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, 1 Section 4 Roosevelt Road, 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, 926.11 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 124.71 curves/second for ECM stage 1 with $B_1 = 8192$ for 280-bit integers using all four cores of a 2.4 GHz Core 2 Quad Q6600.

The extra speed takes advantage of extra hardware, specifically two NVIDIA GTX 295 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 295 the implementation performs 41.88 million modular multiplications per second for a general 280-bit modulus. GMP-ECM, using all four cores of a Q6600, performs 13.03 million modular 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 ($2^{224} - 2^{96} + 1$).

* Permanent ID of this document: 6904068c52463d70486c9c68ba045839. Date of this document: 2009.01.26. This work was sponsored in part by the National Science Foundation under grant ITR–0716498, in part by Taiwan's National Science Council (grants NSC-96-2221-E-001-031-MY3 and -96-2218-E-001-001, also through the Taiwan Information Security Center NSC-97-2219-E-001-001, -96-2219-E-011-008), and in part by the European Commission through the ICT Programme under Contract ICT–2007–216676 ECRYPT II. Part of this work was carried out while Bernstein and Lange visited NTU.

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

1 Introduction

The elliptic-curve method (ECM) 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 [49]).

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 [43], [35], [19], [22], [44], 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 special 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 [40] 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 “cores” running at clock frequencies not much lower than that of state-of-the-art CPUs; e.g., the NVIDIA GeForce 8800 GTS 512 has 128 cores running 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 41.88 million 280-bit modular multiplications per second and has a significantly 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 ECM is given by Zimmermann and Dodson in [48]. 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 $s = \text{lcm}(1, 2, 3, \dots, B_1)$. 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 $R_1 = [p_{k+1}]R, R_2 = [p_{k+2}]R, \dots, R_\ell = [p_{k+\ell}]R$ where $p_{k+1}, p_{k+2}, \dots, p_{k+\ell}$ are the primes between B_1 and another bound 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 showed 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

$$(x_1, y_1) \oplus (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right).$$

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 speedup 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 Review of GPUs and GPU Programming

Today's graphics cards contain powerful GPUs to handle the increasing complexity and screen resolution in video games. GPUs have now developed into a powerful, highly parallel computing platform that finds 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 [38].

Along with the G80 series of GPUs, NVIDIA introduced CUDA, a parallel computing framework with a C-like programming language specifically intended for compilation and execution on a GPU. In this section we describe current NVIDIA graphics cards used for our implementations, give some background information on CUDA programming, and compare NVIDIA GPUs to AMD GPUs.

3.1 The NVIDIA Cards Used for CUDA

An NVIDIA GPU contains many streaming multiprocessors (MPs), each of which contains the following elements:

- a scheduling and dispatching unit that can handle many lightweight threads;
- eight (8) "cores" (often called streaming processors, or SPs) each capable of a fused single-precision floating-point multiply-and-add (MAD), or otherwise one 32-bit integer add/subtract or logical operation every cycle;
- two (2) "super function units" that each can do various complex computations like 32-bit integer multiplications, floating-point divisions, or *two (2) single-precision floating-point multiplications per cycle*;
- *for the more advanced GT2xx GPUs*, additional circuitry that in conjunction with the SPs can do double-precision floating-point arithmetic, albeit with a lower throughput (roughly 1/6 of that of single-precision counterpart);
- fast local shared memory, 16 banks of 1 kB each;
- controllers to access uncached thread-local and global memory;
- fast local read-only cache to device memory on the card, up to 8 kB;
- fast local read-only cache on, and access to, a *texture unit* (2 MPs on a G8x or G9x, and 3 MPs on a GT2xx form a cluster sharing a texture unit);
- a file of 8192 (for G8x or G9x) or 16384 (for GT2xx) 32-bit registers.

Uncached memory has a relatively low throughput and long latency. For example, the 128 SPs on a GeForce 8800 GTX run at 1.35 GHz, and the uncached memory provides a throughput of 86.4 GB/s. That may sound impressive but it is only a single 32-bit floating-point number per cycle per MP, with a latency of 400–600 cycles to boot.

