ECM at Work

Joppe W. Bos[1,*] and Thorsten Kleinjung[2]

[1] Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
[2] Laboratory for Cryptologic Algorithms, EPFL, Lausanne, Switzerland

Abstract. The performance of the elliptic curve method (ECM) for integer factorization plays an important role in the security assessment of RSA-based protocols as a cofactorization tool inside the number field sieve. The efficient arithmetic for Edwards curves found an application by speeding up ECM. We propose techniques based on generating and combining addition-subtracting chains to optimize Edwards ECM in terms of both performance and memory requirements. This makes our approach very suitable for memory-constrained devices such as graphics processing units (GPU). For commonly used ECM parameters we are able to lower the required memory up to a factor 55 compared to the state-of-the-art Edwards ECM approach. Our ECM implementation on a GTX 580 GPU sets a new throughput record, outperforming the best GPU, CPU and FPGA results reported in literature.

Keywords: Elliptic curve factorization, cofactorization, addition-subtraction chains, twisted Edwards curves, parallel architectures.

1 Introduction

Today, more than 25 years after its invention by Hendrik Lenstra Jr., the elliptic curve method [24] (ECM) remains the asymptotically fastest integer factorization method for finding relatively small prime factors of large integers. Although it is not the fastest general purpose integer factorization method, when factoring a composite integer $n = pq$ with $p \approx q \approx \sqrt{n}$ the number field sieve [32,23] (NFS) is asymptotically faster, it has recently received a renewed research interest due to the discovery of an interesting normal form for elliptic curves introduced by Edwards [13]. From a cryptologic point of view the practical performance of ECM is important since it is used to rapidly factor many small (up to one or two hundred bits) integers inside NFS. This is illustrated by the fact that it is estimated that five to twenty percent (cf. Section 2.2 why this is hard to estimate) of the total wall-clock time was spent in ECM in the current world-record factorization of a 768-bit RSA number [20] (and it is expected that this percentage will grow for larger factorizations). Using ECM as a tool to factor many small numbers inside NFS is an active research area by itself. Offloading this work to reconfigurable hardware such as field-programmable gate arrays is studied in [37,16,11,17,25,40] while [5,4]

---

considers parallel architectures such as graphics processing units (GPUs) and the Cell broadband engine architecture. A comparison between software and hardware based solutions is presented in [21]. Traditionally, ECM is implemented using Montgomery curves [26] and uses the various techniques described in [39]. The most-widely used ECM implementation is GMP-ECM [41] and this implementation, or modifications to it, is responsible for setting all recent ECM record factorizations (see a description of some of these record factorizations in [8]). After the invention of Edwards curves Bernstein et al. explored the possibility to use these curves in the ECM setting [3]. Hisil et al. [19] published a coordinate system for Edwards curves which results in the fastest known realization of curve arithmetic. A follow-up paper by Bernstein et al. discusses the usage of these "$a = -1$" twisted Edwards curves [1] for ECM. The speedup from switching to Edwards curves comes at a price, addition chains [35] (or addition-subtraction chains [28]) equipped with large windowing sizes [9] are used (cf. [6] for a summary of these techniques). The memory requirement for Edwards ECM grows roughly linearly with the input parameters of ECM while a small constant number of residues modulo n are sufficient when using Montgomery curves.

In this paper we optimize ECM by exploiting the fact that the same scalar is often used when computing the elliptic curve scalar multiplication (ECSM), allowing one to prepare particularly good addition-subtraction chains for these fixed scalars. Our approach is inspired by the ideas used in the ECM implementation by Dixon and Lenstra [12] from 1992. In [12] the total cost to compute the ECSM, in terms of point doubling and point additions, is lowered by testing if the computation of the ECSM using batches of small prime products is cheaper (requires fewer point additions) than processing the primes one at a time (or all in one big batch). We generalize this idea: many billions of integers, which are constructed such that they can be computed using an addition-subtraction chain with a high doubling/addition ratio, are tested for smoothness and factored. By fixing different popular elliptic curve scalar values used in ECM inside NFS we are able to combine some of these integers using a greedy approach. This results in a *more efficient* ECSM algorithm with a *smaller* memory footprint. To illustrate, compared to the cofactorization setting considered by Bernstein et al. in [5,4] (using the parameter $B_1 = 2^{13}$) the techniques from this paper reduce the memory by a factor 55. This makes our approach particularly interesting for environments where the memory (per thread) is constrained; e.g. GPUs. We illustrate the practical benefits by implementing this approach for GPUs: setting a new throughput speed record compared to the current CPU, GPU and FPGA based results reported in literature. The best addition-subtraction chains found for the various popular B_1 values can be found online [7].

This paper is organized as follows. After recalling the preliminaries in Section 2 the notation and basic idea behind elliptic curve constant scalar multiplication is discussed in Section 3. Section 4 explains how to combine these chains such that they might result in a faster and more memory efficient ECM. Section 5 explains a side-effect why certain chains require more modular multiplications and Section 6 presents the obtained results. Section 7 concludes the paper.

2   Preliminaries

2.1   The Elliptic Curve Method

The elliptic curve method (ECM) for integer factorization [24] is analogous to the Pollard $p-1$ integer factorization method [33] and attempts to factor a composite integer n. The general idea behind ECM is as follows (we follow the description from [24]). First, pick a random point P and construct an elliptic curve E over $\mathbf{Z}/n\mathbf{Z}$ such that $P \in E(\mathbf{Z}/n\mathbf{Z})$ (cf. [22, Sec. 2.B]). Next, compute the elliptic curve scalar multiplication $Q = kP \in E(\mathbf{Z}/n\mathbf{Z})$. The positive integer k is selected such that it is divisible by many small prime powers: e.g. $k = \mathrm{lcm}(1, 2, \ldots, B_1)$ for some bound $B_1 \in \mathbf{Z}$. If for a prime p dividing n the order $\#E(\mathbf{F}_p)$ is B_1-powersmooth (an integer is defined to be B-powersmooth if none of the prime powers dividing this integer is greater than B) then $\#E(\mathbf{F}_p) \mid k$. In other words, $Q = kP$ and the neutral element of the curve become the same modulo p. In this event we have $p \mid \gcd(n, Q_z)$, where Q_z is the z-coordinate of the point Q when using projective Weierstrass coordinates. If $\gcd(n, Q_z) \neq n$ then we have split n.

Hasse proved (see e.g. [36, Theorem 1.1]) that the order $\#E(\mathbf{F}_p)$ is in the interval $[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$. The advantage of ECM is that one can randomize the group order by trying different curves. It has been shown in [24] that the (heuristic) run-time of ECM depends mainly on p, the smallest non-trivial prime divisor of n, and can be expressed as

$$O(\exp((\sqrt{2} + o(1))(\sqrt{\log p \log \log p}))M(\log n))$$

where $M(\log n)$ represents the complexity of multiplication modulo n and the $o(1)$ is for $p \to \infty$. The approach described here is often referred to as "stage 1". There is a "stage 2" continuation for ECM which takes as input a bound $B_2 \in \mathbf{Z}$ and succeeds (in factoring n) if $Q = kP$ has prime order ℓ (for $B_1 < \ell < B_2$) in $E(\mathbf{F}_p)$. This means that $\#E(\mathbf{F}_p)$ is B_1-powersmooth except for one prime factor which is below B_2. There are several techniques [10,26,27] how to perform stage 2 efficiently. In the following we will focus on stage 1 only.

