Revisiting ECM on GPUs

Jonas Wloka[1(✉)], Jan Richter-Brockmann[2], Colin Stahlke[3], Thorsten Kleinjung[4], Christine Priplata[3], and Tim Güneysu[1,2]

[1] DFKI GmbH, Cyber-Physical Systems, Bremen, Germany
jonas.wloka@dfki.de
[2] Ruhr University Bochum, Horst Görtz Institute Bochum, Bochum, Germany
{jan.richter-brockmann,tim.gueneysu}@rub.de
[3] CONET Solutions GmbH, Hennef, Germany
{cstahlke,cpriplata}@conet.de
[4] EPFL IC LACAL, Station 14, Lausanne, Switzerland
thorsten.kleinjung@epfl.ch

Abstract. Modern public-key cryptography is a crucial part of our contemporary life where a secure communication channel with another party is needed. With the advance of more powerful computing architectures – especially Graphics Processing Units (GPUs) – traditional approaches like RSA and Diffie-Hellman schemes are more and more in danger of being broken.

We present a highly optimized implementation of Lenstra's ECM algorithm customized for GPUs. Our implementation uses state-of-the-art elliptic curve arithmetic and optimized integer arithmetic while providing the possibility of arbitrarily scaling ECM's parameters allowing an application even for larger discrete logarithm problems. Furthermore, the proposed software is not limited to any specific GPU generation and is to the best of our knowledge the first implementation supporting multiple device computation. To this end, for a bound of $B_1 = 8192$ and a modulus size of 192 bit, we achieve a throughput of 214 thousand ECM trials per second on a modern RTX 2080 Ti GPU considering only the first stage of ECM. To solve the Discrete Logarithm Problem for larger bit sizes, our software can easily support larger parameter sets such that a throughput of 2 781 ECM trials per second is achieved using $B_1 = 50\,000$, $B_2 = 5\,000\,000$, and a modulus size of 448 bit.

Keywords: ECM · Cryptanalysis · Prime factorization · GPU

1 Introduction

Public-Key Cryptography (PKC) is a neccessary part of any large-scale cryptographic infrastructure in which communicating partners are unable to exchange keys over a secure channel.

PKC systems use a keypair of public and private key, designed such that to retrieve the secret counterpart of a public key, a potential attacker would have to solve a mathematically hard problem. Traditionally – most prominently in RSA and Diffie-Hellman schemes – *factorization of integers* or computing a *discrete logarithm* are the hard problems at the core of the scheme. For reasonable key sizes, both these problems are considered to be computationally infeasible for classical computers.

If built, large-scale quantum computers, are able to compute both factorization and discrete logarithms in polynomial time. However, common problem sizes are not only under threat by quantum computers: With Moore's Law still mostly accurate, classical computing power becomes more readily available at a cheaper price. Additionally, in the last decade more diverse computing architectures are available. Graphics Processing Units (GPUs) have been used in multiple scientific applications – including cryptanalysis – for their massive amount of parallel computing cores. As the problem of factorization and computing a discrete logarithm can (in part) be parallelized, GPU architectures fit these tasks well.

Nowadays the NIST recommends to use 2048- and 3072-bit keys for RSA [3]. Factoring keys of this size is out of reach for current publicly known algorithms on classical computers. However, in [35], the authors found that still tens of thousands of 512-bit keys are used in the wild, which could be factored for around $70 within only a few hours.

To find the prime factorization of large numbers, the currently best performing algorithm is the General Number Field Sieve (GNFS). During a run of the GNFS algorithm, many numbers smaller than the main target need to be factored which is commonly done by Lenstra's Elliptic-Curve Factorization Method (ECM) and is inherently parallel.

Related Work. ECM has been implemented on graphic cards before and several approaches at optimizing the routines used in ECM on the different levels have already been published.

A general overview of factoring, solving the Discrete Logarithm Problem (DLP) and the role of ECM in such efforts is given in [26,27]. The most recent result in factorization and solving a DLP was announced in December 2019 with the factorization of RSA-240 and with solving the DLP for a 795-bit prime [13]. Previous records for the factorization of a 768-bit RSA modulus and the computation of a 768-bit DLP are reported in [21,22] and [23], respectively. A general overview of factorization methods, and the use of ECM on GPUs within the GNFS is given in [30].

The original publication of ECM by Lenstra in [28] has since received much attention, especially the choice of curves [5,7] and parameters [17] was found to have a major impact on the algorithm's performance. Optimizing curve arithmetic for parallel architectures – mostly GPUs – has been a topic of many scientific works as well [1,2,19,25,29]. A detailed description of a parallel implementation for GPUs is given in [11].

To this end, the implementation of ECM on GPUs has attracted a lot of attention in the years around 2010, as General Purpose Computing on GPU (GPGPU) became readily available to the scientific community. The beginning of the usage of GPUs for cryptanalytic purposes is marked by [34], including elliptic curve cryptography, an early implementation of ECM on graphics cards was published in [4,8]. A discussion of the performance of ECM on available GPUs around 2010 is given in [12,32]. With the application of ECM in the cofactorization step of the GNFS, the discussion of an implementation for GPUs that includes ECM's second stage on the GPU was published in [31].