The GPU can achieve more impressive data movement by broadcasting the same data to many threads in a cycle. The shared memory in an MP can deliver 64 bytes every two cycles, or 4 bytes per cycle per SP *if there is no bank conflict.* Latencies of all caches and shared memories are close to that of registers and hence much lower than device memories.

G8x/G9x Series. The chip used in the GeForce 8800 GTX is a typical NVIDIA G80-series GPU, a 90nm-process GPU containing 128 SPs grouped into 16 MPs.

G92 GPUs are a straightforward die shrink of the G80 to a 65nm process and were used in the GeForce 8800 GTS 512 (16 MPs, not to be confused with the "8800 GTS 384", a G80) and 9800-series cards, e.g., the 9800 GTX (16 MPs) and 9800 GX2 (two 9800 GTX's on a PCI Express bridge).

GPUs codenamed G84/G85/G86 are NVIDIA's low-end parts of the G80 series, with the same architecture but only 1–4 MPs and much lower memory throughput. Similarly, G94/G96 describe low-end versions of the G92. These are never cost-effective for our purposes (except maybe testing code on the road).

Note: Manufacturers often sell a top-end, envelope-pushing chip at a huge markup, and slightly weaker chips (often just a batch failing a quality control binning) at far more reasonable prices. The lower-priced G80s (e.g., the "8800 GTS 384", 12 MPs, used in [46], or the older 8800 GT with 14 MPs) with slightly lower clock rates, fewer functional units, and lower memory throughput can achieve better price-performance ratio than the top-end G80s.

GT2xx Series. The GT2xx series started out at the same 65nm process as the G92, with a new and improved design. The GTX 260 and the GTX 280 both run at a slightly lower clock rate than the G92 but the GTX 260 has 24 MPs and the GTX 280 has 30 MPs, almost twice as many as the G92. GT2xx GPUs are also better in other ways. In particular, the size of the register file is doubled, which is very helpful for our implementation.

The GTX 285 and 295 were introduced early in 2009, shortly before the time of this writing. Our initial tests are consistent with reports that (a) the 285 is a simple die-shrink of the 280 to a 55nm process, and (b) the 295 is just two underclocked 285's bolted together.

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 CUDA driver, and finally loaded and run on the GPU.

CUDA adopts a super-threaded, massively parallel computation model, in which computation is divided into many (typically thousands of) threads. A pool of physical processing units (e.g., the 128 SPs in G80) then executes these threads in a seemingly concurrent fashion. This time-sharing of physical units by many threads or computations is necessary because the instruction latency is high: a typical instruction takes 20–24 clock cycles to execute in its entirety. Because the SPs are fully pipelined, with enough instructions "in flight", we can hide this latency and approach the theoretical limit of one dispatched instruction per cycle per SP. CUDA manuals suggest a minimum of 192 threads per MP. This can be understood as 8 SPs $\times$ 24 stages $= 192$ in order to hide instruction latency completely.

Modern CPUs do a lot more than pipelining. They actively search for independent instructions to issue in a program stream, *dispatching them out of order if needed*. However, out-of-order execution requires a lot of extra circuitry. NVIDIA has opted instead to make its chips completely in-order, hence CUDA mostly utilizes what is called *thread-level* (in contrast with instruction-level) parallelism.

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 an MP to issue an instruction for a warp of threads (16 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.

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 physical registers are divided among all active threads. This creates 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 could only use 32 registers, a tight budget for implementing complicated algorithms. The situation improved with the GT2xx family having twice as many registers, relieving the register pressure and making programming much easier.

To summarize, the massive parallelism in NVIDIA's GPU architecture makes programming on graphics cards very different from sequential programming on a traditional CPU. 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 minimize the number of memory accesses and meticulously arrange the parallel execution of hardware threads to avoid resource contention such as bank conflict in memory access.