2.2   Cofactorization Using ECM

The relation collection phase, one of the two main phases of NFS, generates a lot of composite integers which need to be tested for powersmoothness. This is done using different factorization techniques and is denoted as the cofactorization phase. To illustrate, the total time spent in the cofactorization procedure was roughly one third of the sieving time when factoring the 768-bit RSA modulus in [20]. Note that this one third includes the time of pseudo primality tests and different factorization methods: quadratic sieve [34], Pollard $p - 1$ [33] and ECM. In this cofactorization phase only composites up to 140 bits were considered and ECM was used only for composites up to 109 bits. The parameters for ECM varied depending on the size of the composites and ranged from $B_1 = 150$ to $B_1 = 500$ where often only a single curve was tried with a maximum of around

Table 1. Performance comparison between GMP-ECM and EECM-MPFQ using the "$a = -1$" twisted Edwards curves in terms of modular multiplications (**M**) and squarings (**S**) together with the required number of residues modulo n (R) which needs to be kept in memory.

	GMP-ECM [41]				EECM-MPFQ [3]			
B1	#**S**	#**M**	#**S**+#**M**	#R	#**S**	#**M**	#**S**+#**M**	#R
256	1 066	2 025	3 091	14	1 436	1 638	3 074	38
512	2 200	4 210	6 410	14	2 952	3 183	6 135	62
1024	4 422	8 494	12 916	14	5 892	6 144	12 036	134
8192	35 508	68 920	104 428	14	47 156	45 884	93 040	550

eight curves. Observing the trend of past record factorizations, it is conceivable that cofactorization becomes more important in bigger factorizations (cf. [5] for more detailed arguments about the significance of ECM in NFS).

2.3   Montgomery versus Edwards Curves

The main motivation to use Edwards (over Montgomery) curves is performance. There is one implementation of ECM using Edwards curves available: EECM-MPFQ. This implementation includes the "$a = 1$" Edwards curves approach from [3] and the "$a = -1$" Edwards curves approach from [1]. The $a = -1$ Edwards ECM approach is the fastest in practice and we use this as the base setting to compare to. Table 1 compares the required number of multiplications and squarings required in GMP-ECM and EECM-MPFQ for different typical B_1 values used in ECM when used as a cofactorization method in NFS. These numbers show that using Edwards curves results in fewer modular multiplications and squarings. However, the required storage for GMP-ECM (Montgomery curves) is independent of B_1 while it grows almost linearly with the size of B_1 and is significantly higher, due to the use of windowing based methods, for EECM-MPFQ (Edwards curves, see [3, Table 4.1]).

3   Elliptic Curve Constant Scalar Multiplication

Most of the addition-subtraction chains based algorithms in practice use a w-bit windowing technique, for some (optimal) width w, to reduce the number of required elliptic curve additions. The total number of additions may be significantly reduced by using this approach but one also needs to store more points: 2^{w-1} when using sliding windows [38]. In environments where the available memory per thread is low, these methods cannot be used or one is forced to settle for a suboptimal window size. A prime example of such a platform are graphics processing units (GPUs); one of the latest GPU architectures [29] (Fermi) shares 64 kilobyte fast shared memory per 32 processors and each processor typically time-shares multiple threads (e.g., 16 to 32 corresponding to 128 to 64 bytes per thread).

We investigate two approaches to lower the number of elliptic curve additions *and* the storage required to compute the scalar product. Our approach is inspired by the results reported by Dixon and Lenstra [12]. Suppose we have a scalar $k = \text{lcm}(1, \ldots, B_1) = \prod_{i=1}^{\ell} p_i$, where the p_i are primes which can occur multiple times. Typically, the ECSM is implemented processing one such p_i at a time [39]. In [12] it is suggested to process the p_i in *batches*; i.e. multiply a batch of p_i's at a time such that the weight of the product $w(\prod_i p_i)$, the number of ones in the binary representation of $\prod_i p_i$, is (much) lower than the sum of the individual weights $\sum_i w(p_i)$. If this is the case then the number of required EC-additions is reduced when using the straight forward double-and-add approach (which does not require to store any additional precomputed points). Such low-weight products can be constructed by greedily searching through b-tuples of the p_i where b is small. In [12] b was at most 3 which reduced the total weight by approximately a factor three. As an example the following triple is given

$$1028107 \cdot 1030639 \cdot 1097101 = 1162496086223388673$$
$$w(1028107) = 10, \quad w(1030639) = 16, \quad w(1097101) = 11,$$
$$w(1162496086223388673) = 8,$$

where the product of primes of weights 10, 16, and 11 results in a integer of weight eight. The resulting composite integer can be computed using an addition chain requiring only seven additions and 60 doublings using the naive double-and-add algorithm.

In this section we explore different methods to find numbers which can be constructed using even better (higher) doubling/addition ratios. These methods do not aim to construct sequences by combining the different p_i (as in [12]) but we propose an opposite approach by factoring many integers which are the result of addition-subtraction chains with high doubling/addition ratios and subsequently combining these integers such that all p_i's are used. These addition-subtraction chains are constructed such that they do not require any large lookup tables. Notice that the information encoding the sequence of arithmetic operations has to be stored (in all approaches). This does not pose a problem since this information is constant and can be shared among all the computational units (or streamed to the units or even hardcoded) and hence does not result in additional overhead in practice.

In the remainder of the paper we denote addition-subtraction chains simply as chains.

3.1   Chains with Restrictions

In order to generate integers which can be computed using a chain with a high doubling/addition ratio we need to construct and denote chains of a certain length m. A chain is a sequence of doublings, additions and subtractions denoted by D, A and S respectively. A doubling can always be assumed to apply to the previously generated element in the chain (instead of doubling any previous element), since one can reorder the symbols such that doubling always occurs

on the last element without changing the result of the chain. In some cases this might result in a shorter (more efficient) sequence when the same element is doubled multiple times. Let us define the set of symbols $\mathscr{O}$ as

$$\mathscr{O} = \{D\} \cup \{A_{i,j} \mid i,j \in \mathbf{Z}, i > j\} \cup \{S_{i,j} \mid i,j \in \mathbf{Z}, i > j\},$$

where the subscripts indicate on which element in the chain we compute (this is made more precise later). The set of all m-tuples, ordered lists of m elements, of symbols in $\mathscr{O}$ with the restriction that no elements can be used which have not yet been generated is

$$\mathcal{O}_m = \{(o_{m-1}, \ldots, o_0) \in \mathscr{O}^m \mid o_k \in \{D\} \cup \{A_{i,j} \mid i \le k\} \cup \{S_{i,j} \mid i \le k\}, 0 \le k < m\}.$$