Existing Implementations. Although many authors have already worked on implementing ECM on GPUs, only a few implementations are openly available. `GMP-ECM`[1], which features support for computing the first stage of ECM on graphic cards using Montgomery curves, was introduced by Zimmermann et al.

Bernstein et al. published `GPU-ECM` in [8] and an improved version `CUDA-ECM` in [4]. In the following years, Bernstein et al. [5] released `GMP-EECM` – a variant of `GMP-ECM` using Edwards curves –, and subsequently `EECM-MPFQ`, which is available online at https://eecm.cr.yp.to/. Both latter, however, do not support GPUs.

To the authors' knowledge, the most recent implementation, including ECM's second stage by Miele et al. in [31] has not been made publicly available. Additionally, almost all previous implementations of ECM on GPUs only consider a fixed set of parameters for the bit length and ECM bounds. As we will show in Section 2, these restrictions do not meet real world assumptions and scalability seems to be significant even for larger moduli.

Contribution. We propose a complete and scalable implementation of ECM suitable for NVIDIA GPUs, that includes:

1. **State-of-the-art Fast Curve Arithmetic** All elliptic curve computations are performed using $a = -1$ Twisted Edwards curves with the fastest known arithmetical operations.
2. **Scalability to Arbitrary ECM Parameters** We show that currently used parameters for ECM in related work do not meet assumptions in realistic scenarios as most implementations are optimized for a set of small and fixed problem sizes. Hence, we propose an implementation which can be easily scaled to any arbitrary ECM parameter and bit length.
3. **State-of-the-art Integer Arithmetic** We demonstrate that the commonly used CIOS implementation strategy can be outperformed by the less widespread FIOS and FIPS approaches on modern GPUs.
4. **No Limitation to any Specific GPU Generation** Our implementation uses GPU-generation independent low level code based upon the PTX-level abstraction.

The corresponding software is released under an open source license and is available at https://github.com/Chair-for-Security-Engineering/ecmongpu.

Outline. The remainder of this paper is organized as follows: Sect. 2 briefly summarizes the DLP and the background of ECM. In Sect. 3 we describe our

[1] Available at http://ecm.gforge.inria.fr/.

optimizations for stage one and stage two on the algorithmic level. This is followed by Sect. 4 introducing our implementation strategies for GPUs. Before concluding this work in Sect. 6, we evaluate and compare our implementation in terms of throughput in Sect. 5.

2 Preliminaries

This section provides the mathematical background of ECM and introduces cofactorization as part of GNFS.

2.1 Elliptic Curve Method

ECM introduced by H. W. Lenstra in [28] is a general purpose factoring algorithm which works on random elliptic curves defined modulo the composite number to be factored. Thus, ECM operates in the group of points on the curve. Whether ECM is able to find a factor of the composite depends on the smoothness of the order of the curve. Choosing a different random curve most likely results in a different group order. To increase the probability of finding a factor, ECM can be run multiple times (in parallel) on the same composite number. ECM consists of a first stage and an optional second stage.

Stage 1. In the first stage, one chooses a random curve E over $\mathbb{Z}_n$ with n the composite to factor, with p being one of its factors, and a random point P on the curve. With s a large scalar, one computes the scalar multiplication sP and hopes that $sP = \mathcal{O}$ (the identity element) on the curve modulo p, but not modulo n. As p is unknown, all computations are done on the curve defined over $\mathbb{Z}_n$.

This can be regarded as working on all curves defined over $\mathbb{Z}_p$ for all factors p simultaneously. If p was known, reducing the coordinates of a point computed over $\mathbb{Z}_n$ modulo p yields the point on the curve over $\mathbb{Z}_p$.

If s is a multiple of the group order, i.e., the number of points on the curve over $\mathbb{F}_p$, sP is equal to the point at infinity $\mathcal{O} = (0 : 1 : 0)$ modulo p, e.g., on Weierstrass curves, but not n. Thus, the x- and z-coordinates are a multiple of p, and so $\gcd(x, n)$ (or $\gcd(z, n)$) should reveal a factor of n.

One chooses s to be the product of all small powers of prime numbers p_i up to a specific bound B_1, i.e., $s = \mathrm{lcm}(1, 2, 3, \ldots, B_1)$. If the number of points on the chosen curve $\#E$ modulo p divides s, a factor will be found by ECM. This is equivalent to stating that the factorization of $\#E$ consists only of primes $\leq B_1$, thus is B_1-smooth.

Stage 2. Stage two of ECM relaxes the constraint that the group order on E has to be a product of primes smaller than B_1 and allows one additional prime factor larger than B_1 but smaller than a second bound B_2.

Thus, for $Q = sP$ the result from stage one, for each prime p_i with $B_1 < p_1 < p_2 < \cdots \leq B_2$, one computes $p_iQ = p_isP$ and hopes that p_is is a multiple of the group order. If so, p_iQ – as in stage one – is equivalent to the identity element modulo p, the x- and z-coordinates equal 0 modulo p but not modulo n, hence the gcd of the coordinates and n reveals a factor of n.

Curve Selection and Construction. As ECM's compute intensive part is essentially scalar multiplication, we chose *a = -1* Twisted Edwards curves [6] with extended projective coordinates [19] as these offer the lowest operation costs for point additions and doublings. Each point $P = (X : Y : T : Z)$ is thus represented by four coordinates, each of the size of the number to factor.