3.3 Limitations and Alternatives

Race Conditions and Synchronization. A pitfall frequently encountered when programming multiple threads is race conditions. In CUDA, threads are organized into "blocks" so that threads belonging to the same block execute on the same MP and time-share the SPs on a per-instruction, round-robin fashion. Sometimes, the execution of a block of threads will need to be serialized when there is resource contention, e.g., when accessing device memory, or accessing shared memory when there is a 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 stream. Another use of this primitive is to set up synchronization barriers. Without such barriers, the optimizing compiler can sometimes reorder the instructions too aggressively, resulting in race conditions when the code is executed concurrently by a block of threads.

Pressure on Fast On-die Memories. A critically limited GPU resource is memory — in particular, fast memory — including per-thread registers and per-MP shared memory. For example, 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 this 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 L2 cache in an aged Athlon 64 (1.6×10^8 gates)! Unfortunately, CUDA requires many more (NVIDIA recommends 24 times the number of SPs) threads 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.

A Brief Comparison to AMD GPUs The main competition to NVIDIA's GeForce is Radeon from AMD (formerly ATI). The AMD counterpart to CUDA is Brook+, also a C/C++-derived language. Brook+ programming is similar to CUDA programming: GPU programs ("shaders") are compiled into intermediate code, which is converted on the fly into machine instructions. See Table 1 for a comparison between current GeForce and Radeon GPUs.

Table 1. Comparison of Leading NVIDIA and AMD Video Cards.

NVIDIA/AMD Lead GPU Series	NVIDIA GT200	AMD RV770
Top configuration	GeForce GTX 295	Radeon 4870x2
Arithmetic clock	1250 MHz	750 MHz
Registers per MP (or SIMD core)	16k $\times$ 32-bit	16k $\times$ 128-bit
#MPs / #SPs	$2 \times (30 \times 8)$	$2 \times (10 \times 16(\times 5))$
Registers on each chip	491,520 (1.875MB)	163,840 (2.5MB)
Local store per MP/SIMD core	16 kB	16 kB
Global store per chip	None	16 kB
Max threads on chip	30,720	16,384
Max threads per MP	1,024	$> 1,000$

GeForce and Radeon have different hardware and software models. Recall that each GeForce MP has 8 SPs, each with a single-precision floating-point fused multiplier-adder, plus 2 super function units, which can dispatch 4 single-precision multiplications per cycle. The Radeon equivalent of an MP, called an "SIMD core", has 16 VLIW (very long instruction word) SPs, each of which is capable of delivering 5 single-precision floating-point operations every cycle. Pipelines on the Radeon are around a dozen stages deep, half as long as those on the GeForce.

Overall the two architectures pose similar challenges: there are many threads but very little fast memory available to each thread. A naïve calculation suggests that to hide the latency of arithmetic operations one must schedule $16 \times 12 = 192$ threads per SIMD core with a Radeon, and 192 threads per MP with a GeForce, so the number of registers per thread is similar for both architectures.

As a Radeon SIMD core does 80 floating-point operations per cycle to a GeForce MP's 20, but has at most 32 kB of scratch memory vs. 16 kB for the GeForce MP, one can expect that a program for a Radeon would be more storage-starved than for a GeForce. We plan to investigate Radeon cards as an alternative to CUDA and GeForce cards, but our initial estimate is that

ECM's storage pressure makes GeForce cards more suitable than Radeon cards for ECM.

4 High-Throughput Modular Arithmetic on a GPU

Modular arithmetic is the main bottleneck in computing scalar multiplication in ECM. In this section we describe our implementation of modular arithmetic on a GPU, focusing specifically on modular multiplication, the rate-determining mechanism in ECM. We will explain the design choices we have made and show how parallelism is used on this level.

4.1 Design Choices of Modular Multiplication

For our target of 280-bit integers, schoolbook multiplication needs less intermediate storage space and synchronization among cooperative threads than the more advanced algorithms such as Karatsuba. Moreover, despite requiring a smaller number of word multiplications, Karatsuba multiplication is slower on GPUs because there are fewer pairs of multiplications and additions that can be merged into single MAD instructions, resulting in a higher instruction count. It is partly for this reason that we choose to implement the modular multiplier using floating-point arithmetic as opposed to 32-bit integer arithmetic, which does not have the fused multiply-and-add instruction; another reason is that floating-point multiplication currently has a higher throughput on NVIDIA GPU than its 32-bit integer counterpart.

