ECM on Graphics Cards

Daniel J. Bernstein[1], Tien-Ren Chen[2], Chen-Mou Cheng[3], Tanja Lange[4], and Bo-Yin Yang[2]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Institute of Information Science, Academia Sinica,
128 Section 2 Academia Road, Taipei 115-29, Taiwan.
{by,trchen1033}@crypto.tw
[3] Department of Electrical Engineering,
National Taiwan University, Taipei 106-70, Taiwan
doug@crypto.tw
[4] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
tanja@hyperelliptic.org

Abstract. This paper reports record-setting performance for the elliptic-curve method of integer factorization: for example, 604.99 curves/second for ECM stage 1 with $B_1 = 8192$ for 280-bit integers on a single PC. The state-of-the-art GMP-ECM software handles 171.42 curves/second for ECM stage 1 with $B_1 = 8192$ for 280-bit integers using all four cores of a 2.4GHz Core 2 Quad Q6600.

The extra speed takes advantage of extra hardware, specifically two NVIDIA GTX 280 graphics cards, using a new ECM implementation introduced in this paper. Our implementation uses Edwards curves, relies on new parallel addition formulas, and is carefully tuned for the highly parallel GPU architecture. On a single GTX 280 the implementation performs 22.66 million modular multiplications per second for a general 280-bit modulus. GMP-ECM, using all four cores of a Q6600, performs 17.91 million multiplications per second.

This paper also reports speeds on other graphics processors: for example, 2414 280-bit elliptic-curve scalar multiplications per second on an older NVIDIA 8800 GTS (G80), again for a general 280-bit modulus. For comparison, the CHES 2008 paper "Exploiting the Power of GPUs for Asymmetric Cryptography" reported 1412 elliptic-curve scalar multiplications per second on the same graphics processor despite having fewer bits in the scalar (224 instead of 280), fewer bits in the modulus (224 instead of 280), and a *special* modulus $(2^{224} - 2^{96} + 1)$.

Keywords: Factorization, graphics processing unit, modular arithmetic, elliptic curves, elliptic-curve method of factorization, Edwards curves.

[*] Permanent ID of this document: 6904068c52463d70486c9c68ba045839. Date of this document: 2008.11.11. This work has been supported in part by the National Science Foundation under grant ITR–0716498, in part by Taiwan's National Science Council NSC-96-2218-E-001-001, and in part by the European Commission through the IST Programme under Contract ICT–2007–216676 ECRYPT. Part of this work was carried out while Bernstein and Lange visited NTU.

1 Introduction

The elliptic-curve method of factorization was introduced by Lenstra in [34] as a generalization of Pollard's $p-1$ and Williams' $p+1$ method. Many speedups and good choices of elliptic curves were suggested and ECM is now the method of choice to find factors in the range 10^{10} to 10^{60} of general numbers. The largest factor found by ECM was a 222-bit factor of the 1266-bit number $10^{381}+1$ found by Dodson (see [48]).

Cryptographic applications such as RSA use "hard" integers with much larger prime factors. The number-field sieve (NFS) is today's champion method of finding those prime factors. It was used, for example, in the following factorizations:

integer	bits	details reported
RSA–130	430	at ASIACRYPT 1996 by Cowie et al. [16]
RSA–140	463	at ASIACRYPT 1999 by Cavallar et al. [12]
RSA–155	512	at EUROCRYPT 2000 by Cavallar et al. [13]
RSA–200	663	in 2005 posting by Bahr et al. [4]
$2^{1039}-1$	1039 (special)	at ASIACRYPT 2007 by Aoki et al. [2]

A 1024-bit RSA factorization by NFS would be considerably more difficult than the factorization of the special integer $2^{1039}-1$ but has been estimated to be doable in a year of computation using standard PCs that cost roughly $1 billion or using ASICs that cost considerably less. See [42], [35], [19], [22], [43], and [29] for various estimates of the cost of NFS hardware. Current recommendations for RSA key sizes — 2048 bits or even larger — are based directly on extrapolations of the speed of NFS.

NFS is also today's champion index-calculus method of computing discrete logarithms in large prime fields, quadratic extensions of large prime fields, etc. See, e.g., [26], [27], and [5]. Attackers can break "pairing-friendly elliptic curves" if they can compute discrete logarithms in the corresponding "embedding fields"; current recommendations for "embedding degrees" in pairing-based cryptography are again based on extrapolations of the speed of NFS. See, e.g., [30].

NFS factors a "hard" integer n by combining factorizations of many smaller auxiliary "smooth" integers. For example, the factorization of RSA-155 $\approx 2^{512}$ generated a pool of $\approx 2^{50}$ auxiliary integers $< 2^{200}$, found $\approx 2^{27}$ "smooth" integers factoring into primes $< 2^{30}$, and combined those integers into a factorization of RSA-155. See [13] for many more details.

Textbook descriptions of NFS state that prime factors of the auxiliary integers are efficiently discovered by sieving. However, sieving requires increasingly intolerable amounts of memory as n grows. Cutting-edge NFS computations control their memory consumption by using other methods — primarily ECM — to discover large prime factors. Unlike sieving, ECM remains productive with limited amounts of memory.

Aoki et al. in [2] discovered small prime factors by sieving, discarded any unfactored parts above 2^{105}, and then used ECM to discover primes up to 2^{38}.

Kleinjung reported in [29, Section 5] on ECM "cofactorisation" for a 1024-bit n consuming, overall, a similar amount of time to sieving.

The size of the auxiliary numbers to be factored by ECM depends on the size of the number to be factored with the NFS and on the relative speed of the ECM implementation. The SHARK design [19] for factoring 1024-bit RSA makes two suggestions for parameters of ECM — one uses it for 125-bit numbers, the other for 163-bit numbers. The SHARK designers remark that ECM could be used more intensively. In their design, ECM can be handled by conventional PCs or hardware. They write "Special hardware for ECM ... can save up to 50% of the costs for SHARK" and "The importance of using special hardware for factoring the potential sieving reports grows with the bit length of the number to be factored." As a proof of concept Pelzl et al. present in [39] an FPGA-based implementation of ECM for numbers up to 200 bits and state "We show that massive parallel and cost-efficient ECM hardware engines can improve the area-time product of the RSA moduli factorization via the GNFS considerably." Gaj et al. [20] consider the same task and improve upon their results.

Evidently ECM is becoming one of the most important steps in the entire NFS computation. Speedups in ECM are becoming increasingly valuable as tools to speed up NFS.