As ECM works on arbitrarily selected elliptic curves modulo the number to factor, multiple parameterized curve constructions have been proposed (see [37] for an overview). Our implementation constructs random curves according to the proposal by Gélin et al. in [17].

2.2 Discrete Logarithm Problem

In 2016 the DLP was solved for a 768-bit prime p [23]. The computation of a database containing discrete logarithms for small prime ideals took about 4000 CPU core years. Using this database, an individual discrete logarithm modulo p could be computed within about two core days. Using more than one CPU core, the latency could be decreased, but the parallelization is not trivial. Recently, Boudot et al. announced a new record, solving the DLP for a 795-bit prime [13].

The computation of an individual logarithm of a number z consists of two computationally intensive steps: the initial split and the descent. During the initial split the main task is to find two smooth integers that are norms of certain principal ideals, such that their quotient modulo p equals z. The prime factors of these two integers correspond to prime ideals with not too large norms. During the descent step, each of these prime ideals can be replaced by smaller ideals using relations found by sieving realized in the same way as during the sieving step in the first step of the GNFS. Eventually, all prime ideals are so small, that their discrete logarithms can all be found in the database. These discrete logarithms can easily be assembled to the discrete logarithm of the number z.

The initial split is done by first performing some sieving in some lattice. The dimension of this lattice can be two or eight for example, depending on the number fields. This produces a lot of pairs of integers. There are many lattices that can be used for sieving, which offer obvious opportunities for parallelization and lead to even more pairs of integers. It is enough to find just one pair such that both integers are smooth enough. Smoothness of integers can be tested by a combination of several factorization algorithms. The most popular are trial division, Pollard-$(p-1)$ and ECM.

One goal of our work was to reduce the latency of two CPU core days for the computation of individual 786-bit discrete logarithms using 25 CPUs with 4 cores each (Intel Core i7-4790K CPU @ 4.00GHz). In the initial split it is important to find good parameters for the factorization algorithms. For our purpose we found that

$$B_1 \approx 7 \cdot \exp(n/9)$$
$$B_2 \approx 600 \cdot \exp(0{,}113 \cdot n)$$

are good choices for ECM to detect an n-bit factor ($n \in \{44, 45, \ldots, 80\}$) using our CPUs. This is close to the widely used $B_2 \approx 100 \cdot B_1$. The sieving of the descent step was parallelized with Open MPI and the sieving strategy was carefully chosen and balanced with the factorization strategies used in the initial split. Finally, we managed to compute individual discrete logarithms on 25 CPUs (i.e., 100 cores) within three minutes.

The implementation of ECM on GPUs provides several opportunities to speed up the computation of discrete logarithms. First, it can be used for smoothness testing in the sieving step in the first step of the GNFS in order to reduce the 4000 core years by supporting the CPUs with GPUs. Second, in the same way it can speed up the sieving in the descent step. Third, it can be used for speeding up the smoothness tests in the initial split.

In our experiment we utilized two GeForce RTX 2080 TI GPUs filtering the pairs of integers in the initial split between the sieving and the smoothness tests. The parameters of the sieving in the initial split were relaxed, such that the sieving was faster, producing more pairs (and reducing their quality). This leaves us with a huge amount of pairs of integers, most of them not smooth enough. These integers were reduced to a size of 340 bit at most by trial division (or otherwise dropped). The surviving integer pairs were sent to the two GPUs in order to find factors with ECM using two curves, $B_1 = 5\,000$, and $B_2 = 20\,000$. A pair survived this step, if ECM found a factor in both integers and after division by this factor the integers were still composite. The remaining survivors were sent to the GPUs in order to find factors with ECM using 50 curves, $B_1 = 5000$, and $B_2 = 30000$. The final survivors were sent to a factorization engine on CPUs. Eventually, the use of GPUs reduced the latency of the computation of individual logarithms from three minutes to two minutes.

To this end, the aforementioned experiments demonstrate that our ECM implementation on GPUs can support the GNFS substantially, speeding up the computation of discrete logarithms and possibly also the factorization of RSA moduli with the GNFS.

After building a database for a prime p, individual discrete logarithms can be computed rather easily. We estimate the cost for building such a database within a year using CPUs to roughly 10^6 US$ for 768 bit, to 10^9 US$ for 1024 bit and to at least 10^{14} US$s for 1536 bit. In our experiments we could compute individual logarithms for 1024 bit within an hour on 100 CPU cores (up to the point of looking up in a database which we did not have). This is an upper bound since we did not focus on optimizations on polynomial selection and on choosing good parameters and a good balance between initial split and descent. The initial split produced 448-bit integers after trial division and the parameters for ECM went up to $B_1 = 50\,000$ and $B_2 = 5\,000\,000$. Due to the opportunity to scale our ECM implementation to any arbitrary parameter set, these numbers can be processed on GPUs which should considerably reduce the latency of one hour for 1024-bit individual logarithms.

3 Algorithmic Optimizations

With the general algorithm and background of ECM discussed in Sect. 2.1, this section introduces optimizations to both stages of the algorithm suitable for efficient GPU implementations.