We represent an integer using L limbs in radix 2^r, with each limb stored as a floating-point number between -2^{r-1} and 2^{r-1}. This allows us to represent any integer between $-R/2$ and $R/2$, where $R = 2^{Lr}$. We choose to use Montgomery representation [37] of the integers modulo m, where m is the integer to be factored by ECM, and thus represent $x \bmod m$ as $x' \equiv Rx \pmod{m}$. Note that our limbs can be negative, 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 $x' \equiv Rx \pmod{m}$ and $y' \equiv Ry \pmod{m}$ 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 $\beta \equiv R(xy) \pmod{m}$.

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 MP with enough threads to effectively hide the instruction latency, we choose a block size of 256 threads; together such a block of threads is in charge of computing eight 280-bit arithmetic operations at a time. This means that we have an 8-way modular multiplier per MP. Each modular multiplication needs three 280-bit integer multiplications: one to obtain $x'y'$, one to obtain α, and the third to obtain β. Each of the three integer multiplications is carried out by 28 threads, each of which is responsible for cross-multiplying 7 limbs from one operand with 4 from the other. The reason why we do not use all 32 threads is clear now: because $\gcd(7, 4) = 1$, there can never be any bank conflict or race condition in the final stage when these 28 threads are accumulating the partial products in shared memory. Bank conflicts, on the other hand, can still occur when threads are loading the limbs into their private registers before computing the partial products, so we carefully arrange x' and y' from different curves in shared memory, inserting appropriate padding when necessary, to avoid all bank conflicts in accessing shared memory.

4.2 Instruction Count Analysis

We give an estimate of the instruction count of our design on GPUs. Recall that, in our design, each thread is responsible for cross-multiplying the limbs in a 7-by-4 region. In the inner loop of integer multiplication, each thread needs to load these limbs into registers (11 loads from on-die shared memory), multiply and accumulate them into temporary storage (28 MAD instructions), and then accumulate the result in a region shared by all 28 threads. That last part includes 10 load-and-adds, 10 stores, and 10 synchronization barriers (`__syncthreads`) to prevent the compiler from reordering instructions incorrectly. Together, it should take 69 instructions per thread (plus other 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 that is of twice the length, so we essentially need 4 full carries in each modular multiplication, resulting in 56 extra instructions per thread. This gives a total of 263 instructions per modular multiplication.

5 Fast ECM on a GPU

We now describe our implementation of ECM on a GPU using the modular multiplier 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 NFS add a further dimension in that factorizations of many auxiliary numbers are needed. We decided to use the parallelism of the GPU to handle several curves for a given auxiliary integer, which can thus be stored in the shared memory of an MP. All SPs in an MP 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 inside NFS can be handled by different MPs in a GPU or different GPUs in parallel since no communication is necessary among the factorizations. For the rest of this section we consider one fixed m and s for the computation on a single MP.

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

Fig. 1. Explicit formulas for DBL-DBL

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 an MP in a GPU into an 8-way modular arithmetic unit (MAU) that is capable of 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, so 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

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

Fig. 2. Explicit formulas for mADD-DBL

subsequent additions. At the very end of the scalar multiplication, one might encounter a single DBL or mADD, in that case one MAU is idle in the final multiplication.

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 one extra variable compared to the versions on a single MAU but now two MAUs share the computation. This frees up enough memory so that we can store the eight points $\tilde{P}, [3]\tilde{P}, [5]\tilde{P}, \ldots, [15]\tilde{P}$ per curve. We store these points in affine coordinates using only two $\mathbf{Z}/m$ elements' worth of storage

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

Fig. 3. Explicit formulas for DBL-mADD

space. With these precomputations we can use a signed-sliding-window method 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 our results in Tables 2 and 3. Our experiments consist of running stage-1 ECM on the product of two 140-bit prime numbers with B_1 ranging from 2^{10} to 2^{20} on various CPUs and GPUs. For CPU experiments, we run GMP-ECM, the state-of-the-art implementation of ECM, whereas for GPU experiments, we run our GPU ECM implementation as described in Sections 4 and 5.