This paper suggests graphics processing units (GPUs) as computation platforms for ECM, presents algorithmic improvements that are particularly helpful in the GPU context, and reports new ECM implementations for several NVIDIA GPUs. GPUs achieve high throughput through massive parallelism — usually more than 100 processes are run at clock frequencies which are not much lower than that of state-of-the-art CPUs; e.g. the GeForce 8800GTS runs 128 processes at 1.625 GHz. This parallelism is well suited for ECM factorizations inside the NFS, although it also creates new resource-allocation challenges, as discussed later in this paper. We focus on moduli of 200–300 bits since we (correctly) predicted that our ECM implementation would be faster than previous ones and since we are looking ahead to larger NFS factorizations than 1024 bits.

Measurements show that a computer running this paper's new ECM implementation on a GPU performs 22.7 million 280-bit modular multiplications per second and has a better price-performance ratio than a computer running the state-of-the-art GMP-ECM software on all four cores of a Core 2 Quad CPU. The best price-performance ratio is obtained by a computer that has a CPU and two GPUs contributing to the ECM computation.

2 Background on ECM

A thorough presentation of the Elliptic-Curve Method (ECM) of factorization is given by Zimmermann and Dodson in [47]. Their paper also describes extensive details of the GMP-ECM software, essentially the fastest known ECM implementation to date. For more recent improvements of bringing together ECM with the algorithmic advantages of Edwards curves and improved curve choices we refer to [8] by Bernstein et al.

2.1 Overview of ECM

ECM tries to factor an integer m as follows.

Let E be an elliptic curve over $\mathbf{Q}$ with neutral element O. Let P be a non-torsion point on E. If the discriminant of the curve or any of the denominators in the coefficients of E or P happens not to be coprime with m without being divisible by it we have found a factor and thus completed the task of finding a nontrivial factor of m; if one of them is divisible by m we choose a different pair (E, P). We may therefore assume that E has good reduction modulo m. In particular we can use the addition law on E to define an addition law on $\tilde{E}$, the reduction of E modulo m; let $\tilde{P} \in \tilde{E}$ be the reduction of P modulo m.

Let ϕ be a rational function on E which has a zero at O and has non-zero reduction of $\phi(P)$ modulo m. In the familiar case of Weierstrass curves this function can simply be Z/Y. For elliptic curves in Edwards form a similarly simple function exists; see below.

Let s be an integer that has many small factors. A standard choice is $s = \operatorname{lcm}(1, 2, 3, \ldots, B_1)$. Here B_1 is a bound controlling the amount of time spent on ECM. The main step in ECM is to compute $R = [s]\tilde{P}$. The computation of the scalar multiple $[s]\tilde{P}$ on $\tilde{E}$ is done using the addition law on E and reducing intermediate results modulo m.

One then checks $\gcd(\phi(R), m)$; ECM succeeds if the gcd is nontrivial. If this first step — called stage 1 — was not successful then one enters stage 2, a postprocessing step that significantly increases the chance of factoring m. In a simple form of stage 2 one computes $R_1 = [p_{k+1}]R, R_2 = [p_{k+2}]R, \ldots, R_\ell = [p_{k+\ell}]R$ where $p_{k+1}, p_{k+2}, \ldots, p_{k+\ell}$ are the primes between B_1 and B_2, and then does another gcd computation $\gcd(\phi(R_1)\phi(R_2)\cdots\phi(R_\ell), m)$. There are more effective versions of stage 2. Stage 2 takes significantly less time than stage 1 when ECM as a whole is optimized.

If q is a prime divisor of m, and the order of P modulo q divides s, then $\phi([s]\tilde{P}) \equiv 0 \pmod{q}$. If $\phi([s]\tilde{P}) \not\equiv 0 \bmod m$ we obtain a nontrivial factor of m in stage 1 of ECM as $\gcd(m, \phi([s]\tilde{P}))$. This happens exactly if there are two prime divisors of m such that s is divisible by the order of P modulo one of them but not modulo the other. Choosing s to have many small factors increases the chance of m having at least one prime divisor q such that the order of P modulo q divides s. Note that it is rare that this happens for all factors of m simultaneously unless s is huge.

Similar comments apply to stage 2, with s replaced by sp_{k+1}, sp_{k+2}, etc.

Trying a single curve with a large B_1 is usually less effective than spending the same amount of time trying many curves, each with a smaller B_1. For each curve one performs stage 1 and then stage 2.

2.2 Edwards curves

Edwards curves were introduced by Edwards in [18] and studied for cryptography by Bernstein and Lange in [10]. An Edwards curve is given by an equation of the form $x^2+y^2 = 1+dx^2y^2$, for some $d \notin \{0, 1\}$. Bernstein and Lange show that each

elliptic curve with a point of order 4 is birationally equivalent to an Edwards curve over the same field. For ECM we are interested in curves with smooth order modulo factors of m, so in particular the condition of having a point of order 4 is not a problem. On the contrary, curves with large $\mathbf{Q}$-rational torsion subgroup are more likely to lead to factorizations since the torsion subgroup is mapped injectively under reduction. For our implementation we used Edwards curves with $\mathbf{Q}$-torsion group isomorphic to $\mathbf{Z}/2 \times \mathbf{Z}/8$ which were generated with the Edwards analogue of the Atkin-Morain construction [3] as described in [8].

The addition law on Edwards curves is given by

$$(x_1, y_1) \oplus (x_2, y_2) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right).$$

The neutral element is $(0, 1)$, so ϕ in the previous subsection can simply be the x-coordinate.

For an overview of explicit formulas for arithmetic on elliptic curves we refer to the Explicit-Formulas Database (EFD) [9]. For doublings on Edwards curves we use the formulas by Bernstein and Lange [10]. For additions we use the mixed-addition formulas by Hisil et al. [25, page 8] to minimize the number of multiplications. Earlier formulas by Bernstein and Lange are complete, and can take advantage of fast multiplications by small parameters d, but completeness is not helpful for ECM, and curves constructed by the Atkin-Morain method have large d.

Note that the paper [8] also contains improvements of ECM using curves with small coefficients and base point, in which case using inverted twisted Edwards coordinates (see [7]) becomes advantageous. In Section 4 we describe how we implemented modular arithmetic on the GPU. The chosen representation does not give any speed-up for multiplication by small parameters. This means that we do not get the full benefit of [8] — but we still benefit from the faster elliptic-curve arithmetic and the more successful curve choices on top of the fast and highly parallel computation platform. In Section 5 we present new parallelized formulas for Edwards-curve arithmetic.