3.1 Stage 1 Optimizations

As introduced in Sect. 2.1, during stage one of ECM a random point P on an elliptic curve is multiplied by a large scalar $s = \mathrm{lcm}(1, 2, \ldots, \mathrm{B}_1 - 1, \mathrm{B}_1)$, e.g., for a $B_1 = 50\,000$ s is 72115 bit. To this end, stage one of ECM is essentially a scalar multiplication of a point on an elliptic curve. This section will deal with the possible optimizations, leading to a faster computation of $s \cdot P$ for large s. This section introduces methods for reducing that effort.

Non-Adjacent Form. In general, our implementation uses a w-NAF (Nonadjacent form) representation for the scalar $s = \sum_{i=0}^{t-1} s_i 2^i$, where $s_i \in C = \{-2^{w-1}+1, -2^{w-1}+3, .., -1, 0, 1, ..., 2^{w-1}-1\}$. While the necessary point doublings roughly stay the same, the number of point additions is reduced at the cost of needing to precompute and store a small multiple of the base point for each positive coefficient from C. For example, choosing $w = 4$ reduces the number of point additions to 14 455 for a $B_1 = 50\,000$ at the cost of storage for three additional points $(3P, 5P, 7P)$.

The w-NAF representation of any scalar can be computed on-the-fly during program startup. For all upcoming experiments we decided to set $w = 4$ allowing a fair comparison and removing one degree of freedom.

Different Scalar Representations. For *fixed* values of B_1 used repeatedly, other representations of the scalar can be found with significantly more precomputation. Addition chains have been proposed by Bos et al. in [12], however finding (optimal) addition chains with low operation cost for large scalars is still an open question. In [15] Dixon et al. also proposed so-called batching for splitting the scalar s into batches of primes in order to lower the overall number of required point operations. In [9] Bernstein et al. introduced fast tripling formulas for points on Edwards curves and presented an algorithm finding the optimal chain for a target scalar s with the lowest amount of modular multiplications. Bouvier et al. also used tripling formulas and employed double-base chains and double-base expansions in combination with batches to generate multiple chains to compute scalars for somewhat larger bounds in [14]. Recently Yu et al. provided a more efficient algorithm to compute optimal double-base chains in [36].

However, these approaches are limited to small bounds B_1 (i.e., for $B_1 \leq 8\,192$) and therefore do not match our requirements of an arbitrary value for B_1. Nevertheless, we generated double-base chains for small values of B_1 with the algorithm from [9] choosing $S = \pm\{1, 3, 5, 7\}$ and benchmarked them.

The two approaches – batching and addition chains – can be combined by generating multiple addition chains, one for each batch [12,14]. We also included

Table 1. Comparison of different strategies optimizing the ECM throughput for stage one setting $B_1 = 8\,192$ and the modulus size to 192 bit. To count modular multiplications, we assume $1\mathbf{M} = 1\mathbf{S}$.

B_1	Optimal Chains [9]			4-NAF			Random Batching			Adapted from [14]		
	$\mathbf{M}^b$	$\mathbf{I}^c$	$\frac{\text{trials}}{\text{second}}$	$\mathbf{M}^b$	$\mathbf{I}^c$	$\frac{\text{trials}}{\text{second}}$	$\mathbf{M}^b$	$\mathbf{I}^c$	$\frac{\text{trials}}{\text{second}}$	$\mathbf{M}^b$	$\mathbf{I}^c$	$\frac{\text{Trials}}{\text{second}}$
4 096	48 442	4	311 032	49 777	4	354 422	48 307	20	294 466	N/A		
8 192	N/A[a]			99 328	4	215 495	95 756	64	163 751	90503	0	138565
50 000	N/A[a]			605 983	4	37 476	585 509	432	25 718	N/A		

[a] The calculation of an optimal chain is too computation-intensive [b] Modular multiplications
[c] Modular inversions (during computation of small multiples and/or point optimization)

results for a slightly modified version of the chains from [14]. We used their batching but generated *only* optimal double-base chains using the code from [9] with $S = \pm 1$ (no precomputation), whereas the authors use 22 double-base expansions and switch to Montgomery coordinates for 4 batches out of a total of 222 batches. As a result, our variant needs to perform 931 additional modular multiplications. We disabled our optimized point representation (see Sect. 4.2) due to the high number of chains resulting in many costly inversions.

While in general the best batching strategy for larger B_1 is unclear, we were able to generate multiple addition chains for a $B_1 = 50\,000$ by randomly selecting subsets of primes smaller than B_1 and using the algorithm from [9]. Keeping only the best generated chains, we continued generating new batching variants for the rest of the primes still to cover until the overall cost of the chains stabilized. This strategy will be called *Random Batching* in the following. We supply all generated batched double-base addition chain for our B_1 with the software.

Table 1 compares the ECM stage one's throughputs for $B_1 \in \{4\,096, 8\,192, 50\,000\}$ using the naive 4-NAF approach, our random batching, the results from [9] and our adaptation of [14] on an NVIDIA RTX 2080Ti. Although the batching based approaches require less modular operations (also compared to an optimal chain for $B_1 = 4\,096$), the absolute throughput is drastically lowered. We found that in practice the cost of using multiple chains quickly remedied the benefit of requiring less point operations: For each chain one needs to compute small multiples of the (new) base point when using a larger window size. Our implementation stores precomputed points in a variant of affine coordinates to reduce the cost of this point's addition, each requiring one inversion during precomputation (cf. Sect. 4.2). This approach is not well suited if precomputed points are only used for relatively few additions on a single chain.