The first column of each table lists the coprocessors. The next three columns list their specifications: number of cores, clock frequency, and theoretical maximal arithmetic throughput (Rmax). Note that the Rmax figures tend to underestimate CPUs' computational power while overestimating GPUs' because CPUs have wider data paths and are better at exploiting instruction-level parallelism. Also, in calculating GPUs' Rmax, we exclude the contribution from texture processing units because we do not use them. The next two columns give the actual performance numbers derived from our measurements.

Table 2 includes an extra row, the first row, that does not correspond to any experiments we have performed. This row is extrapolated from the result of Szerwinski and Güneysu published in CHES 2008 [46]. 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 throughput by 224/11797 to fit

Table 2. Performance results of stage-1 ECM

Coprocessor	#Cores	Freq (GHz)	Rmax (GFLOPS)	Mulmods (10^6/sec)	Curves (1/sec)
CHES 2008 [46] (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	7.85	75.17
Core 2 Quad Q6600	4	2.4	76.8	13.03	124.71
Core 2 Quad Q9550	4	2.83	90.7	14.85	142.17
GTX 260 (parallel)	192	1.242	476.9	16.61	165.58
GTX 280 (parallel)	240	1.296	622.1	22.66	216.78
GTX 295 (parallel)	480	1.242	1192.3	41.88	400.70
Q6600+GTX 295×2				96.79	926.11

into our context. We also note that their modulus is a special prime, which should lead to faster modular reduction, and that it only has 224 bits, as opposed to 280 in our implementation. 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, more than twice as many curves per second.

The remaining rows report two sets of performance numbers based on our cycle-accurate measurements of ECM execution time: per-second throughput of modular multiplication, and per-second throughput of elliptic-curve scalar multiplication with $B_1 = 8192$. For the GTX 260 and the GTX 280 we tried our ECM implementation using serial elliptic-curve arithmetic and our ECM implementation using parallel elliptic-curve arithmetic; both results are presented in the table. We are unable to make parallel arithmetic 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 the 8800 GTS (both G80 and G92), we used the CPU clock-cycle counter, so our scalar-multiplication measurements include the overhead of setting up the computation and returning the computed result. Our modular-multiplication measurements used separate experiments with $B_1 = 1048576$ to effectively eliminate this overhead. For the remaining GPUs we used the GPU clock-cycle counter, and used $B_1 = 8192$ in all cases to avoid overflow in the counter. By experimenting with additional choices of B_1 we have verified that, in all cases, modular-multiplication throughput roughly remains the same for different B_1's and thus can be used to accurately predict scalar-multiplication throughput given the number of modular multiplications executed in each scalar multiplication.

In Section 4.2 we have estimated that a modular multiplication needs at least 263 instructions. Take GTX 280 as an example: if we divide its Rmax in Table 2 by the achieved modular multiplication throughput, we see that in the experiment each

Table 3. 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	1005	0.1140
8800 GTS (G92)	178	0.59	1123	0.1855
GTX 260	275	0.43	1317	0.1808
GTX 280	334	0.46	1435	0.2164
Core 2 Duo E6850	172	0.44	829	0.0907
Core 2 Quad Q6600	189	0.66	847	0.1472
Core 2 Quad Q9550	282	0.50	939	0.1541
GTX 260 (parallel)	275	0.60	1317	0.2515
GTX 280 (parallel)	334	0.65	1435	0.3021
GTX 295 (parallel)	510	0.79	2001	0.4005
Q6600+GTX 295×2	1210	0.77	2226	0.4160

modular multiplication consumes about 27454 floating-point operations, which can be delivered in 13727 GPU instructions. Given that 32 threads are dedicated to computing one single modular multiplication, each thread gets to execute about 429 instructions per modular multiplication. This number is about 60% more than what we have estimated. We believe that the difference is due to the fact that there are other minor operations such as modular additions and subtractions, as well as managerial operations like data movement and address calculations.