3 The Working Platforms from NVIDIA

Driven by applications in video games, graphics-processing units (GPUs) have developed into very powerful and highly parallel processing units that find more and more interest outside graphics-processing applications. In cryptography so far mostly secret-key applications were implemented (see e.g. [14] and the book [15]) while taking full advantage of GPUs for public-key cryptography remained a challenge [37]. This may be attributable to the fact that programming graphics cards used to involve arcane programming with OpenGL.

With the G80 series of GPUs, NVIDIA not only significantly boosted the hardware capabilities of its cards, but also presented programmers with a significant advance in the programming model by introducing CUDA, a C-like programming language which can be compiled to run on the GPU. In this section

we review details of the current generation of NVIDIA graphics cards which we used for our implementation and also give some background on CUDA.

3.1 NVIDIA GeForce Video Cards in the G8x/G9x/G2xx Series

G8x Series A typical NVIDIA G80 GPU can be found in the GeForce 8800 GTX. It contains 128 streaming processors (SPs), grouped into 16 multiprocessors (MPs). Each MP thus has 8 SPs. In every cycle, each SP is capable of delivering three single-precision floating-point operations: one multiply-and-add, plus a multiplication from the texture processing unit. Or, it can deliver one 32-bit integer addition/subtraction or simple logical operation such as bitwise AND/OR/XOR per cycle. More complicated arithmetic operations, such as floating-point division and 32-bit integer multiplication, are performed by two "super" function units on each MP. Thus, the effective throughput of these more complicated operations is 25% of that of the simple arithmetic operations performed by SPs.

There are several different types of memory available on NVIDIA graphics cards. There are global and thread-local memories, both uncached. On a GeForce 8800 GTX, the SPs run at 1.35 GHz, while the uncached memory provides a throughput of 86.4 GB/s. This translates to 4 bytes per cycle per MP, not to mention the associated high latency (typically 400 to 600 cycles). There are also cached, read-only constant and texture memories, whose latency is low on cache hits. The GPU is also able to broadcast data to multiple threads in a single cycle if the data fetched is needed by more than one thread. Finally, there is a 16-bank, 16 KB shared memory per MP, which can deliver 64 bytes every two cycles, or 4 bytes per cycle per SP if there is no bank conflict. All caches and shared memories have latency close to that of registers and certainly much lower than device memories.

Graphic chips codenamed G84/G85/G86 are low-end parts of the G80 series marketed by NVIDIA with the same architecture but fewer (maximum of 4, sometimes as few as 1) MPs and much lower memory throughput. These are not cost-effective for our purposes (except for occasionally testing code on the road).

The G84/G85/G86 parts are not to be confused with lower-priced G80 parts (e.g., the "old" GeForce 8800GTS in [45], which has 96 SPs = 12 MPs, and the old 8800GT with 14 MPs) with slightly lower clock rates, fewer functional units, and lower memory throughput. Due to marketing reasons, chip companies often sell a top-end, envelope-pushing part at a huge markup, so it is often advantageous to buy "just below the top" (we see this also with the Core 2 Quad Q6600).

G9x and GT2xx Series NVIDIA's G92 GPUs is more or less a straightforward die shrink of the G80 from 90nm to 65nm processes. Versions of the G92 were used in the "new" GeForce 8800GTS 512MB (16 MPs, not to be confused with the older part) as well as the 9800 series cards such as the 9800GTX (16 MPs) and the 9800GX2 (in almost every aspect, two 9800GTX's bolted back-to-back

and sold together). Just as in the G8x series, there are low-end G94/G96 series GPUs sold, which are again not at all cost-effective for us.

In contrast, the newest GT2xx (formerly codenamed the G10x) series does offer an improved design. Now each GPU has 24 (in the GTX 260) to 30 (in the 280) MPs, nearly double the number at a slightly lower clock rate. These GPUs also have various features which should yield significant benefits if properly taken advantage of. In particular we use that there are twice as many registers.

3.2 The CUDA Programming Paradigm

CUDA provides an environment in which software programmers can program GPUs using a high-level, C-like language. A CUDA program (called a "kernel") starts with a source file `foo.cu`, which is first compiled by `nvcc`, the CUDA compiler, into code for a virtual machine (`foo.ptx`), then converted into actual machine code by the NVIDIA driver, and finally loaded and run.

CUDA adopts a super-threaded, massively parallel computational model, in which computation is broken down and executed by a large number (typically thousands) of threads. The seemingly concurrent execution of these threads is mapped onto to a pool of physical processors (e.g., the 128 SPs in G80). Those threads mapped onto the same physical processor time-share the underlying processor. Such time-sharing is necessary for effectively hiding instruction latency, similar to the reason why modern superscalar CPUs search for independent instructions to issue in a program stream. As a result, the parallelism utilized by CUDA is termed *thread-level parallelism*, in contrast with instruction-level parallelism achieved by superscalar CPUs.

At the programming level, the minimal scheduling entity is a *warp* of threads, which consists of 32 threads in the current version of CUDA. A warp must be executed by a single MP. It takes four cycles for a multiprocessor to issue an instruction for a warp of threads (or 16 cycles if the instruction is to be executed by the super function units). To achieve optimal instruction throughput, the threads belonging to the same warp must execute the same instruction, for there is only one instruction-decoding unit on each MP. We may hence regard an MP as a 32-way SIMD vector processor. Furthermore, the typical instruction has a latency around 20 to 24 cycles even when it involves only on-die memories (such as registers and shared memory). Thus, it is recommended to have at least 6 warps of threads in order to completely hide this latency.

We note that the GPU threads are lightweight hardware threads, which incur little overhead in context switch. In order to support fast context switch, the set of physical registers is divided among all active threads. This creates some pressure when programming GPUs. For example, on G80 and G92 there are only 8192 registers per MP. If we were to use 256 threads, then each thread would have a share of 32 registers, which is a tight budget for implementing complicated algorithms. The situation has improved in the latest GTX 260/280 family, which has twice as many registers, relieving the pressure of register scarcity and making programming much easier.

To summarize, the massive parallelism in the architectural design of NVIDIA GPUs makes their programming very different from a traditional CPU architecture with mostly sequential execution. Due to overhead, once a program loads (with its data), one wants to run it on the SPs for as long as possible. In general, GPUs are most suitable for executing the data-parallel part of an algorithm. Finally, to get the most out of the theoretical arithmetic throughput, one must manage memory use carefully to minimize memory access (except for on-die fast memory), as well as meticulously arrange the parallel execution of hardware threads to avoid bank conflicts in memory access.

4 High-throughput Modular Arithmetic on a GPU

In this section we give details and explain the design choices made in our implementation of the modular arithmetic on the GPU and show how parallelism is used on this level.