In order to construct a chain from such an m-tuple of symbols we define functions $\sigma_m : \mathscr{O} \times \mathbf{Z}^{m+1} \to \mathbf{Z}^{m+2}$ such that $(o, (t_m, \ldots, t_0)) \mapsto (t_{m+1}, t_m, \ldots, t_0)$ where

$$t_{m+1} = \begin{cases} 2t_m & \text{if } o = D, \\ t_i + t_j & \text{if } o = A_{i,j}, \\ t_i - t_j & \text{if } o = S_{i,j}. \end{cases}$$

Given an m-tuple of symbols $(o_{m-1}, \ldots, o_0) \in \mathcal{O}_m$ the $(m+1)$-tuple of integers associated to this chain is $\sigma_{m-1}(o_{m-1}, \sigma_{m-2}(o_{m-2}, \ldots, \sigma_0(o_0, 1)\ldots))$ and the *resulting integer* produced by this chain is t_m. As an example consider the 7-tuple of symbols $(S_{6,0}, D, D, A_{3,0}, D, D, D) \in \mathcal{O}_7$ which corresponds to the 8-tuple of integers in the chain $(35, 36, 18, 9, 8, 4, 2, 1)$ computed as

$$\sigma_6(S_{6,0}, \sigma_5(D, \sigma_4(D, \sigma_3(A_{3,0}, \sigma_2(D, \sigma_1(D, \sigma_0(D, 1))))))).$$

The function σ_m is the correspondence between a tuple of symbols and the actual chain. The example shows how to compute the resulting integer 35 using one subtraction, one addition and five doublings.

The set of tuples $\mathcal{O}_m$ consists of the most generic type of chains, a significant amount of tuples corresponds to chains which perform useless (unnecessary) computations. An example is computing the addition (or subtraction) of two previous values without using this result. To address this we define a more restricted set of tuples $\mathcal{P}_m \subset \mathcal{O}_m$ as

$$\mathcal{P}_m = \{(o_{m-1}, \ldots, o_0) \in \mathcal{O}_m \mid o_k \in \{D\} \cup \{A_{i,j} \mid i = k\} \cup \{S_{i,j} \mid i = k\}, 0 \le k < m\}.$$

These additional restrictions ensure that, just as for the doubling, we only add or subtract to the last integer in the sequence to obtain the next one. Such chains are known as *Brauer chains* or *star addition chains* [18, Section C6].

In this setting we write A_j and S_j for $A_{i,j}$ and $S_{i,j}$, respectively, and $k > 0$ subsequent instances of D are denoted by D^k. The previous example can now be written as $S_0 D^2 A_0 D^3 \in \mathcal{P}_7$ by abusing the notation: omitting the brackets and comma's. In practice we would generate sequences of symbols such that a number of elliptic curve additions $\mathbf{A}$ and doublings $\mathbf{D}$ are fixed and look at sequences of symbols of length $m = \mathbf{A} + \mathbf{D}$ which use $\mathbf{A}$ times A_j or S_j and

D times D. Different tuples might compute the same integer result. Using our example, the number 35 can be obtained with $\mathbf{D} = 5$ and $\mathbf{A} = 2$ in different ways

$$35 = (2^3 + 1) \cdot 2^2 - 1 \quad S_0 D^2 A_0 D^3 \in \mathcal{P}_7$$
$$= (2^4 + 1) \cdot 2 + 1 \quad A_0 D A_0 D^4 \in \mathcal{P}_7.$$

3.2 Generating Chains

We discuss how to efficiently generate the resulting integers t_m in a low-storage and no-storage setting.

The Low-Storage Setting. Let $\mathbf{A}$ be the number of elliptic curve additions and $\mathbf{D}$ the number of elliptic curve doublings (with $\mathbf{D} \geq \mathbf{A}$). The generation of *all* the tuples in $\mathcal{P}_m$, with $m = \mathbf{A} + \mathbf{D}$ results in many identical integers t_m. Removing these duplicate integers can be achieved by first generating and storing all the resulting integers and subsequently sorting and keeping exactly one of consecutive equal integers. To avoid storing all the resulting integers for a given pair $(\mathbf{A}, \mathbf{D})$, which requires a significant amount of storage as we will see later, and to avoid sorting this huge data set we define a more restricted set of rules $\mathcal{Q}_m \subset \mathcal{P}_m \subset \mathcal{O}_m$ as follows

$$\mathcal{Q}_m = \big\{ (o_{m-1}, \ldots, o_0) \in \mathcal{P}_m \mid o_0 = D, o_{m-1} \in \{A_i, S_i\}, \text{and for } 0 < k < m-1:$$
$$o_k \in \{D\} \cup \{A_i, S_i\}, o_k \in \{A_i, S_i\} \Rightarrow o_{k-1} = D$$
$$\wedge \ (i = 0 \ \vee \ o_{i-1} \in \{A_\ell, S_\ell\}) \big\}.$$

The restrictions used in the definition of $\mathcal{Q}_m$ ensure that the resulting integer is odd and only addition (or subtraction) of an odd number to the current (even) number is allowed. This approach significantly reduces the amount of chains which produce the same resulting integer at the cost of slightly reducing the number of unique integers produced. To illustrate, for $\mathbf{D} = 50$ the total number of tuples generated by $\mathcal{P}_{53}$ is more than 140 times higher compared to $\mathcal{Q}_{53}$ while the number of unique odd resulting integers is only 1.09 times higher.

The list of $m + 1$ integers u_i corresponding to the m-tuple of symbols from $\mathcal{Q}_m$ can be efficiently generated recursively using

$$u_{i+1} = \begin{cases} 2u_i \\ u_i \pm u_j \text{ for } j < i \text{ and } 2 \mid u_i, 2 \nmid u_j \end{cases}$$

with $u_0 = 1$ and ensuring that the final operation is not a doubling (to make the resulting integer odd). Hence, the next integer in the sequence can always be obtained by doubling or adding a previous odd number u_j to the current even integer u_i. The required storage depends on which u_j are used in subsequent additions and at which indices they are used. In practice we generate all sequences using a fixed number of doublings $\mathbf{D}$ and additions $\mathbf{A}$ making sure that the resulting storage requirement is never too large.

