations in this paper. These results show that the current parallel implementations are coming close to deliver low-latency while still significantly outperforming the CPU in terms of throughput.

6 Conclusion

We presented an algorithm which is particularly well-suited for parallel computer architectures to compute the scalar multiplication of an elliptic curve

[1] To enable the faster implementation using the techniques described in [29] OpenSSL 1.0.1 needs to be configured using "*./Configure enable-ec_nistp_64_gcc_128*", otherwise the throughput is 3 685 Diffie-Hellman operations per second.

point with as main goal to reduce latency. When applied to a 224-bit standardized elliptic curve used in cryptography and running on a GTX 580 graphics processing unit the minimum time required is 1.9 milliseconds; improving on previous low-latency results by an order of magnitude.

Acknowledgements

This work was supported by the Swiss National Science Foundation under grant numbers 200020-132160 and 200021-119776. We gratefully acknowledge Deian Stefan, for granting us access to the NVIDIA GTX 480 to benchmark our programs and the insightful comments by the International Journal of Parallel Programming reviewers.

References

1. AMD: ATI CTM Reference Guide. Technical Reference Manual (2006)
2. Antao, S., Bajard, J.C., Sousa, L.: Elliptic curve point multiplication on GPUs. In: Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on, pp. 192–199 (2010)
3. Bernstein, D.J., Chen, H.C., Chen, M.S., Cheng, C.M., Hsiao, C.H., Lange, T., Lin, Z.C., Yang, B.Y.: The billion-mulmod-per-second PC. In: Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009, pp. 131–144 (2009)
4. Bernstein, D.J., Chen, H.C., Cheng, C.M., Lange, T., Niederhagen, R., Schwabe, P., Yang, B.Y.: ECC2K-130 on NVIDIA GPUs. In: G. Gong, K.C. Gupta (eds.) Indocrypt 2010, *Lecture Notes in Computer Science*, vol. 6498, pp. 328–346. Springer-Verlag Berlin Heidelberg (2010)
5. Bernstein, D.J., Chen, T.R., Cheng, C.M., Lange, T., Yang, B.Y.: ECM on graphics cards. In: A. Joux (ed.) Eurocrypt 2009, *Lecture Notes in Computer Science*, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)
6. Bernstein, D.J., Lange, T.: Faster addition and doubling on elliptic curves. In: K. Kurosawa (ed.) Asiacrypt, *Lecture Notes in Computer Science*, vol. 4833, pp. 29–50. Springer, Heidelberg (2007)
7. Bernstein, D.J., Lange, T.: Analysis and optimization of elliptic-curve single-scalar multiplication. In: G.L. Mullen, D. Panario, I.E. Shparlinski (eds.) Finite Fields and Applications, *Contemporary Mathematics Series*, vol. 461, pp. 1–19. American Mathematical Society (2008)
8. Bevand, M.: MD5 Chosen-Prefix Collisions on GPUs. Black Hat (2009). Whitepaper
9. Blythe, D.: The Direct3D 10 system. ACM Transactions on Graphics **25**(3), 724–734 (2006)
10. Bos, J.W.: High-performance modular multiplication on the Cell processor. In: M.A. Hasan, T. Helleseth (eds.) Arithmetic of Finite Fields – WAIFI 2010, *Lecture Notes in Computer Science*, vol. 6087, pp. 7–24. Springer, Heidelberg (2010)
11. Bos, J.W., Kaihara, M.E., Kleinjung, T., Lenstra, A.K., Montgomery, P.L.: On the security of 1024-bit RSA and 160-bit elliptic curve cryptography. Cryptology ePrint Archive, Report 2009/389 (2009). `http://eprint.iacr.org/`
12. Bos, J.W., Stefan, D.: Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In: S. Mangard, F.X. Standaert (eds.) Cryptographic Hardware and Embedded Systems – CHES 2010, *Lecture Notes in Computer Science*, vol. 6225, pp. 279–293. Springer, Heidelberg (2010)
13. Brier, E., Joye, M.: Weierstraß elliptic curves and side-channel attacks. In: D. Naccache, P. Paillier (eds.) Public Key Cryptography – PKC 2002, *Lecture Notes in Computer Science*, vol. 2274, pp. 335–345. Springer, Heidelberg (2002)