4.1 Limitation of CUDA

Pressure on Fast On-die Memories The major source of resource limitation when programming with CUDA is memory — in particular, fast memory — including per-thread registers and per-multiprocessor shared memory. Most significantly, on a G8x/G9x/G2xx GPU the per-SP working set of 2 KB is barely enough room to hold the base point and intermediate point for a scalar multiplication on an elliptic curve without any precomputation. To put it in perspective, all 240 SPs on a gigantic (1.4×10^9 gates) GTX 280 have between them 480 KB fast memory. That is less than the 512 KB of fast L2 cache in an aged Athlon64 (1.6×10^8 gates)! Unfortunately, CUDA requires many more threads, i.e., 24 times the number of streaming processors, to hide instruction latency effectively. Therefore, we will need collaboration and hence communication among groups of threads in order to achieve a high utilization of the instruction pipelines when implementing modular arithmetic operations on GPUs.

Race Conditions Another pitfall frequently encountered when programming with CUDA is race conditions. In CUDA, threads are organized into blocks with the constraint that threads belonging to the same block must execute on the same multiprocessor. Sometimes, the execution of a block of threads will need to be serialized when there is resource contention, e.g., when accessing device memory. Sometimes even accessing shared memory can result in serialization when there is bank conflict. Synchronization among a block of threads is achieved by calling the intrinsic `__syncthreads()` primitive, which blocks the execution until all threads in a block have reached the same point in the program. This is most useful when accessing slow device memory, whose latency is typically on the order of 400 to 600 cycles. There is no atomic transaction when accessing the fast shared memory for older GPUs such as G80 and G92. Therefore, when accessing the same memory position, threads from a same block may suffer from

race conditions and thus may require explicit synchronization (i.e., serialization) instructions, which also carry heavy performance penalties.

4.2 Design Choices of Modular Multiplication

As mentioned above, due to the special characteristics of the computing platform at hand, we need to focus on minimizing the storage space and communication overhead. For our target of 280-bit numbers, we opt for schoolbook multiplication because it involves the least amount of synchronization and intermediate storage. We have actually also implemented the standard Karatsuba technique which has a lower number of word multiplications in theory. However, on the GPU its performance is worse than the straightforward schoolbook method because the latter offers more opportunities of merging multiplications and additions into multiply-and-adds, resulting in a lower instruction count.

We represent an integer using L limbs in radix 2^r, with each limb between -2^{r-1} and 2^{r-1}. This allows us to easily represent any integer between $-R/2$ and $R/2$, where $R = 2^{Lr}$. To lower the costs of modular reduction we chose to use Montgomery representation of the integers modulo m, where m is the auxiliary number of be factored, and thus represent $x \bmod m$ as $x' = Rx \bmod m$. Note that our limbs can be negative and so we use a signed representative in $-m/2 \leq x' \bmod m < m/2$. In Montgomery representation, addition and subtraction are performed on the representatives as usual. Let m' be the unique positive integer between 0 and R such that $RR' - mm' = 1$. Given $x' = Rx \bmod m$ and $y' = Ry \bmod m$ the multiplication is computed on the representatives as $\alpha = (x'y' \bmod R)m' \bmod R$ followed by $\beta = (x'y' + \alpha m)/R$. Note that since R is a power of 2, modular reductions modulo R correspond to taking the lower bits while divisions by R correspond to taking the higher bits. One verifies that $-m < \beta < m$ and $\beta = R(xy) \bmod m$, so β equals xy in Montgomery representation.

The Chosen Parameters In the implementation described in this paper we take $L = 28$ and $r = 10$. Thus, we can handle integers up to around 280 bits. To fill up each multiprocessor with enough threads to effectively hide the instruction latency, we choose a block size of 256 threads, which is in charge of computing eight 280-bit arithmetic operations at a time. This means that we have an 8-way modular multiplier per multiprocessor. This also means that we can have up to 32 threads in the computation of one 280-bit arithmetic operation. We use up to 28 threads in the computation of the most intensive arithmetic operation, namely, the Montgomery multiplication. We need 3 integer multiplications here: one to obtain $x'y'$, one to obtain α, and the third to obtain β. The multiplications are implemented using 28 threads, each of which is responsible for cross-multiplying 7 limbs from x' with 4 from y'. The reason why we choose this topology is to avoid bank conflicts and race conditions in reading from and writing to shared memory. We note that a bank conflict can still occur when the memory addresses accessed by threads from a half warp do not fall evenly onto 16 banks, so we need to carefully arrange x' and y' from different curves in shared memory and pad when necessary.

4.3 Theoretical Performance Analysis

We give a back-of-envelope estimation of the performance of our design on a GTX 280. This GPU has 240 cores running at 1.296 GHz, each of which can deliver two floating-point operations per clock cycle via the multiply-and-add (MAD) instruction, which multiplies two floating-point numbers and then adds a third floating-point number.

As we have mentioned in Section 4.2, each thread is responsible for cross-multiplying the limbs in a 7-by-4 region. In the inner loop of a big-integer multiplication without carry, each thread needs to load these numbers into registers (11 loads from fast on-die memory), multiply and accumulate them into temporary storage (28 MAD instructions), and then appropriately accumulate the result back to a region shared by all 32 threads (10 load-and-adds, 10 `__syncthreads` to prevent compiler from reordering the instructions, plus 10 stores). So it would take about 75 instructions per thread, including overhead, to complete such a vanilla multiplication. A partial parallel carry takes about 7 instructions by properly manipulating floating-point arithmetic instructions, and we need two partial carries in order to bring the value in each limb to its normal range. Furthermore, in Montgomery reduction we need a full carry for an intermediate result of twice the length, so we essentially need 4 full carries in each modular multiplication, resulting in 56 extra instructions per thread. This gives an estimate of 281 instructions per modular multiplication.

5 Fast ECM on a GPU

We now describe our implementation of the ECM computation on GPUs using the high-throughput modular arithmetic described in the previous section. Recall that the speed bottleneck of ECM is scalar multiplication on an elliptic curve modulo m and that the factorization of m involves this computation on many curves.

Applications such as the number-field sieve add a further dimension in that factorizations of many auxiliary numbers are needed. We decided to use the parallelism of the GPU to handle many curves for a given auxiliary integer which thus can be stored in shared memory. All processors follow the same series of instructions which is a scalar multiplication on the respective curve modulo the same m and with the same scalar s. Different auxiliary factorizations can be handled by different GPUs in parallel since no communication is necessary in the NFS computation; for other parameter choices, e.g. smaller numbers m or GPUs with even more processors, one can also consider handling multiple integers on one GPU. For the rest of this section we consider one fixed m and s.