A sequence of additions and doublings corresponding to the chains resulting from $\mathcal{Q}_m$ looks like

$$A_{i_{\mathbf{A}-1}} D^{d_{\mathbf{A}-1}} \ldots A_{i_1} D^{d_1} A_{i_0} D^{d_0} = (A_{i_{\mathbf{A}-1}} D) D^{d_{\mathbf{A}-1}-1} \ldots (A_{i_1} D) D^{d_1-1} (A_{i_0} D) D^{d_0-1}$$

$$(1)$$

with $\mathbf{D} = \sum_{i=0}^{\mathbf{A}-1} d_i$, $d_i > 0$, and indices i_j that satisfy the restrictions of $\mathcal{Q}_m$, i.e., i_j takes one of the values $\sum_{g=0}^{h}(d_g + 1)$ for $-1 \le h < j$. Such a sequence starts with a doubling, ends with an addition and an addition is always preceded by a doubling. Hence, there are $\binom{\mathbf{D}-1}{\mathbf{A}-1}$ choices for the order of the $\mathbf{A} - 1$ pairs $(A_{i_j} D)$ and the $\mathbf{D} - \mathbf{A}$ doublings D. Since every addition can be substituted by a subtraction the number of possibilities is multiplied by a factor $2^{\mathbf{A}}$. The indices i_j can be chosen in $\mathbf{A}!$ ways, hence the total number of resulting integers produced by $\mathcal{Q}_m$ is

$$\binom{\mathbf{D}-1}{\mathbf{A}-1} \cdot \mathbf{A}! \cdot 2^{\mathbf{A}} = 2^{\mathbf{A}} \cdot \mathbf{A} \cdot \prod_{i=1}^{\mathbf{A}-1}(\mathbf{D} - \mathbf{A} + i).$$

The No-Storage Setting. The second setting we consider is constructing chains which do not require any additional stored points, besides the in- and output (and possibly some auxiliary variables required to calculate the elliptic curve group operation). This means we are looking for integers which can be computed using chains which only use doublings and add or subtract the input point. We can define the set of tuples $\mathcal{R}_m \subset \mathcal{Q}_m$ as $\mathcal{R}_m = \{(o_{m-1}, \ldots, o_0) \in \mathcal{Q}_m \mid o_k \in \{A_0, S_0, D\}, 0 \le k < m\}$. All resulting integers of no-storage chains which can be constructed using $\mathbf{A}$ elliptic curve additions and $\mathbf{D}$ elliptic curve doublings are of the form

$$2^{\mathbf{D}} + \sum_{i=0}^{\mathbf{A}-1} \pm 2^{n_i}, \quad \text{with } 0 = n_0 < n_1 < \ldots < n_i < \ldots < n_{\mathbf{A}-1} < \mathbf{D}.$$

This follows from (1) by setting $i_j = 0$; we have $n_i = \sum_{g=1}^{i} d_{\mathbf{A}-g}$. Using the same argument as in the low-storage setting the number of resulting integers generated by $\mathcal{R}_m$ is $\binom{\mathbf{D}-1}{\mathbf{A}-1} \cdot 2^{\mathbf{A}}$. Compared to the low-storage setting the number is reduced by a factor of $\mathbf{A}!$, reflecting the missing choice of the indices i_j.

4   Combining Chains

Recall that, given a bound B_1, we want to perform an elliptic curve scalar multiplication with the integer $k = \prod_{i=1}^{\ell} p_i = \mathrm{lcm}(1, \ldots, B_1)$ where the product ranges over ℓ (not necessarily distinct) primes. We can get rid of the problems posed by the primes 2 in this product by noticing that they can be handled by a sequence of doublings at the end of the ECSM and assuming in the following that all s_i are odd. The techniques from the previous section provide us with a lot of integers which can be constructed using a known number of additions (here we count subtractions as additions) and doublings. Since different chains can lead to the same integer we pick for each of these integers one chain (preferably the one with the lowest cost). In this way we get a list of distinct integers, each with an associated chain. We index this list by an index set I and call s_i the integer corresponding to $i \in I$. For $i \in I$ denote by $\mathrm{add}(s_i)$ resp. $\mathrm{dbl}(s_i)$ the number

of additions resp. doublings in the chain and by $\{s_{i,1}, \ldots, s_{i,t_i}\}$ the multiset of the primes in the prime decomposition of s_i. Furthermore, let $\mathrm{cost}(s_i)$ be the cost of performing a scalar multiplication with s_i using the associated chain. A reasonable choice for Edwards curves is $\mathrm{cost}(s_i) = 7\mathrm{dbl}(s_i) + 8\mathrm{add}(s_i) + 1$ which will be discussed in the next section.

Ideally, we want to find a subset $I' \subset I$ such that $k \mid \prod_{i \in I'} s_i$ and $\sum_{i \in I'} \mathrm{cost}(s_i)$ is minimal. To facilitate our task we will modify this in two ways. If the product in the first condition is bigger than k we do more work than necessary. This can lead to a lower cost, but we assume that replacing the first condition by $k = \prod_{i \in I'} s_i$ will not increase the minimum of $\sum_{i \in I'} \mathrm{cost}(s_i)$ significantly. The second modification is the replacement of $\sum_{i \in I'} \mathrm{cost}(s_i)$ by $\sum_{i \in I'} \mathrm{add}(s_i)$. To explain why we think that this does not increase the minimum too much we consider subsets I' for which $\sum_{i \in I'} \mathrm{cost}(s_i)$ is close to the minimum. Then most s_i have a high ratio $\frac{\mathrm{dbl}(s_i)}{\mathrm{add}(s_i)}$ and therefore we have for most of them $s_i \approx 2^{\mathrm{dbl}(s_i)}$. Since $\prod_{i \in I'} s_i = k$ the sum $\sum_{i \in I'} \mathrm{dbl}(s_i) \approx \log_2(k)$ does not vary too much. Furthermore, the summand 1 in the cost function is the least significant term and the cardinality of I' does not vary much. We are aware that the second modification is more delicate than the first one, but, as explained below, we will generate many sets I' and will pick the best one amongst them using the more costly function $\mathrm{cost}(s_i)$.

The condition $k = \prod_{i \in I'} s_i$ implies that every s_i in this product is B_1-powersmooth which suggests the following two stage approach:

1. Restrict to $\hat{I} = \{i \in I \mid s_i \text{ is } B_1\text{-powersmooth}\}$.
2. Find a subset $I' \subset \hat{I}$ such that the multisets $\bigcup_{i \in I'} \{s_{i,1}, \ldots, s_{i,t_i}\} = \{p_1, \ldots, p_\ell\}$ coincide and that $\sum_{i \in I'} \mathrm{add}(s_i)$ is minimal.

Testing a large list of numbers for B_1-powersmoothness can be done using the method from [15, Section 4]. The main idea is to build a product tree from the list, replace the root node R (the product of all numbers of the list) by $k \bmod R$ (where $k = \mathrm{lcm}(1, \ldots, B_1)$ is precomputed) and then tree-wise replace each node by the residue of k modulo the node. The leaves resulting in 0 contained B_1-powersmooth numbers and their factorizations can be obtained by other means.