In addition, for each digit in *double-base* chains the software has to check whether it has to perform a doubling or tripling. Even if using only the base point and disabling the optimization of its coordinates, the overhead introduced by the interruption of the GPU program flow between chains slows down the computation, even though the full set of batches are processed on the GPU with one kernel launch.

In our experiments, we found that using our optimized coordinates for point addition with w-NAF scalar representation is more beneficial to the overall throughput than using multiple addition chains without the optimization of precomputed points. Hence, our NAF approach achieves better results as only doublings are executed, the program flow is uninterrupted and no switching between operations is needed.

3.2 ECM Stage 2 Optimizations

As introduced above, in the second stage of ECM one hopes that the number of points on E is B_1-powersmooth, except for one additional prime factor. For stage two, a second bound B_2 is set, and each multiple of the result of stage one $p_{k+i}Q$ for each prime $B_1 < p_{k+1} < p_{k+2} < \cdots < p_{k+l} \leq B_2$ is computed.

Reducing Point Operations. The number of point operations can be reduced by employing a baby-step giant-step approach as in [30]. Each prime p_{k+i} is written as $p_{k+i} = vg \pm u$, with g a giant-step size and u the number of baby-steps. To cover all the primes between B_1 and B_2, set

$$\begin{aligned} u \in U &= \left\{ u \in \mathbb{Z} \;\middle|\; 1 \leq u \leq \frac{g}{2}, \gcd(u, g) = 1 \right\} \\ v \in V &= \left\{ v \in \mathbb{Z} \;\middle|\; \left\lceil \frac{B_1}{g} - \frac{1}{2} \right\rceil \leq v \leq \left\lfloor \frac{B_2}{g} + \frac{1}{2} \right\rfloor \right\}. \end{aligned}$$

As in stage two, one tries to find a prime $p_{k+i} = vg \pm u$ such that $(vg \pm u)Q = \mathcal{O}$ on the curve modulo a factor p. This is equivalent to finding a pair of vg and u, such that $vgQ = \pm uQ \mod p$. If this is the case, then the (affine) y-coordinates of vgQ and uQ are also equal and

$$\frac{y_{vgQ}}{z_{vgQ}} - \frac{y_{uQ}}{z_{uQ}} = 0 \mod p.$$

Since $\frac{y_P}{z_P} = \frac{y_{(-P)}}{z_{(-P)}}$ on Twisted Edwards curves, one only needs to check for $y_{vgQ}z_{uQ} - y_{uQ}z_{vgQ}$, if either $vg+u$ or $vg-u$ is a prime, thus saving computation on roughly half the prime numbers. Our implementation uses a bitfield to mark those combinations that are prime.

The result of the difference for all l primes of y-coordinates can be collected, so that stage two only outputs a single value m with

$$m = \prod_{v \in V} \prod_{u \in U} y_{vgQ}z_{uQ} - y_{uQ}z_{vgQ}.$$

If any of the differences $y_{vgQ}z_{uQ} - y_{uQ}z_{vgQ}$ equals zero modulo p, $\gcd(m, n)$ is divisible by p and usually not n thus a non-trivial factor of n is found.

When for all $u \in U$ points the point uQ is precomputed together with the giant-step stride of gQ, this approach only needs $|V| + |U| + 1$ point additions, plus $3|V||U|$ modular multiplications for the computation of m.

Reducing Multiplications. Our approach is to normalize all points vgQ and uQ to the *same* projective z-coordinates instead of affine coordinates. This way the computation of m only requires y-coordinates, because – as introduced above – the goal is to find equal points modulo p. Given $a \geq 2$ points $P_1, \ldots, P_a$ – in this case all giant-step points vgQ and baby-step points uQ – the following approach sets all z_{P_i} to $\prod_{1 \leq i \leq a} z_{P_i}$: To do so, each z_{P_i} needs to be multiplied by $\prod_{1 \leq i \leq a, i \neq k} z_{P_i}$. An efficient method to compute each $\prod_{1 \leq i \leq a, i \neq k} z_{P_i}$ is given in [24, p. 31].

Normalizing all points to the same z-coordinate costs $4(|V| + |U|)$ multiplications during precomputation and the cost of computing m drops down to $|V||U|$ modular multiplications, as $m = \prod_{v \in V} \prod_{u \in U} y_{vgQ} - y_{uQ}$.

However, for this normalization all baby- and giant-step points need to be precomputed which needs quite a lot of memory to store z- and y-coordinates of all $|V| + |U|$ baby-step and giant-step points, as well as the storage of the batch cross multiplication algorithm from [24, p. 31] with $|V||U|$ entries. If less memory is available, the giant-step points can be processed in batches. In this case, the normalization has to be computed again for each batch.

4 Implementation Strategies

The following sections discuss in more detail the implementation of multi-precision arithmetic and elliptic curve operations tuned to our requirements and those of GPUs.