The CPU first prepares the curve parameters (including the coordinates of the starting point) in an appropriate format and passes them to the GPU for scalar multiplication, whose result will be returned by the GPU. The CPU then does the gcd computation to determine whether we have found any factors.

Our implementation of modular arithmetic in essence turns a multiprocessor on a GPU into an 8-way modular arithmetic unit (MAU) that is capable of

Fig. 1. Explicit formulas for DBL-DBL.

Step	MAU 1	MAU 2	
1	$A{=}X_1^2$	$B{=}Y_1^2$	S
2	$X_1{=}X_1 + Y_1$	$C{=}A + B$	a
3	$X_1{=}X_1^2$	$Z_1{=}Z_1^2$	S
4	$X_1{=}X_1 - C$	$Z_1{=}Z_1 + Z_1$	a
5	$B{=}B - A$	$Z_1{=}Z_1 - C$	a
6	$X_1{=}X_1 \times Z_1$	$Y_1{=}B \times C$	M
7	$A{=}X_1 \times X_1$	$Z_1{=}Z_1 \times C$	M
8	$Z_1{=}Z_1^2$	$B{=}Y_1^2$	S
9	$Z_1{=}Z_1 + Z_1$	$C{=}A + B$	a
10	$B{=}B - A$	$X_1{=}X_1 + Y_1$	a
11	$Y_1{=}B \times C$	$X_1{=}X_1 \times X_1$	M
12	$B{=}Z_1 - C$	$X_1{=}X_1 - C$	a
13	$Z_1{=}B \times C$	$X_1{=}X_1 \times B$	M
			4M+3S+6a

carrying out 8 modular arithmetic operations simultaneously. How to map our elliptic-curve computation onto this array of 8-way MAUs on a GPU is of crucial importance.

We have explored two different approaches to use the 8-way MAUs we have implemented. The first one is straightforward: we compute on 8 curves in parallel, each of which uses a dedicated MAU. This approach results in 2 KB of working memory per curve, barely enough to store the curve parameters (including the base point) and the coordinates of the intermediate point. Besides the base point, we cannot cache any other points, which implies that the scalar multiplication can use only a non-adjacent form (NAF) representation of s. So we need to compute $\log_2 s$ doublings and on average $(\log_2 s)/3$ additions to compute $[s]\tilde{P}$.

In the second approach, we combine 2 MAUs to compute the scalar multiplication on a single curve. As mentioned in Sections 2 and 4, our implementation uses Montgomery representation of integers, and thus it does not benefit from multiplications with small values. In particular, multiplications with the curve coefficient d take the same time as general multiplications. We provide the base point and all precomputed points (if any) in affine coordinates, so all curve additions are mixed additions. Inspecting the explicit formulas, one notices that both addition and doubling require an odd number of multiplications/squarings. In order to avoid idle multiplication cycles, we have developed new parallel formulas that pipeline two group operations. The scalar multiplication can be composed of the building blocks DBL-DBL (doubling followed by doubling), mADD-DBL (mixed addition followed by doubling) and DBL-mADD. Note that there are never two subsequent additions. At the very end of the scalar multiplication, one might encounter a single DBL or mADD, in that case 1 processor is idle in the final multiplication.

Fig. 2. Explicit formulas for mADD-DBL.

Step	MAU 1	MAU 2	
1	$B{=}x_2 \times Z_1$	$C{=}y_2 \times Z_1$	M
2	$A{=}X_1 \times Y_1$	$Z_1{=}B \times C$	M
3	$E{=}X_1 - B$	$F{=}Y_1 + C$	a
4	$X_1{=}X_1 + C$	$Y_1{=}Y_1 + B$	a
5	$E{=}E \times F$	$Y_1{=}X_1 \times Y_1$	M
6	$F{=}A + Z_1$	$B{=}A - Z_1$	a
7	$E{=}E - B$	$Y_1{=}Y_1 - F$	a
8	$Z_1{=}E \times Y_1$	$X_1{=}E \times F$	M
9	$Y_1{=}Y_1 \times B$	$A{=}X_1 \times X_1$	M
10	$Z_1{=}Z_1^2$	$B{=}Y_1^2$	S
11	$Z_1{=}Z_1 + Z_1$	$C{=}A + B$	a
12	$B{=}B - A$	$X_1{=}X_1 + Y_1$	a
13	$Y_1{=}B \times C$	$X_1{=}X_1 \times X_1$	M
14	$B{=}Z_1 - C$	$X_1{=}X_1 - C$	a
15	$Z_1{=}B \times C$	$X_1{=}X_1 \times B$	M
			7M+1S+7a

The detailed formulas are given in Fig. 1, Fig. 2, and Fig. 3. The input to all algorithms is the intermediate point, given in projective coordinates $(X_1 : Y_1 : Z_1)$; the algorithms involving additions also take a second point in affine coordinates (x_2, y_2) as input. The variables x_2, y_2 are read-only; the variables X_1, Y_1, Z_1 are modified to store the result. We have tested the formulas against those in the EFD [9] and ensured that there would be no concurrent reads/writes by testing the stated version and the one with the roles of MAU 1 and MAU 2 swapped. The horizontal lines indicate the beginning of the second operation. There are no idle multiplication stages and only in DBL-mADD there is a wait stage for an addition; another addition stage is used for a copy, which can be implemented as an addition $Z_1 = X_1 + 0$. So the pipelined algorithms achieve essentially perfect parallelism.

We note that in our current implementation, concurrent execution of a squaring and a multiplication does not result in any performance penalty since squaring is implemented as multiplication of the number by itself. Even if squarings could be executed somewhat faster than general multiplications the performance loss is minimal, e.g., instead of needing 3M+4S per doubling, the pipelined DBL-DBL formulas need 4M+3S per doubling.

We also kept the number of extra variables to a minimum. The pipelined versions need 1 extra variable compared to the versions on a single processor but now two processors share the computation. This frees up enough memory so that we can store the 8 points $\tilde{P}, [3]\tilde{P}, [5]\tilde{P}, \ldots, [15]\tilde{P}$ per curve. We store these points in affine coordinates using only 2 $\mathbf{Z}/m$ elements' worth of storage space. With these precomputations we can use a signed-sliding-window method

Fig. 3. Explicit formulas for DBL-mADD.