Table 3 shows price-performance figures for different ECM coprocessors. For each coprocessor, the next column shows the cheapest retail price pulled from on-line vendors such as NewEgg.com as of January 23, 2009, which in turn gives the per-US-dollar scalar-multiplication throughput listed in the next column. This price-performance ratio can be misleading because one could not compute ECM with a bare CPU or GPU — one needs a complete computer system with a motherboard, power supply, etc. In the last column we give the per-US-dollar scalar-multiplication throughput for an entire ECM computing system, based on the advice given by a web site for building computer systems of good price-performance ratio [6]. The baseline configuration consists of one dual-PCI-Express motherboard and one 750 GB hard drive in a desktop enclosure with a built-in 430-Watt power supply and several cooling fans. For CPU systems, we include the CPU, 8 GB of ECC RAM, and a low-price graphics card. In contrast, for GPU systems we include *two* identical graphics cards (since the motherboard can take two video cards). We also add a 750-Watt (1200-Watt in the case of GTX 295) power supply in order to provide enough power for the two graphics cards, plus a lower-priced Celeron CPU and 2 GB of ECC RAM. This is justified because when we use GPUs for ECM computation, we use the CPU only for light, managerial tasks. Finally, the configuration in the last row has both CPU and GPU working on ECM, which achieves the best price-performance ratio since the cost of the supporting hardware is shared by both CPU and GPUs.

We did not consider multi-socket motherboards with Opterons or Xeons because they are not competitive in price-performance ratio.

We conclude that although the Q6600 has a very good price-performance ratio among Intel CPUs — there is often such a "sweet spot" in market pricing for a high-end (but not quite highest-end) part, especially toward the end of the product life — the configuration of two GTX 295's achieves a superior price-performance ratio both component- and system-wise, not to mention that they can be aided by a CPU to achieve an even better price-performance ratio. The cost-effectiveness of GPUs for ECM makes GPUs suitable as a component of designs such as SHARK and allows ECM cofactorization to play a larger role inside the number-field sieve. To our knowledge, this is the first GPU implementation of elliptic-curve computation in which the GPU results are better than CPU results in the number of scalar multiplications per dollar and per second.

References

1. 13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), Napa, CA, USA, April 17–20, 2005. IEEE Computer Society, Los Alamitos (2005); ISBN 0-7695-2445-1. See [44]
2. Aoki, K., Franke, J., Kleinjung, T., Lenstra, A.K., Osvik, D.A.: A Kilobit Special Number Field Sieve Factorization. In: ASIACRYPT 2007 [31], pp. 1–12 (2007) (Cited in §1, §1)
3. Atkin, A.O.L., Morain, F.: Finding suitable curves for the elliptic curve method of factorization. Mathematics of Computation 60, 399–405 (1993) ISSN 0025-5718. MR 93k:11115, `http://www.lix.polytechnique.fr/ morain/Articles/articles.english.html` (Cited in §2.2)
4. Bahr, F., Boehm, M., Franke, J., Kleinjung, T.: Subject: rsa200 (2005), `http://www.crypto-world.com/announcements/rsa200.txt` (Cited in §1)
5. Bahr, F., Franke, J., Kleinjung, T.: Discrete logarithms in GF(p) - 160 digits (2007), `http://www.nabble.com/Discrete-logarithms-in-GFp-----160-digits-td8810595.html`(Cited in §1)
6. Bernstein, D.J.: How to build the 2009.01.23 standard workstation, `http://cr.yp.to/hardware/build-20090123.html` (Cited in §6)
7. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards Curves. In: AFRICACRYPT [47], pp. 389–405 (2008), `http://eprint.iacr.org/2008/013` (Cited in §2.2)
8. Bernstein, D.J., Birkner, P., Lange, T., Peters, C.: ECM using Edwards curves (2008), `http://eprint.iacr.org/2008/016` (Cited in §2, §2.2, §2.2, §2.2)
9. Bernstein, D.J., Lange, T.: Explicit-formulas database (2008), `http://hyperelliptic.org/EFD` (Cited in §2.2, §5)
10. Bernstein, D.J., Lange, T.: Faster addition and doubling on elliptic curves. In: ASIACRYPT 2007 [31], pp. 29–50., `http://cr.yp.to/papers.html#newelliptic` (Cited in §2.2, §2.2)
11. Boneh, D. (ed.): CRYPTO 2003. LNCS, vol. 2729. Springer, Heidelberg (2003) See [43]
12. Cavallar, S., Dodson, B., Lenstra, A.K., Leyland, P., Lioen, W., Montgomery, P.L., Murphy, B., te Riele, H., Zimmermann, P.: Factorization of RSA-140 Using the Number Field Sieve. In: ASIACRYPT 1999 [33], pp. 195–207 (1999) (Cited in §1)