4.1 Large Integer Representation on GPUs

Our implementation follows the straight-forward approach of, e.g., [29,31] and uses 32-bit integer limbs to store multi-precision values. The number of limbs for any number is set at compile time to the size of the largest number to factor. Thus, all operations iterating over limbs of multi-precision numbers, can be completely unrolled during compilation, yielding long sequences of straight-line machine code. To avoid inversions during multi-precision arithmetic, all computation on the GPU is carried out in the Montgomery domain. All multi-precision arithmetic routines use inline Parallel Thread Execution (PTX) assembly to make use of carry-flags and multiply-and-add instructions. Note that PTX code, while having an assembly-like syntax, is code for a virtual machine that is compiled to the actual architecture specific instructions set. PTX has the advantage of being hardware independent and ensures our proposed implementation is executable on a variety of NVIDIA hardware.

To enable fast parallel transfer of multi-precision values from global device memory to registers and shared memory of the GPU cores, multi-precision values in global memory are stored strided: Consecutive 32-bit integers in memory are arranged such that they are retrieved by different GPU threads, thus coalescing memory accesses to the same limb of different values by multiple threads into one memory transaction.

GPU-Optimized Montgomery Multiplication. As the modular multiplication is at the core of elliptic curve point operations, the speed of the implementation is most influenced by the speed of the modular multiplication routine. As in the implemented software architecture, a single thread performs the full multiplication to avoid any synchronization overhead between threads, reducing the amount of registers per multiplication is of high importance.

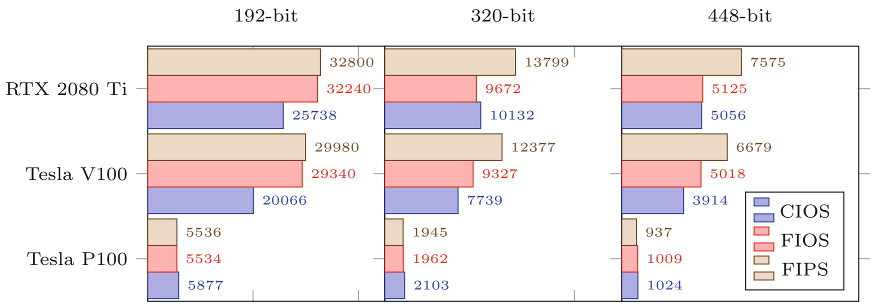

Fig. 1. Million modular multiplication per second for different Montgomery implementation strategies and architectures.

Different strategies to implement multi-precision Montgomery multiplication and reduction have been surveyed in [20]. These differ in two aspects: The tightness of interleaving of multiplication and reduction, and the access to operands' limbs. In [32], Neves et al. claimed that the Coarsely Integrated Operand Scanning (CIOS), Finely Integrated Operand Scanning (FIOS) and Finely Integrated Product Scanning (FIPS) strategies are the most promising, and CIOS is most widely used, e.g., in [34]. All three methods need $2l^2 + l$ multiplications of limbs for l-limb operands (see [20, Table 1] for a complete cost overview). Using PTX, each of these multiplications requires two instructions to retrieve the lower and upper half of the $2l$ product. PTX offers multiply-and-add instructions with carry-in and -out to almost entirely eliminate additional `add` instructions.

Our implementation of FIOS follows [18] in accumulating carries in an additional register to prevent excessive memory accesses and carry propagation loops. Our FIPS implementation follows [32, Algorithm 4].

Comparing FIPS, FIOS and CIOS on current GPUs, our benchmarks show varying results for newer architectures. Figure 1 shows the runtime of different strategies on different hardware architectures. For each of these benchmarks, 32 768 threads are started in 256 blocks, with 128 threads in each block. Each thread reads its input data from strided arrays in global memory and performs one million multiplications (reusing the result as operand for the next iteration) and writes the final product in strided form back to global memory.

For the most recent Volta and Turing architectures featuring integer arithmetic units, the FIPS strategy is the most efficient especially for larger moduli. On the older Pascal architecture, the difference between the implementation strategies' efficiency is much smaller. However, on the Tesla P100 CIOS slightly outperformed both finely integrated methods.

GPU-Optimized Montgomery Inversion. While modular inversions are costly compared to multiplications and are not used during any *hot* arithmetic, precomputed points are transformed needing one modular inversion per point. Montgomery Inversion, given a modulus n and a number $\tilde{A} = AR$ to invert in Montgomery representation, computes its inverse $\tilde{A}^{-1} = A^{-1}R \mod n$, again in Montgomery representation.

The algorithm implemented in this work is based on the Binary Extended Euclidean Algorithm as in [33, Algorithm 3]. Divisions by two within the algorithm are accomplished by using PTX *funnel shifts* to the right. The PTX instruction `shf.r.clamp` takes two 32-bit numbers, concatenates them and shifts the 64-bit value to the right, returning the lower 32 bit. Thus, each division by two can be achieved with l instructions for an l-word number. However, the inversion algorithm needs four branches depending on the number to invert and thus produces inner warp thread divergence.

4.2 Elliptic Curve Arithmetic on GPUs

Based on the modular arithmetic of the last section, the elliptic curve arithmetic can be implemented. With offering the lowest operation count (in terms of multiplications/squarings) of all proposed elliptic curves, our GPU implementation uses $a = -1$ twisted Edwards curves, with coordinates represented in extended projective format.