Step	MAU 1	MAU 2	
1	$A{=}X_1^2$	$B{=}Y_1^2$	S
2	$X_1{=}X_1 + Y_1$	$C{=}A + B$	a
3	$X_1{=}X_1^2$	$Z_1{=}Z_1^2$	S
4	$X_1{=}X_1 - C$	$Z_1{=}Z_1 + Z_1$	a
5	$B{=}B - A$	$Z_1{=}Z_1 - C$	a
6	$X_1{=}X_1 \times Z_1$	$Y_1{=}B \times C$	M
7	$Z_1{=}Z_1 \times C$	$A{=}X_1 \times Y_1$	M
8	$B{=}x_2 \times Z_1$	$C{=}y_2 \times Z_1$	M
9	$E{=}X_1 - B$	$F{=}Y_1 + C$	a
10	$X_1{=}X_1 + C$	$Y_1{=}Y_1 + B$	a
11	$E{=}E \times F$	$Z_1{=}B \times C$	M
12	$F{=}A + Z_1$	$B{=}A - Z_1$	a
13	$E{=}E - B$	$Z_1{=}X_1$	a
14	$A{=}Z_1 \times Y_1$	$X_1{=}E \times F$	M
15	$A{=}A - F$		a
16	$Z_1{=}E \times A$	$Y_1{=}A \times B$	M
			6M+2S+8a

to compute $[s]\tilde{P}$. This reduces the number of mixed additions to an average of $(\log_2 s)/6$ (and worst case of $(\log_2 s)/5$).

6 Experimental Results

We summarize the experimental results in Tables 1 and 2. Our experiments consist of running stage-1 ECM with B_1 ranging from 2^{10} to 2^{20} on integers m of 280 bits on various CPUs and GPUs. For CPU experiments, we run GMP-ECM, the state-of-the-art implementation of ECM. Details on the implementation on the GPUs are given in Sections 4 and 5.

Note that for some GPUs, we have serial and parallel ECM implementations, in which case both are presented in the table. We are unable to make parallel ECM to run on G80 and G92 because they do not have enough registers to accommodate the more complicated control code. The bottommost row represents the situation in which we use CPUs and GPUs simultaneously for ECM computations.

For each implementation we derive two sets of performance numbers based on cycle-accurate measurements of ECM execution time: the per-second throughput of modular multiplication, and the per-second throughput of elliptic-curve scalar multiplication. In some GPU experiments we are unable to obtain such cycle-accurate measurement when B_1 is too large because of overflow in the built-in clock cycle counter. However, we have verified that for a fixed coprocessor, the modular-multiplication throughput roughly remains the same for different B_1's

Table 1. Performance results of stage-1 ECM.

Coprocessor	#Cores	Freq (GHz)	Rmax (GFLOPS)	Mulmods (10^6/sec)	Curves (1/sec)
CHES 2008 [45] (scaled)	96	1.2	230.4		26.81
8800 GTS (G80)	96	1.2	230.4	7.51	57.30
8800 GTS (G92)	128	1.625	416.0	13.64	104.14
GTX 260	192	1.242	476.9	14.97	119.05
GTX 280	240	1.296	622.1	19.53	155.29
Core 2 Duo E6850	2	3.0	48.0	11.19	107.14
Core 2 Quad Q6600	4	2.4	76.8	17.91	171.42
Core 2 Quad Q9550	4	2.83	90.7	21.77	208.39
GTX 260 (parallel)	192	1.242	476.9	16.61	165.58
GTX 280 (parallel)	240	1.296	622.1	22.66	216.78
Q6600+GTX 280×2				63.23	604.99

and can be used to accurately predict the scalar-multiplication throughput when multiplied by the number of modular multiplications executed in each scalar multiplication. Therefore, in Table 1 we only report the result obtained with $B_1 = 8192$, the largest B_1 for which we have obtained cycle-accurate measurements in all experiments — except for the modular-multiplication throughput of 8800 GTS (both G80 and G92), where the measurement is done using the clock cycle counter on the CPU, and therefore it also includes the overhead of setting up the computation and returning the computed result. For elliptic-curve scalar multiplications, this overhead is negligible. But for modular multiplications, this overhead can be significant, so we use $B_1 = 1048576$ when calculating the modular-multiplication throughput.

In Table 1, the first column lists the coprocessors. The next three columns list their specifications: number of cores, clock frequency, and theoretical maximal arithmetic throughput. Note that this throughput figure tends to underestimate CPU's computational power while overestimating GPU's because CPUs have wider data paths while GPUs suffer from lower degree of instruction-level parallelism. The next two columns give the actual performance numbers derived from measurements.

We note that the first row in Table 1 does not correspond to any experiments we have performed. It is extrapolated based on the result of Szerwinski and Güneysu, published in CHES 2008 [45]. In their result, the scalar in the scalar multiplications is 224 bits long, whereas in our experiments, it is 11797 bits long. Therefore, we have scaled their result by 11797/224 to fit into our context. We also note that their modulus is a special prime, which leads to faster modular reduction and that it has only 224 bits, as opposed to 280 in our implementations. We did not account for this difference in the performance figure stated. In spite of that, our implementation on the same platform achieves a significantly higher throughput of more than twice the number of curves per second.

In Section 4.3 we estimated that a modular multiplication on the GTX 280 would consume 281 instructions. Dividing the Rmax of the GTX 280 in Table 1 by the achieved modular multiplication throughput, we see that in the implementation each modular multiplication requires about 27454 floating-point operations, which can be delivered by 13727 GPU instructions. Given that we use 32 threads to compute one single modular multiplication, each thread gets to execute about 429 instructions per modular multiplication. This number is about 1.5 times the instruction budget we derived before. We think the difference is due to the fact that there are other minor operations associated with each modular multiplication such as modular additions and subtractions, as well as other managerial operations like data movement and address calculations.

Table 2. Price-performance results of stage-1 ECM.