14. Brown, M., Hankerson, D., López, J., Menezes, A.: Software implementation of the NIST elliptic curves over prime fields. In: D. Naccache (ed.) CT-RSA, *Lecture Notes in Computer Science*, vol. 2020, pp. 250–265. Springer, Heidelberg (2001)
15. Cohen, H., Miyaji, A., Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates. In: K. Ohta, D. Pei (eds.) Asiacrypt 1998, *Lecture Notes in Computer Science*, vol. 1514, pp. 51–65. Springer, Heidelberg (1998)
16. Edwards, H.M.: A normal form for elliptic curves. Bulletin of the American Mathematical Society **44**, 393–422 (2007)
17. Fischer, W., Giraud, C., Knudsen, E.W., Seifert, J.P.: Parallel scalar multiplication on general elliptic curves over $\mathbb{F}_p$ hedged against non-differential side-channel attacks. Cryptology ePrint Archive, Report 2002/007 (2002). `http://eprint.iacr.org/`
18. Garland, M., Grand, S.L., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with CUDA. IEEE Micro **28**(4), 13–27 (2008)
19. Garner, H.L.: The residue number system. IRE Transactions on Electronic Computers **8**, 140–147 (1959)
20. Granlund, T.: GMP small operands optimization. In: Software Performance Enhancement for Encryption and Decryption – SPEED 2007 (2007)
21. Group, K.: OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`
22. Harrison, O., Waldron, J.: AES encryption implementation and analysis on commodity graphics processing units. In: P. Paillier, I. Verbauwhede (eds.) Cryptographic Hardware and Embedded Systems – CHES 2007, *Lecture Notes in Computer Science*, vol. 4727, pp. 209–226. Springer, Heidelberg (2007)
23. Harrison, O., Waldron, J.: Practical symmetric key cryptography on modern graphics hardware. In: Proceedings of the 17th conference on Security symposium, pp. 195–209. USENIX Association (2008)
24. Harrison, O., Waldron, J.: Efficient acceleration of asymmetric cryptography on graphics hardware. In: B. Preneel (ed.) Africacrypt 2009, *Lecture Notes in Computer Science*, vol. 5580, pp. 350–367. Springer, Heidelberg (2009)
25. Hisil, H., Wong, K.K.H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: J. Pieprzyk (ed.) Asiacrypt 2008, *Lecture Notes in Computer Science*, vol. 5350, pp. 326–343. Springer, Heidelberg (2008)
26. Izu, T., Takagi, T.: A fast parallel elliptic curve multiplication resistant against side channel attacks. In: D. Naccache, P. Paillier (eds.) Public Key Cryptography – PKC 2002, *Lecture Notes in Computer Science*, vol. 2274, pp. 371–374. Springer, Heidelberg (2002)
27. Joye, M., Yen, S.M.: The Montgomery powering ladder. In: B.S. Kaliski Jr., Ç. K. Koç, C. Paar (eds.) Cryptographic Hardware and Embedded Systems – CHES 2002, *Lecture Notes in Computer Science*, vol. 2523, pp. 1–11. Springer, Heidelberg (2003)
28. Karatsuba, A.A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. No. 145 in Proceedings of the USSR Academy of Science, pp. 293–294 (1962)
29. Käsper, E.: Fast elliptic curve cryptography in OpenSSL (to appear). In: G. Danezis, S. Dietrich, K. Sako (eds.) The 2nd Workshop on Real-Life Cryptographic Protocols and Standardization, *Lecture Notes in Computer Science*, vol. 7126. Springer (2012). `http://research.google.com/pubs/archive/37376.pdf`
30. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**(177), 203–209 (1987)
31. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: N. Koblitz (ed.) Crypto 1996, *Lecture Notes in Computer Science*, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
32. Lenstra, A.K., Verheul, E.R.: Selecting cryptographic key sizes. Journal of Cryptology **14**(4), 255–293 (2001)
33. Lenstra Jr., H.W.: Factoring integers with elliptic curves. Annals of Mathematics **126**(3), 649–673 (1987)
34. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA tesla: A unified graphics and computing architecture. Micro, IEEE **28**(2), 39–55 (2008)