Point Arithmetic. The implementation of point addition and subtraction is a straight-forward application of the addition and doubling formulas from [19] using the multi-precision arithmetic detailed in the previous section.

Point Addition. Addition of an arbitrary point with $Z \neq 1$ is only needed seldom: During precomputation of small multiples of the base point for the w-NAF multiplication and during computation of the giant-steps for stage two. General point addition is implemented by a straight-forward application of the formulas from [10,19] as given in Algorithm 1.

Algorithm 1: Point addition on $a = -1$ twisted Edwards curves [10, 19].

```
Data: Points $P = (x_P, y_P, z_P, t_P)$ and $Q = (x_Q, y_Q, z_Q, t_Q)$ in extended projective
      coordinates, curve parameter $k = 2d$
Result: Point $R = P + Q = (x_R, y_R, z_R, t_R)$
1  $a \leftarrow (y_P - x_P) \cdot (y_Q - x_Q)$
2  $b \leftarrow (y_P + x_P) \cdot (y_Q + x_Q)$
3  $c \leftarrow t_P \cdot k \cdot t_Q$
4  $d \leftarrow z_P \cdot z_Q$
5  $d \leftarrow d + d$
6  $e \leftarrow b - a$
7  $f \leftarrow d - c$
8  $g \leftarrow d + c$
9  $h \leftarrow b + a$
10 $x_R \leftarrow e \cdot f$
11 $y_R \leftarrow g \cdot h$
12 $z_R \leftarrow f \cdot g$
13 $t_R \leftarrow e \cdot h$
14 return $(x_R, y_R, z_R, t_R)$
```

Table 2. Modular operation cost of the implemented point arithmetic.

	projective*			extended*		
	M	**S**	ADD	**M**	**S**	ADD
Doubling†	3	4	8	4	4	8
Tripling†	9	3	10	11	3	10
Addition*	8		9	9		9
Precomputed addition‡	6		7	7		7

* result coordinate format † operand in projective coordinates * operand in extended coordinates ‡ one operand in our modified coordinates

If one of the points of the addition is precomputed and used in many additions, further optimization is beneficial. As in the w-NAF point multiplication, precomputed points are only used for addition, all operations that solely depend on values of the point itself are done once during precomputation. These are addition and subtraction of x- and y-coordinates, as well as the multiplication of the t-coordinate with the curve constant $k = 2d$. To further save one multiplication per point addition, the precomputed point can be normalized such that its z-coordinate equals one at the cost of one inversion and three multiplications. Applying these optimizations yields the modified format of a precomputed point $\tilde{P}$ from the general point representation P, such that

$$x_{\tilde{P}} = y_P - x_P \qquad y_{\tilde{P}} = y_P + x_P \qquad z_{\tilde{P}} = 1 \qquad t_{\tilde{P}} = 2 \cdot d_{curve} \cdot t_P$$

Using this representation, point additions require seven multiplications only. Computing the inverse of a point $-P = (-x_P, y_P, z_P, -t_P)$ in its modified representation is achieved by switching the x- and y-coordinates, and computing $-t_{\tilde{P}} = n - t_{\tilde{P}} \mod n$, i.e., $-\tilde{P} = (y_{\tilde{P}}, x_{\tilde{P}}, 1, n - t_{\tilde{P}})$.

Point Doubling and Tripling. Point doubling is used for each digit of the scalar in scalar multiplication, tripling also on double-base chains. As all intermediate values do not fulfill the condition of $Z = 1$, no further optimized doubling formulas can be applied in this case. The implemented doubling and tripling routines follow [10,19] and [9].

Mixed Representation. Using extended projective coordinates, the point doubling formula does not use the t-coordinate of the input point. When using the w-NAF scalar multiplication, the number of non-zero digits is approximately $\frac{l}{w-1}$ for an l-bit scalar. Thus, there are long runs of zero bits in the w-NAF, resulting in many successive doublings without intermediate addition.

Thus, to further reduce multiplications during scalar multiplication computing the t-coordinate can be omitted if the scalar's next digit is zero, as no addition follows in this case. Furthermore, as each point addition is followed by a point doubling, which does not rely on the correct extended coordinate, again, the multiplication computing t_R can be omitted from all point additions within the scalar multiplication. The same applies to tripling. The resulting operation counts as implemented are listed in Table 2.

Scalar Multiplication. To compute the scalar multiple of any point P, as in the first stage of ECM, w-NAF multiplication is used. The first stage's scalar $s = \mathrm{lcm}(1, \ldots, B_1)$ is computed on the host and transformed into w-NAF representation, with w a configurable compile time constant defaulting to $w = 4$. Thus, each digit of $s_{w\text{-NAF}}$ is odd or zero and in the range of -2^{w-1} to 2^{w-1}.

Our precomputation generates $2P$ by point doubling and the small odd multiples of P, i.e., $\{3P, \ldots, (2^{w-1}-1)P\}$ with repeated addition of $2P$. Precomputed points are stored with strided coordinates along with other batch data in global memory, as registers and shared memory are not sufficiently available.

All threads read their corresponding precomputed point's coordinates from global memory to registers with coalesced memory accesses. In case the current digit of the NAF is negative, the precomputed point is inverted before addition. Again, as all threads are working on the same limb, this does not create any divergence.