Finding an optimal set I' is in general a difficult problem and has been studied in [31]. We choose to use a greedy approach which produces satisfactory results. We start with an empty set I' and the multiset $M = \{p_1, \ldots, p_\ell\}$ of primes to be matched. As long as M is non-empty we select an integer $s_i = \prod_{j=1}^{t_i} s_{i,j}$ with $\{s_{i,1}, \ldots s_{i,t_i}\} \subset M$ such that the ratio $\frac{\mathrm{dbl}(s_i)}{\mathrm{add}(s_i)}$ is high and replace I' by $I' \cup \{i\}$ and M by $M \setminus \{s_{i,1}, \ldots s_{i,t_i}\}$. This may fail because we might not be able to satisfy the condition $\{s_{i,1}, \ldots s_{i,t_i}\} \subset M$ at a given point. There are several ways to overcome this problem, e.g., we could increase our supply of s_i by generating more chains. Another way consists in aborting the greedy search at this point, getting $k = c \cdot \prod_{i \in I'} s_i$ for some integer c. Using the method of Dixon/Lenstra, we can search for a decomposition of c into several factors, each having a good chain. For the sizes of B_1 considered in this paper, namely $B_1 \leq 8192$, c consisted of

very few primes and was often 1. Therefore the usually lower doubling/addition ratio of the c-part does not pose a problem for small B_1.

A refinement to this approach is to also take the size of the prime factors $s_{i,j}$ into account. A strategy could be to prefer choosing integers s_i which have mostly large prime divisors, since the majority of the primes p_i is large. The idea is to attach a score to a B_1-powersmooth integer given its prime factorization with respect to the currently unmatched prime factors in k. For a multiset N of primes bounded by B_1 the ratio of j-bit primes is defined as

$$a_j(N) := \frac{\#\{p \in N \mid \lceil \log_2(p) \rceil = j\}}{\#M},$$

where $1 \leq j \leq \lceil \log_2(B_1) \rceil$. Given M, the multiset of currently unmatched primes, the *score* of s_i is defined as

$$\text{score} \left(s_i = \prod_{j=1}^{t_i} s_{i,j}, M \right) = \sum_{\substack{h=1: \\ a_h(M) \neq 0}}^{\lceil \log_2(B_1) \rceil} \frac{a_h(\{s_{i,1}, \ldots, s_{i,t_i}\})}{a_h(M)}$$

The higher the score the more small prime divisors are likely to be present. In general, for a given ratio, we select the integers which have a low score.

To illustrate, consider $B_1 = 1024$ where the initial a_i are

$$a_2 = 0.032, \ a_3 = 0.037, \ a_4 = 0.021, \ a_5 = 0.053, \ a_6 = 0.037,$$
$$a_7 = 0.069, \ a_8 = 0.122, \ a_9 = 0.229, \ a_{10} = 0.399$$

(with $\sum_{i=2}^{10} a_i = 1$). Almost 40 percent of all the primes fall in the largest (10-bit) category. An example of a low score-integer is $11529215054666795009 = 743 \cdot 719 \cdot 677 \cdot 461 \cdot 457 \cdot 449 \cdot 337$ where the size of the smallest prime is 9-bit, the score is 3.57 and this integer can be computed using 63 doublings and five additions as $A_0 D^{11} A_0 D^{12} A_0 D^{10} A_0 D^{28} A_0 D^2 \in \mathcal{R}_{68}$. On the other hand, an example of a high-score integer, consisting of mainly small primes, is $1048575 = 41 \cdot 31 \cdot 11 \cdot 5^2 \cdot 3$, its score is significant higher (29.62) and it can be computed with 20 doublings and a single subtraction as $S_0 D^{20} \in \mathcal{R}_{21}$.

This approach using scores is outlined in Algorithm 1. Note that the scores are recalculated each time an s_i is chosen. In practice one could reduce the amount of these costly recalculations by picking several s_i in lines 10-13 of the algorithm; in this case one has to check that the union of the prime factors of the chosen s_i is still a multisubset of M.

A Randomized Variant. In the current state, Algorithm 1 returns a single solution given a set of input parameters. To increase the amount of different subsets I', and thereby hopefully improving the results, we randomize the selection process of the index that is added in lines 10-13 of the algorithm. With probability $x \in \mathbf{R}$ $(0 < x < 1)$ select the s_i corresponding to score_1 or, with probability $1 - x$, skip it and repeat this procedure for score_2 and so on. If we have reached the end of the list (after j trials) one could apply a deterministic choice.

Algorithm 1. Given a bound B_1 and a set of B_1-powersmooth integers $\{s_i \mid i \in \hat{I}\}$, which can be computed with a chain using $\text{add}(s_i)$ resp. $\text{dbl}(s_i)$ elliptic curve additions resp. doublings, together with the prime factorization of these integers $(s_i = \prod_j s_{i,j})$ the algorithm attempts to output triples $(s_j, \text{add}(s_j), \text{dbl}(s_j))$ such that $\text{lcm}(1, \ldots, B_1) = c \cdot \prod_j s_j$ for a small integer c. This algorithm considers scores $\leq T$ only and combines integers s_i for which $\dfrac{\text{dbl}(s_i)}{\text{add}(s_i)} \geq r$ where r starts at r_h and is decreased until r_l.

Input: $\begin{cases} \text{Bound } B_1 \in \mathbf{Z}, \text{ we have } \text{lcm}(1, \ldots, B_1) = \prod_{i=1}^{\ell} p_i \text{ with } p_i \text{ prime.} \\ \text{Set of integers } \{s_i \mid i \in \hat{I}\} \text{ with } s_i = \prod_j s_{i,j} \text{ for } s_{i,j} \text{ prime and } i \in \hat{I}. \\ \text{Upper and lower bound on the doubling/addition ratio: } r_h \text{ and } r_l. \\ \text{A threshold value for the score: } T. \end{cases}$

Output: Triples $(s_i, \text{add}(s_i), \text{dbl}(s_i))$ and c such that $c \cdot \prod_i s_i = \text{lcm}(1, \ldots, B_1)$.

1. $M \leftarrow \{p_1, \ldots, p_\ell\}$, $I' \leftarrow \emptyset$
2. **for** $r = r_h$ to r_l **do**
3.   found $\leftarrow$ true
4.   **while** found=true **do**
5.   found $\leftarrow$ false, $j \leftarrow 0$
6.   **for** $i \in \hat{I}$ **do**
7.   **if** $\{s_{i,1}, \ldots, s_{i,t_i}\} \subset M$ and $\dfrac{\text{dbl}(s_i)}{\text{add}(s_i)} \geq r$ and $\text{score}(s_i, M) \leq T$ **then**
8. $j \leftarrow j + 1$, $\text{score}_j \leftarrow (\text{score}(s_i, M), i)$
9.   sort score_i for $1 \leq i \leq j$ with respect to $\text{score}(s_i, M)$
10.   **if** $j \geq 1$ **then**
11. $i \leftarrow$ index from score_1, output $(s_i, \text{add}(s_i), \text{dbl}(s_i))$
12. $I' \leftarrow I' \cup \{i\}$, $M \leftarrow M \setminus \{s_{i,1}, \ldots, s_{i,t_i}\}$
13.   found $\leftarrow$ true
14. output $\{(s_i, \text{add}(s_i), \text{dbl}(s_i)) \mid i \in I'\}$ and $c = \prod_{p \in M} p$