13. Cavallar, S., Dodson, B., Lenstra, A.K., Lioen, W., Montgomery, P.L., Murphy, B., te Riele, H., Aardal, K., Gilchrist, J., Guillerm, G., Leyland, P., Marchand, J., Morain, F., Muffett, A., Putnam, C., Putnam, C., Zimmermann, P.: Factorization of a 512-Bit RSA Modulus. In: EUROCRYPT 2000 [41], pp. 1–18 (2000) (Cited in §1, §1)
14. Cook, D.L., Ioannidis, J., Keromytis, A.D., Luck, J.: CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In: CT-RSA 2005 [36], pp. 334–350 (2005) (Cited in §3)
15. Debra, L., Cook, A.D.: CryptoGraphics: Exploiting Graphics Cards For Security. In: Advances in Information Security, vol. 20. Springer, Heidelberg (2006) (Cited in §3)
16. Cowie, J., Dodson, B., Elkenbracht-Huizing, R.M., Lenstra, A.K., Montgomery, P.L., Zayer, J.: A World Wide Number Field Sieve Factoring Record: On to 512 Bits. In: ASIACRYPT 1996 [28], pp. 382–394 (1996) (Cited in §1)
17. Dwork, C. (ed.): CRYPTO 2006. LNCS, vol. 4117. Springer, Heidelberg (2006) See [27]
18. Edwards, H.M.: A normal form for elliptic curves. Bulletin of the American Mathematical Society 44, 393–422 (2007), `http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/home.html` (Cited in §2.2)
19. Franke, J., Kleinjung, T., Paar, C., Pelzl, J., Priplata, C., Stahlke, C.: SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers. In: CHES 2005 [42], pp. 119–130 (2005) (Cited in §1, §1)
20. Gaj, K., Kwon, S., Baier, P., Kohlbrenner, P., Le, H., Khaleeluddin, M., Bachimanchi, R.: Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware. In: CHES 2006 [23], pp. 119–133 (2006) (Cited in §1)
21. Galbraith, S.D. (ed.): Cryptography and Coding 2007. LNCS, vol. 4887. Springer, Heidelberg (2007) See [38]
22. Geiselmann, W., Shamir, A., Steinwandt, R., Tromer, E.: Scalable Hardware for Sparse Systems of Linear Equations, with Applications to Integer Factorization. In: CHES 2005 [42], pp. 131–146 (2005) (Cited in §1)
23. Goubin, L., Matsui, M. (eds.): CHES 2006. LNCS, vol. 4249. Springer, Heidelberg (2006) See [20]
24. Hess, F., Pauli, S., Pohst, M. (eds.): ANTS 2006. LNCS, vol. 4076. Springer, Heidelberg (2006) See [48]
25. Hisil, H., Wong, K., Carter, G., Dawson, E.: Faster group operations on elliptic curves (2007), `http://eprint.iacr.org/2007/441` (Cited in §2.2)
26. Joux, A., Lercier, R.: Improvements to the general number field sieve for discrete logarithms in prime fields. A comparison with the Gaussian integer method, Mathematics of Computation 72, 953–967 (2003) (Cited in §1)
27. Joux, A., Lercier, R., Smart, N.P., Vercauteren, F.: The Number Field Sieve in the Medium Prime Case. In: CRYPTO 2006 [17], pp. 326–344 (2006) (Cited in §1)
28. Kim, K.-c., Matsumoto, T. (eds.): ASIACRYPT 1996. LNCS, vol. 1163. Springer, Heidelberg (1996) See [16]
29. Kleinjung, T.: Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024-bit integers. In: Proceedings of SHARCS (2006), `http://www.math.uni-bonn.de/people/thor/cof.ps` (Cited in §1, §1)
30. Koblitz, N., Menezes, A.: Pairing-Based Cryptography at High Security Levels. In: Coding and Cryptography [45], pp. 13–36 (2005) (Cited in §1)
31. Kurosawa, K. (ed.): ASIACRYPT 2007. LNCS, vol. 4833. Springer, Heidelberg (2007) See [2], [10]