35. Manavski, S.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on, pp. 65–68 (2007)
36. Merrill, R.D.: Improving digital computer performance using residue number theory. Electronic Computers, IEEE Transactions on **EC-13**(2), 93–101 (1964)
37. Miller, V.S.: Use of elliptic curves in cryptography. In: H.C. Williams (ed.) Crypto 1985, *Lecture Notes in Computer Science*, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
38. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177), 243–264 (1987)
39. Moss, A., Page, D., Smart, N.P.: Toward acceleration of RSA using 3D graphics hardware. In: S.D. Galbraith (ed.) Proceedings of the 11th IMA international conference on Cryptography and coding, Cryptography and Coding 2007, pp. 364–383. Springer-Verlag (2007)
40. National Security Agency: Fact sheet NSA Suite B Cryptography. `http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml` (2009)
41. Nickolls, J., Dally, W.J.: The GPU computing era. IEEE Micro **30**(2), 56–69 (2010)
42. NVIDIA: NVIDIA's next generation CUDA compute architecture: Fermi (2009)
43. NVIDIA: NVIDIA CUDA Programming Guide 3.2 (2010)
44. OpenSSL: The Open Source toolkit for SSL/TLS. `http://www.openssl.org/` (2012)
45. Osvik, D.A., Bos, J.W., Stefan, D., Canright, D.: Fast software AES encryption. In: S. Hong, T. Iwata (eds.) Fast Software Encryption – FSE 2010, *Lecture Notes in Computer Science*, vol. 6147, pp. 75–93. Springer, Heidelberg (2010)
46. Owens, J.: GPU architecture overview. In: Special Interest Group on Computer Graphics and Interactive Techniques – SIGGRAPH 2007, p. 2. ACM (2007)
47. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface, fourth edn. Morgan Kaufmann, San Francisco, California (2009)
48. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**, 120–126 (1978)
49. Segal, M., Akeley, K.: The OpenGL graphics system: A specification (version 2.0). Silicon Graphics, Mountain View, CA (2004)
50. Silverman, J.H.: The Arithmetic of Elliptic Curves, *Gradute Texts in Mathematics*, vol. 106. Springer-Verlag (1986)
51. Solinas, J.A.: Generalized Mersenne numbers. Technical Report CORR 99–39, Centre for Applied Cryptographic Research, University of Waterloo (1999)
52. of Standards, N.I., Technology: Special publication 800-57: Recommendation for key management part 1: General (revised). `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf`
53. Szerwinski, R., Güneysu, T.: Exploiting the power of GPUs for asymmetric cryptography. In: E. Oswald, P. Rohatgi (eds.) Cryptographic Hardware and Embedded Systems – CHES 2008, *Lecture Notes in Computer Science*, vol. 5154, pp. 79–99. Springer, Heidelberg (2008)
54. U.S. Department of Commerce and National Institute of Standards and Technology: Recommendation for pair-wise key establishment schemes using discrete logarithm cryptography. See `http://csrc.nist.gov/publications/nistpubs/800-56A/SP800-56A_Revision_Mar08-2007.pdf` (2007)
55. U.S. Department of Commerce/National Institute of Standards and Technology: Digital Signature Standard (DSS). FIPS-186-3 (2009). `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`
56. Yang, J., Goodman, J.: Symmetric key cryptography on modern graphics hardware. In: K. Kurosawa (ed.) Asiacrypt, *Lecture Notes in Computer Science*, vol. 4833, pp. 249–264. Springer, Heidelberg (2007)