5 Additional Multiplications

The fastest arithmetic for Edwards curves is due to Hisil et al. [19]. They propose to use extended twisted Edwards coordinates, which are twisted Edwards coordinates plus an auxiliary coordinate. This allows faster addition but slower doubling. Using a mixing technique, by switching between extended twisted Edwards and regular twisted Edwards, the overall cost for scalar multiplication is reduced [19]. This is realized by performing the doublings using the cheaper regular twisted Edwards coordinates when a doubling is followed by a doubling. When an addition is required after a doubling one can use the doubling formula in the extended twisted Edwards coordinates (which does not need the auxiliary coordinate as input) at the cost of an extra multiplication to compute the auxiliary coordinate of the result. Next, the fast addition is performed in extended twisted Edwards coordinates; one multiplication (to compute the auxiliary coordinate of the output) can be saved, cancelling the extra multiplication used when doubling, since a doubling is always performed after an addition in

Table 2. The left table shows the number of integers (#int) generated with an addition-subtraction chain using **A** and **D** elliptic curve additions and doublings respectively. All these integers were tested for $2.9 \cdot 10^9$-powersmoothness and, if smooth, the prime divisors are stored. The **bold** ranges indicate that 2^{31} random integers per single **A**, **D** combination were tested for smoothness instead of the full range. The right table shows the number of unique B_1-powersmooth integers in the no-storage and low-storage setting for different values of B_1.

No-storage setting			Low-storage setting		
A	**D**	#int	**A**	**D**	#int
1	$5-200$	$3.920 \cdot 10^2$	1	$5-250$	$4.920 \cdot 10^2$
2	$10-200$	$7.946 \cdot 10^4$	2	$10-250$	$2.487 \cdot 10^5$
3	$15-200$	$1.050 \cdot 10^7$	3	$15-250$	$1.235 \cdot 10^8$
4	$20-200$	$1.035 \cdot 10^9$	4	$20-250$	$6.101 \cdot 10^{10}$
5	$25-200$	$8.114 \cdot 10^{10}$	5	$25-158$	$2.956 \cdot 10^{12}$
			5	**$159-220$**	**$1.331 \cdot 10^{11}$**
6	$30-150$	$9.150 \cdot 10^{11}$	**6**	**$60-176$**	**$2.513 \cdot 10^{11}$**
7	$35-66$	$9.900 \cdot 10^{10}$			
Total		$1.096 \cdot 10^{12}$			$3.403 \cdot 10^{12}$

B_1	No-Storage	Low-Storage
256	$2.423 \cdot 10^5$	$9.210 \cdot 10^6$
512	$1.470 \cdot 10^6$	$3.159 \cdot 10^7$
1 024	$5.691 \cdot 10^6$	$7.861 \cdot 10^7$
8 192	$9.352 \cdot 10^7$	$4.400 \cdot 10^8$
$2.9 \cdot 10^9$	$2.274 \cdot 10^{10}$	$3.997 \cdot 10^{10}$

ECSM-algorithms. This approach assumes that both inputs of the elliptic curve addition are in extended twisted Edwards coordinates. This is the case for double-and-add algorithms and (signed) windowing algorithms where the computation of the auxiliary coordinates of the lookup table are a minor overhead.

In both our settings, where we consider low- and no-storage, this does not hold. The computation of the large elliptic curve scalar product is done by processing batches of prime products (the s_i) at a time. All the additions or subtractions required in the chain to compute s_i require that the points are in extended twisted Edwards coordinates. When required, the odd intermediate results are stored in extended twisted Edwards coordinates at a cost of a single additional multiplication. The cost of computing a low-storage chain $(o_{m-1}, \ldots, o_0) \in \mathcal{Q}_m$ resulting in s_i is increased by $x(s_i)$ multiplications, where $x(s_i) = \#\{j \mid \exists h : o_h \in \{A_j, S_j\}, 0 \leq h < m\}$; i.e. the unique number of indices used in the additions and subtractions. Therefore we get for the cost function from the previous section $\text{cost}(s_i) = 7\text{dbl}(s_i) + 8\text{add}(s_i) + x(s_i)$. In the no-storage setting we always have $x(s_i) = 1$ leading to the choice for $\text{cost}(s_i)$ given at the beginning of the previous section. In total we have $\#\{\text{chains used}\}$ additional multiplications in the no-storage setting and a potentially higher number in the low-storage setting. We can save one multiplication due to the sequence containing the power of 2 (which consists of doublings only) and another multiplication if we assume that the input point is already in extended twisted Edwards coordinates.

6   Results

Using the rules given in Section 3.2 for both the no-storage and the low-storage setting, we generated more than 10^{12} integers for many choices of the number of additions **A** and doublings **D**. Table 2 summarizes the ranges we have covered where bold ranges (in the low-storage setting) indicate that only 2^{31} random integers were generated instead of the full range. All these integers were subjected

Table 3. The table shows the number of modular multiplications (**M**) and squarings (**S**) required to calculate **A** elliptic curve additions and **D** doublings for various B_1 parameters when factoring an integer n with ECM. The memory required is expressed as the number of residues (R), integers modulo n, which are kept in memory. The performance speedup **PS** (in terms of #**M**+#**S**) and memory reduction **MR** compared to the ECM approach from [1] using "$a = -1$" twisted Edwards curves is given.

B_1		#**M**	#**S**	#**M** + #**S**	**PS**	**A**	**D**	#R	**MR**
256	[1]	1 638	1 436	3 074		69	359	38	
No-storage		1 400	1 444	2 844	1.08	38	361	10	3.80
Low-storage		1 383	1 448	2 831	1.09	35	362	14	2.71
512	[1]	3 183	2 952	6 135		120	738	62	
No-storage		2 842	2 964	5 806	1.06	75	741	10	6.20
Low-storage		2 776	2 964	5 740	1.07	65	741	18	3.44
1 024	[1]	6 144	5 892	12 036		215	1 473	134	
No-storage		5 596	5 912	11 508	1.05	141	1 478	10	13.40
Low-storage		5 471	5 904	11 375	1.06	123	1 476	18	7.44
8 192	[1]	45 884	47 156	93 040		1 314	11 789	550	
No-storage		43 914	47 160	91 074	1.02	1 043	11 790	10	55.00
Low-storage		42 855	47 136	89 991	1.03	878	11 784	18	30.56

to $2.9 \cdot 10^9$-powersmoothness tests which reduced the number of integers by about two orders of magnitude. This large powersmoothness-bound was chosen to facilitate searching for efficient chains for much larger B_1 parameters. From the reduced set of integers we extracted those that are B_1-powersmooth for the values of B_1 used in this paper (see right part in Table 2). These computations were done on five 8-core Intel Xeon E5430 (2.66GHz) and took more than a year, i.e., in total over 40 core years. The smoothness testing required most of the run-time and up to 4.6GB of memory. Using the approach outlined in Algorithm 1 one of these nodes was occasionally used for the combining experiments which consisted of thousands of runs of the randomized greedy approach, each of them taking only a couple of seconds for these low values of B_1.