32. Laih, C.-S. (ed.): ASIACRYPT 2003. LNCS, vol. 2894. Springer, Heidelberg (2003) See [35]
33. Lam, K.-Y., Okamoto, E., Xing, C. (eds.): ASIACRYPT 1999. LNCS, vol. 1716. Springer, Heidelberg (1999) See [12]
34. Lenstra Jr., H.W.: Factoring integers with elliptic curves. Annals of Mathematics 126, 649–673 (1987), `http://links.jstor.org/sici?sici=0003-486X1987112:126:3649:FIWEC2.0.CO;2-V` (Cited in §1)
35. Lenstra, A.K., Tromer, E., Shamir, A., Kortsmit, W., Dodson, B., Hughes, J., Leyland, P.C.: Factoring Estimates for a 1024-Bit RSA Modulus. In: ASIACRYPT 2003 [32], pp. 55–74 (2003) (Cited in §1)
36. Menezes, A. (ed.): CT-RSA 2005. LNCS, vol. 3376. Springer, Heidelberg (2005) See [14]
37. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44, 519–521 (1985), `http://www.jstor.org/pss/2007970` (Cited in §4.1)
38. Moss, A., Page, D., Smart, N.P.: Toward Acceleration of RSA Using 3D Graphics Hardware. In: Cryptography and Coding 2007 [21], pp. 364–383 (2007) (Cited in §3)
39. Oswald, E., Rohatgi, P. (eds.): CHES 2008. LNCS, vol. 5154. Springer, Heidelberg (2008) See [46]
40. Pelzl, J., Šimka, M., Kleinjung, T., Franke, J., Priplata, C., Stahlke, C., Drutarovský, M., Fischer, V., Paar, C.: Area-time efficient hardware architecture for factoring integers with the elliptic curve method. IEE Proceedings on Information Security 152, 67–78 (2005) (Cited in §1)
41. Preneel, B. (ed.): EUROCRYPT 2000. LNCS, vol. 1807. Springer, Heidelberg (2000) See [13]
42. Rao, J.R., Sunar, B. (eds.): CHES 2005. LNCS, vol. 3659. Springer, Heidelberg (2005) See [19], [22]
43. Shamir, A., Tromer, E.: Factoring Large Numbers with the TWIRL Device. In: CRYPTO 2003 [11], pp. 1–26 (2003) (Cited in §1)
44. Šimka, M., Pelzl, J., Kleinjung, T., Franke, J., Priplata, C., Stahlke, C., Drutarovský, M., Fischer, V.: Hardware Factorization Based on Elliptic Curve Method. In: FCCM 2005, pp. 107–116 (2005) (Cited in §1)
45. Smart, N.P. (ed.): Cryptography and Coding 2005. LNCS, vol. 3796. Springer, Heidelberg (2005) See [30]
46. Szerwinski, R., Güneysu, T.: Exploiting the Power of GPUs for Asymmetric Cryptography. In: CHES 2008 [39], pp. 79–99 (2008) (Cited in §3.1, §6, §2)
47. Vaudenay, S. (ed.): AFRICACRYPT 2008. LNCS, vol. 5023. Springer, Heidelberg (2008) See [7]
48. Zimmermann, P., Dodson, B.: 20 Years of ECM. In: ANTS 2006 [24], pp. 525–542 (2006) (Cited in §2)
49. Zimmermann, P.: 50 largest factors found by ECM, `http://www.loria.fr/~zimmerma/records/top50.html` (Cited in §1)