Coprocessor	Component-wise Cost ($)	Component-wise performance/cost (1/(sec·$))	System-wise Cost ($)	System-wise performance/cost (1/(sec·$))
8800 GTS (G80)	119	0.48	1006	0.1139
8800 GTS (G92)	178	0.59	1124	0.1853
GTX 260	250	0.48	1268	0.1878
GTX 280	400	0.39	1568	0.1981
Core 2 Duo E6850	200	0.54	972	0.1102
Core 2 Quad Q6600	185	0.93	957	0.1791
Core 2 Quad Q9550	325	0.64	1097	0.1900
GTX 260 (parallel)	250	0.66	1268	0.2612
GTX 280 (parallel)	400	0.54	1568	0.2765
Q6600+GTX 280×2	985	0.61	1889	0.3203

Table 2 shows price-performance figures. For each processor it states the cheapest list price pulled from the online retail giant NewEgg.com (on September 3, 2008), which in turn gives the per-US-dollar scalar-multiplication throughput listed in the next column. This performance-cost ratio can be misleading because one could not possibly compute ECM with a bare CPU or GPU — one would definitely need a complete computer system with a motherboard, power supply, etc. Therefore, we give the per-US-dollar scalar-multiplication throughput for entire ECM computing systems, based on the advice given by a web site for building computer systems of better performance-cost ratio [6]. The baseline configuration consists of one dual-PCIe motherboard and one 750 GB hard drive packed in a desktop enclosure with a built-in 430-Watt power supply and several cooling fans. For CPU systems, we install the CPU under consideration with its price updated from NewEgg.com if necessary, plus 8 GB of ECC RAM and a cheap video card. In contrast, for GPU systems, we install *two* identical graphic cards since the motherboard can take two video cards. We also add a 700-Watt power supply in order to provide enough power for the two graphic cards, plus

a lower-priced Celeron CPU with only 2 GB of ECC RAM. This is justified because we use GPUs for ECM computation and thus only use the CPU for light managerial tasks. Finally, the configuration in the last row has both CPU and GPU working as the ECM computing engine. Unsurprisingly this achieves the best performance-cost ratio since the cost of the supporting hardware is shared by both CPU and GPUs. Also not surprisingly, multi-socket motherboards with Opterons and Xeons are simply too expensive.

From Table 2, we can see that although the Core 2 Quad Q6600 has an unbeatably high performance-cost ratio as a component — there is often such a "sweet spot" in market pricing for a high-end-but-not-quite-highest part, especially toward the end of its life — the configuration of two GTX 280's actually achieves a higher performance-cost ratio system-wise, not to mention that it can be aided by a CPU to achieve an even better performance-cost ratio. To our knowledge, this is the first GPU implementation of elliptic-curve computations in which the GPU results are better in the number of scalar multiplications per dollar and per second.

References

1. — (no editor), *13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), 17–20 April 2005, Napa, CA, USA*, IEEE Computer Society, 2005. ISBN 0-7695-2445-1. See [43].
2. Kazumaro Aoki, Jens Franke, Thorsten Kleinjung, Arjen K. Lenstra, Dag Arne Osvik, *A Kilobit Special Number Field Sieve Factorization*, in Asiacrypt 2007 [31] (2007), 1–12. Cited in §1, §1.
3. A. O. L. Atkin, Francois Morain, *Finding suitable curves for the elliptic curve method of factorization*, Mathematics of Computation **60** (1993), 399–405. ISSN 0025-5718. MR 93k:11115. URL: `http://www.lix.polytechnique.fr/~morain/Articles/articles.english.html`. Cited in §2.2.
4. Friedrich Bahr, Michael Boehm, Jens Franke, Thorsten Kleinjung, *Subject: rsa200* (2005). URL: `http://www.crypto-world.com/announcements/rsa200.txt`. Cited in §1.
5. Friedrich Bahr, Jens Franke, Thorsten Kleinjung, *Discrete logarithms in GF(p) – 160 digits* (2007). URL: `http://www.nabble.com/Discrete-logarithms-in-GF(p)-----160-digits-td8810595.html`. Cited in §1.
6. Daniel J. Bernstein, *How to build the 2007.12.03 standard workstation*. URL: `http://cr.yp.to/hardware/build-20071203.html`. Cited in §6.
7. Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, Christiane Peters, *Twisted Edwards Curves*, in Africacrypt [46] (2008), 389–405. URL: `http://eprint.iacr.org/2008/013`. Cited in §2.2.
8. Daniel J. Bernstein, Peter Birkner, Tanja Lange, Christiane Peters, *ECM using Edwards curves* (2008). URL: `http://eprint.iacr.org/2008/016`. Cited in §2, §2.2, §2.2, §2.2.
9. Daniel J. Bernstein, Tanja Lange, *Explicit-formulas database* (2008). URL: `http://hyperelliptic.org/EFD`. Cited in §2.2, §5.
10. Daniel J. Bernstein, Tanja Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007 [31] (2007), 29–50. URL: `http://cr.yp.to/papers.html#newelliptic`. Cited in §2.2, §2.2.

11. Dan Boneh (editor), *Advances in Cryptology — CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17–21, 2003*, Lecture Notes in Computer Science, 2729, Springer, 2003. ISBN 3-540-40674-3. See [42].
12. Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Paul C. Leyland, Walter M. Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Paul Zimmermann, *Factorization of RSA-140 Using the Number Field Sieve*, in Asiacrypt 1999 [33] (1999), 195–207. Cited in §1.
13. Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter M. Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul C. Leyland, Joël Marchand, Francois Morain, Alec Muffett, Chris Putnam, Craig Putnam, Paul Zimmermann, *Factorization of a 512-Bit RSA Modulus*, in Eurocrypt 2000 [40] (2000), 1–18. Cited in §1, §1.
14. Debra L. Cook, John Ioannidis, Angelos D. Keromytis, Jake Luck, *CryptoGraphics: Secret Key Cryptography Using Graphics Cards*, in CT-RSA 2005 [36] (2005), 334–350. Cited in §3.
15. Debra L. Cook, Angelos D. Keromytis, *CryptoGraphics: Exploiting Graphics Cards For Security*, Advances in Information Security, 20, Springer, 2006. ISBN 978-0-387-29015-7. Cited in §3.
16. James Cowie, Bruce Dodson, R. Marije Elkenbracht-Huizing, Arjen K. Lenstra, Peter L. Montgomery, Jörg Zayer, *A World Wide Number Field Sieve Factoring Record: On to 512 Bits*, in Asiacrypt 1996 [28] (1996), 382–394. Cited in §1.
17. Cynthia Dwork (editor), *Advances in Cryptology — CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20–24, 2006*, Lecture Notes in Computer Science, 4117, Springer, 2006. ISBN 3-540-37432-9. See [27].
18. Harold M. Edwards, *A normal form for elliptic curves*, Bulletin of the American Mathematical Society **44** (2007), 393–422. URL: `http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/home.html`. Cited in §2.2.
19. Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, Colin Stahlke, *SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers*, in CHES 2005 [41] (2005), 119–130. Cited in §1, §1.
20. Kris Gaj, Soonhak Kwon, Patrick Baier, Paul Kohlbrenner, Hoang Le, Mohammed Khaleeluddin, Ramakrishna Bachimanchi, *Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware*, in CHES 2006 [23] (2006), 119–133. Cited in §1.
21. Steven D. Galbraith (editor), *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18–20, 2007*, Lecture Notes in Computer Science, 4887, Springer, 2007. ISBN 978-3-540-77271-2. See [37].
22. Willi Geiselmann, Adi Shamir, Rainer Steinwandt, Eran Tromer, *Scalable Hardware for Sparse Systems of Linear Equations, with Applications to Integer Factorization*, in CHES 2005 [41] (2005), 131–146. Cited in §1.
23. Louis Goubin, Mitsuru Matsui (editors), *Cryptographic Hardware and Embedded Systems — CHES 2006, 8th International Workshop, Yokohama, Japan, October 10–13, 2006*, Lecture Notes in Computer Science, 4249, Springer, 2006. ISBN 3-540-46559-6. See [20].
24. Florian Hess, Sebastian Pauli, Michael E. Pohst (editors), *Algorithmic Number Theory, 7th International Symposium, ANTS-VII, Berlin, Germany, July 23–28, 2006*, Lecture Notes in Computer Science, 4076, Springer, Berlin, 2006. ISBN 3-540-36075-1. See [47].