Table 3 shows the results obtained using Algorithm 1 on our dataset (see Table 2). The memory required is expressed in the number of residues (R), integers modulo n, which need to be kept in memory. Here we assume that extended twisted Edwards coordinates are used, i.e., every point is represented by four coordinates. In the setting of EECM-MPFQ [3,1] we assume that an optimal window size is used and that besides the window table only the input point needs to be kept in memory while we assume that two points (the input point and the current active point) are required in the no- and low-storage setting. The implementation of the elliptic curve group operation is assumed to require at most two auxiliary variables (residues). Hence, the no-storage setting requires memory for $2 \times 4 + 2 = 10$ residues modulo n. The low-storage results presented in Table 3 require to store at most two additional points (8 more residues modulo n compared to the no-storage setting). This is still significantly less compared to the approach used in [3,1].

6.1   Application to GPUs

When running ECM on memory constrained devices, like GPUs, the large number of precomputed points required for the windowing methods cannot be stored in fast memory. Typically one is forced to settle for a (much) smaller window size reducing the advantage from using twisted Edwards curves. For example, in [5] no large window sizes are used at all, the authors remark: "Besides the base point, we cannot cache any other points". Memory is also a problem in [4], the faster curve arithmetic from Hisil et al. [19] is not used since this requires storing a fourth coordinate per point.

From the data given in Table 3 it becomes clear that our approach reduces the memory requirements significantly. For example, the memory required to run ECM in the cofactorization setting on GPUs using $B_1 = 8\,192$ can be reduced by a factor 55. This setting was already considered in [5,4] where the authors were forced to reduce memory requirements by using suboptimal window sizes. Hence, when using the methods described in this paper *less* memory is required allowing the usage of the *faster* curve arithmetic and *reducing* the number of elliptic curve additions required in the computation of the elliptic curve scalar multiplication.

6.2   Performance Comparison

In order to measure the practical speedup of the methods described in this paper we implemented the no-storage approach on GPUs. This implementation uses the Compute Unified Device Architecture (CUDA) which facilitates the development of massively-parallel general purpose applications for GPUs [30]. Our implementation is targeted at the third generation CUDA GPUs called "Fermi" [29]. Table 4 compares the performance results of different hardware platforms for $B_1 = 960$ and $B_1 = 8192$, numbers chosen such that we can directly compare to results reported in the literature on other (hardware) platforms. For $B_1 = 960$, which is used as the example B_1 value in [40,11] and not spending as much effort as for $B_1 = 1024$, we were able to construct a no-storage chain requiring 1 371 doublings and 135 additions. The FPGA and GTX295 results are quadratically scaled to 192-bit arithmetic to compare the different performance results. The other GPU results are from [4] and this implementation is optimized for the second generation CUDA GPUs. The pricing for this card is omitted since it is no longer sold (this card was launched January 2009). The results on the Intel i7-2600K CPUs have been obtained with the ECM implementation (using Montgomery curves) from the NFS software suite [14] which is responsible for all recent record NFS factorizations (e.g. [20]) and the EECM-MPFQ software package [2] which uses Edwards curves. The FPGA results are from [11,40] and the FPGA prices are taken from [40]. Note that the prices are for the GPU, CPU or FPGA devices only; in order to get a fully operational system more hardware is required. Note also that for all of the considered devices newer versions with better price performance ratio exist, but we do not expect that these will change this comparison significantly.

Table 4. Performance comparison of ECM on different platforms (using the "$a = -1$" twisted Edwards curves if available). The first table lists the different hardware properties. The second and third table state results for $B_1 = 960$ and $B_1 = 8192$ respectively. The scaled number of curves are when using 192-bit moduli. The performance ratio is the ratio between the GTX 580 no-storage row and the current row for the scaled number of curves per 100 USD.

properties	GPU		CPU	FPGA	
	GTX 295	GTX 580	Intel i7-2600K	V4SX35-10	V4SX25-10
#cores	480	512	4	24	1
clock (MHz)	1 242	1 544	3 400	200	220
price (USD)	-	400	300	468	298
#threads	46 080	8 192	4	24	1
#bits in moduli	210	192	192	202	135

performance (#curves), $B_1 = 960$				performance
	(1/sec)	(1/sec, scaled)	(1/100 USD, scaled)	ratio
GTX 580, no-storage	171 486	171 486	42 872	1.00
GTX 580, windowing	79 170	79 170	19 793	2.17
Intel i7 [14]	13 661	13 661	4 554	9.41
Intel i7 [2]	8 677	8 677	2 892	14.82
V4SX35-10 [40]	3 240	3 586	766	55.97
V4SX25-10 [11]	16 000	7 910	2 654	16.15

performance (#curves), $B_1 = 8192$				
GTX 295 [4]	4 928	5 895	-	-
GTX 580, no-storage	19 869	19 869	4 967	1.00
GTX 580, windowing	9 106	9 106	2 277	2.18
Intel i7 [14]	1 629	1 629	543	9.15
Intel i7 [2]	1 092	1 092	364	13.65

For the sake of comparison we also implemented Edwards ECM for GPUs using the same 192-bit arithmetic but using the windowing based approach. For $B_1 = 960$ ($B_1 = 8192$) we used a signed sliding window of size 2^6 (2^8), precomputing and storing 2^5 (2^7) extended twisted Edwards coordinates. These results are stated in Table 4 as well. On the GTX 580 the no-storage approach is more than twice as fast as the approach based on windowing techniques. This is significantly better than the theoretical numbers from Table 3. When running exactly the same experiment on 96-bit (three 32-bit limbs instead of six 32-bit limbs) moduli the number of curves per second for the no-storage and windowing approach is 76 665 and 75 584 for $B_1 = 8 192$ and 649 904 and 618 111 for $B_1 = 960$, respectively. We think that this behaviour can be partially explained by an increased memory usage for the windowing approach and a better handling of the no-storage approach by the compiler since this approach uses fewer variables.

Another interesting observation is that the FPGA performance per 100 USD is lower than that of the CPU-based approaches. Furthermore, aided by the no-storage approach outlined in this paper, the GPU performance is almost an order of magnitude faster per 100 USD than the CPU and more than a order

of magnitude faster compared to the fastest FPGA results. This suggests that GPUs are the best platform, i.e. give the best performance / price ratio, for integer cofactorization.

7   Conclusion

The relatively new Edwards curves combined with the fast arithmetic from extended twisted Edwards coordinates are faster compared to using Montgomery curves. This speed-up comes at a price, namely a larger memory requirement which, when optimizing for speed, grows roughly linearly in the size of B_1, whereas the memory requirement in the Montgomery curves setting is constant and small. Inspired by the approach from Dixon and Lenstra and using the fact that only a few popular B_1-values are used in practice in NFS, we have presented techniques to reduce the memory requirement significantly by doing precomputations for these B_1-values. In these precomputations we tested over 10^{12} integers coming from chains with a low addition/doubling ratio for smoothness and combined them using a greedy approach. Our results show that we require significantly less memory compared to the current state-of-the-art Edwards ECM approach, and are even slightly faster. This makes our approach extremely suitable for memory-constrained parallel architectures like GPUs. This is demonstrated by our GPU implementation which sets a new ECM cofactorization throughput speed record.

References

1. Bernstein, D.J., Birkner, P., Lange, T.: Starfish on Strike. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 61–80. Springer, Heidelberg (2010)
2. Bernstein, D.J., Birkner, P., Lange, T., Peters, C.: EECM: ECM using Edwards curves (2010), Software, http://eecm.cr.yp.to/
3. Bernstein, D.J., Birkner, P., Lange, T., Peters, C.: ECM using Edwards curves. Mathematics of Computation (to appear, 2012)
4. Bernstein, D.J., Chen, H.-C., Chen, M.-S., Cheng, C.-M., Hsiao, C.-H., Lange, T., Lin, Z.-C., Yang, B.-Y.: The billion-mulmod-per-second PC. In: Special-purpose Hardware for Attacking Cryptographic Systems, SHARCS 2009, pp. 131–144 (2009)
5. Bernstein, D.J., Chen, T.-R., Cheng, C.-M., Lange, T., Yang, B.-Y.: ECM on Graphics Cards. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 483–501. Springer, Heidelberg (2009)

6. Bernstein, D.J., Lange, T.: Analysis and Optimization of Elliptic-Curve Single-Scalar Multiplication. In: Mullen, G.L., Panario, D., Shparlinski, I.E. (eds.) Finite Fields and Applications. Contemporary Mathematics Series, vol. 461, pp. 1–19. American Mathematical Society (2008)

7. Bos, J.W., Kleinjung, T.: ECM at work, project page (2012), http://research.microsoft.com/ecmatwork/

8. Bos, J.W., Kleinjung, T., Lenstra, A.K., Montgomery, P.L.: Efficient SIMD arithmetic modulo a Mersenne number. In: IEEE Symposium on Computer Arithmetic, ARITH-20, pp. 213–221. IEEE Computer Society (2011)

9. Brauer, A.: On addition chains. Bulletin of the American Mathematical Society 45, 736–739 (1939)

10. Brent, R.P.: Some integer factorization algorithms using elliptic curves. Australian Computer Science Communications 8, 149–163 (1986)

11. de Meulenaer, G., Gosset, F., de Dormale, G.M., Quisquater, J.-J.: Integer factorization based on elliptic curve method: Towards better exploitation of reconfigurable hardware. In: Field-Programmable Custom Computing Machines, FCCM 2007, pp. 197–206. IEEE Computer Society (2007)

12. Dixon, B., Lenstra, A.K.: Massively Parallel Elliptic Curve Factoring. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 183–193. Springer, Heidelberg (1993)

13. Edwards, H.M.: A normal form for elliptic curves. Bulletin of the American Mathematical Society 44, 393–422 (2007)

14. Franke, J., Kleinjung, T.: GNFS for linux. Software (2012)

15. Franke, J., Kleinjung, T., Morain, F., Wirth, T.: Proving the Primality of Very Large Numbers with fastECPP. In: Buell, D.A. (ed.) ANTS 2004. LNCS, vol. 3076, pp. 194–207. Springer, Heidelberg (2004)

16. 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: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 119–133. Springer, Heidelberg (2006)

17. Güneysu, T., Kasper, T., Novotny, M., Paar, C., Rupp, A.: Cryptanalysis with COPACOBANA. IEEE Transactions on Computers 57, 1498–1513 (2008)

18. Guy, R.: Unsolved problems in number theory, 3rd edn., vol. 1. Springer (2004)

19. Hisil, H., Wong, K.K.-H., Carter, G., Dawson, E.: Twisted Edwards Curves Revisited. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 326–343. Springer, Heidelberg (2008)

20. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H., Timofeev, A., Zimmermann, P.: Factorization of a 768-Bit RSA Modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer, Heidelberg (2010)

21. Kruppa, A.: A software implementation of ECM for NFS. Research Report RR-7041, INRIA (2009), http://hal.inria.fr/inria-00419094/PDF/RR-7041.pdf

22. Lenstra, A.K., Lenstra Jr., H.W.: Algorithms in number theory. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science (vol. A: Algorithms and Complexity), pp. 673–715. Elsevier and MIT Press (1990)

23. Lenstra, A.K., Lenstra Jr., H.W.: The Development of the Number Field Sieve. Lecture Notes in Mathematics, vol. 1554. Springer (1993)

24. Lenstra Jr., H.W.: Factoring integers with elliptic curves. Annals of Mathematics 126(3), 649–673 (1987)

25. Loebenberger, D., Putzka, J.: Optimization Strategies for Hardware-Based Cofactorization. In: Jacobson Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 170–181. Springer, Heidelberg (2009)
26. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987)
27. Montgomery, P.L.: An FFT extension of the elliptic curve method of factorization. PhD thesis, University of California (1992)
28. Morain, F., Olivos, J.: Speeding up the computations on an elliptic curve using addition-subtraction chains. Informatique Théorique et Applications/Theoretical Informatics and Applications 24, 531–544 (1990)
29. NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi (2009)
30. NVIDIA. NVIDIA CUDA Programming Guide 3.2 (2010)
31. Pisinger, D.: A minimal algorithm for the multiple-choice knapsack problem. European Journal of Operational Research 83(2), 394–410 (1995)
32. Pollard, J.M.: The lattice sieve. In: [23], pp. 43–49
33. Pollard, J.M.: Theorems on factorization and primality testing. Proceedings of the Cambridge Philosophical Society 76, 521–528 (1974)
34. Pomerance, C.: The Quadratic Sieve Factoring Algorithm. In: Beth, T., Cot, N., Ingemarsson, I. (eds.) EUROCRYPT 1984. LNCS, vol. 209, pp. 169–182. Springer, Heidelberg (1985)
35. Scholz, A.: Aufgabe 253. Jahresbericht der deutschen Mathematiker-Vereingung 47, 41–42 (1937)
36. Silverman, J.H.: *The Arithmetic of Elliptic Curves*. Gradute Texts in Mathematics, vol. 106. Springer (1986)
37. Šimka, M., Pelzl, J., Kleinjung, T., Franke, J., Priplata, C., Stahlke, C., Drutarovský, M., Fischer, V.: Hardware factorization based on elliptic curve method. In: Field-Programmable Custom Computing Machines, FCCM 2005, pp. 107–116. IEEE Computer Society (2005)
38. Thurber, E.G.: On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. Duke Mathematical Journal 40, 907–913 (1973)
39. Zimmermann, P., Dodson, B.: 20 Years of ECM. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS 2006. LNCS, vol. 4076, pp. 525–542. Springer, Heidelberg (2006)
40. Zimmermann, R., Güneysu, T., Paar, C.: High-performance integer factoring with reconfigurable devices. In: Field Programmable Logic and Applications, FPL 2010, pp. 83–88. IEEE (2010)
41. Zimmermann, P., et al.: GMP-ECM (elliptic curve method for integer factorization) (2012), Software, https://gforge.inria.fr/projects/ecm/