25. Huseyin Hisil, Kenneth Wong, Gary Carter, Ed Dawson, *Faster group operations on elliptic curves* (2007). URL: `http://eprint.iacr.org/2007/441`. Cited in §2.2.
26. Antoine Joux, Reynald Lercier, *Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the Gaussian integer method*, Mathematics of Computation **72** (2003), 953–967. Cited in §1.
27. Antoine Joux, Reynald Lercier, Nigel P. Smart, Frederik Vercauteren, *The Number Field Sieve in the Medium Prime Case*, in Crypto 2006 [17] (2006), 326–344. Cited in §1.
28. Kwangjo Kim, Tsutomu Matsumoto (editors), *Advances in Cryptology — ASIACRYPT '96, International Conference on the Theory and Applications of Cryptology and Information Security, Kyongju, Korea, November 3–7, 1996*, Lecture Notes in Computer Science, 1163, Springer, 1996. ISBN 3-540-61872-4. See [16].
29. Thorsten Kleinjung, *Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024-bit integers*, in Proceedings of SHARCS'06 (2006). URL: `http://www.math.uni-bonn.de/people/thor/cof.ps`. Cited in §1, §1.
30. Neal Koblitz, Alfred Menezes, *Pairing-Based Cryptography at High Security Levels*, in Coding and Cryptography [44] (2005), 13–36. Cited in §1.
31. Kaoru Kurosawa (editor), *Advances in cryptology — ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2–6, 2007*, Lecture Notes in Computer Science, 4833, Springer, 2007. See [2], [10].
32. Chi-Sung Laih (editor), *Advances in Cryptology — ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 – December 4, 2003*, Lecture Notes in Computer Science, 2894, Springer, 2003. ISBN 3-540-20592-6. See [35].
33. Kwok-Yan Lam, Eiji Okamoto, Chaoping Xing (editors), *Advances in Cryptology — ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14–18, 1999*, Notes in Computer Science, 1716, Springer, 1999. ISBN 3-540-66666-4. See [12].
34. Hendrik W. Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), 649–673. ISSN 0003-486X. MR 89g:11125. URL: `http://links.jstor.org/sici?sici=0003-486X(198711)2:126:3<649:FIWEC>2.0.CO;2-V`. Cited in §1.
35. Arjen K. Lenstra, Eran Tromer, Adi Shamir, Wil Kortsmit, Bruce Dodson, James Hughes, Paul C. Leyland, *Factoring Estimates for a 1024-Bit RSA Modulus*, in Asiacrypt 2003 [32] (2003), 55–74. Cited in §1.
36. Alfred J. Menezes (editor), *Topics in Cryptology — CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14–18, 2005*, Lecture Notes in Computer Science, 3376, Springer, 2005. ISBN 3-540-24399-2. See [14].
37. Andrew Moss, Dan Page, Nigel P. Smart, *Toward Acceleration of RSA Using 3D Graphics Hardware*, in Cryptography and Coding 2007 [21] (2007), 364–383. Cited in §3.
38. Elisabeth Oswald, Pankaj Rohatgi (editors), *Cryptographic Hardware and Embedded Systems — CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10–13, 2008*, Lecture Notes in Computer Science, 5154, Springer, 2008. ISBN 978-3-540-85052-6. See [45].

39. Jan Pelzl, Martin Šimka, Thorsten Kleinjung, Jens Franke, Christine Priplata, Colin Stahlke, Miloš Drutarovský, Viktor Fischer, Christof Paar, *Area-time efficient hardware architecture for factoring integers with the elliptic curve method*, IEE Proceedings on Information Security **152** (2005), 67–78. Cited in §1.
40. Bart Preneel (editor), *Advances in Cryptology — EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14–18, 2000*, Lecture Notes in Computer Science, 1807, Springer, 2000. ISBN 3-540-67517-5. See [13].
41. Josyula R. Rao, Berk Sunar (editors), *Cryptographic Hardware and Embedded Systems — CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 – September 1, 2005*, Lecture Notes in Computer Science, 3659, Springer, 2005. ISBN 3-540-28474-5. See [19], [22].
42. Adi Shamir, Eran Tromer, *Factoring Large Numbers with the TWIRL Device*, in Crypto 2003 [11] (2003), 1–26. Cited in §1.
43. Martin Šimka, Jan Pelzl, Thorsten Kleinjung, Jens Franke, Christine Priplata, Colin Stahlke, Miloš Drutarovský, Viktor Fischer, *Hardware Factorization Based on Elliptic Curve Method*, in FCCM 2005 [1] (2005), 107–116. Cited in §1.
44. Nigel P. Smart (editor), *Cryptography and Coding, 10th IMA International Conference, Cirencester, UK, December 19–21, 2005*, Lecture Notes in Computer Science, 3796, Springer, 2005. See [30].
45. Robert Szerwinski, Tim Güneysu, *Exploiting the Power of GPUs for Asymmetric Cryptography*, in CHES 2008 [38] (2008), 79–99. Cited in §3.1, §1, §6.
46. Serge Vaudenay (editor), *Progress in Cryptology — AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11–14, 2008*, Lecture Notes in Computer Science, 5023, Springer, 2008. ISBN 978-3-540-68159-5. See [7].
47. Paul Zimmermann, Bruce Dodson, *20 Years of ECM*, in ANTS 2006 [24] (2006), 525–542. Cited in §2.
48. Paul Zimmermann, *50 largest factors found by ECM*. URL: `http://www.loria.fr/~zimmerma/records/top50.html`. Cited in §1.