5 Evaluation

Three different GPU platforms were available during this work, a Tesla P100 belonging to the Pascal family, a Tesla V100 manufactured in the Volta architecture, and a RTX 2080 Ti with a Turing architecture.

As the actual curves in use for ECM are not within the scope of this paper, the *yield*, i.e., the numbers for which a factor is found, is not part of this evaluation. Of interest is, however, the throughput of the implementation: *How many ECM trials can be performed per second on moduli of a given bit length.* Therefore, each benchmark in this work is conducted on 32 768 randomly generated numbers $n = pq$, with $\sqrt{n} \approx p \approx q$ and p and q prime.

Benchmarks for different problem sizes are carried out in two standard configurations, with the first being a somewhat standard throughout the literature to enable a comparison with previous works. As most previously reported GPU implementations only support the first stage of ECM on the GPU, this first case only executes stage one of the implementation with a bound of $B_1 = 8\,192$. The second benchmark parameter set is aimed at much larger ECM bounds and does include the second stage, with bounds $B_1 = 50\,000$ and $B_2 = 5\,000\,000$.

5.1 Stage One Bound

Firstly, we evaluate the impact of the bound B_1. Figure 2 gives the number of ECM trials per second for moduli of 192 bit and 320 bit for growing values of B_1. Note that the size of the scalar $s = \mathrm{lcm}(1, \ldots, B_1)$ grows very fast with B_1. Using w-NAF multiplication, the runtime of ECM mainly depends on the number of digits in s, resulting in the values seen in Fig. 2. Note that each single trial (per second) on the y-axis is equivalent to $\log_2 \operatorname{lcm}(1, \ldots, B_1)$ operations (double and possibly add) per second and thus changes for each value of B_1, e.g., 1 trial is equivalent to 14 447 ops for $B_1 = 10\,000$ and 28 821 ops for $B_1 = 20\,000$.

5.2 Stage Two Bound

For a given bound B_2, the number of primes less than or equal to B_2 are the key factor in determining the runtime of stage two. Via the prime number theorem, with a fixed negligible value for B_1, this value is approximately $\pi(B_2) \approx \frac{B_2}{\ln B_2}$. See Fig. 3 for the achieved ECM trials per second for different values of B_2. While for small values of B_2, the RTX 2080 Ti outperforms the Tesla V100, as soon as B_2 grows larger than 1 000 000, the Tesla V100 performs slightly better. As described in Sect. 3.2 for larger values of B_2 not all baby-step and giant-step points can fit into GPU memory, but have to be processed in batches. Our Tesla V100 setup features 16 GB of GPU memory while the RTX 2080 Ti only has 11 GB available. Again, note that the plot shows $\frac{\text{trials}}{\text{second}}$ where with growing B_2 the number of operations per trial increases with B_2.

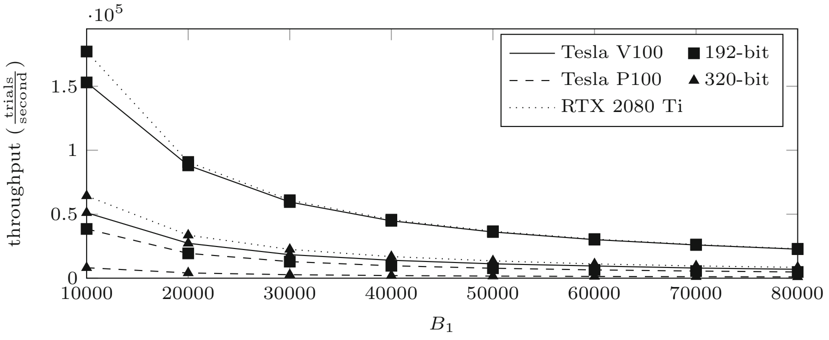

Fig. 2. ECM first stage trials per second for varying size of B_1.

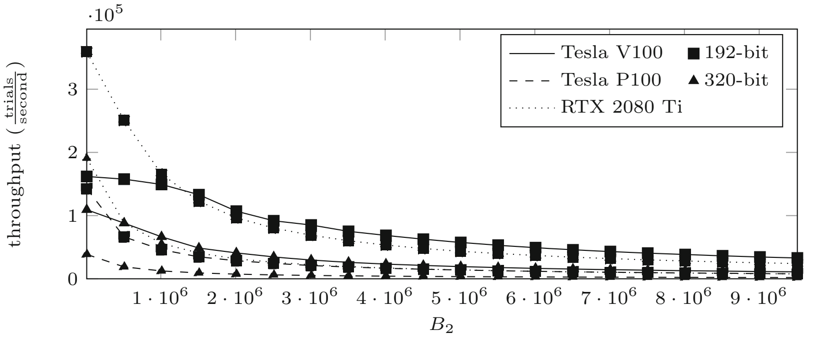

Fig. 3. ECM first and second stage trials per second for varying size of B_2, with $B_1 = 256$, and a stage two window size of $w = 2310$ (cf. Sect. 3.2).

5.3 ECM Throughput

With these benchmarks giving the runtime dependency on different parameters, this section gives absolute throughput numbers for the two exemplary cases of first stage only ECM with $B_1 = 8\,192$, and both stages with more ambitious $B_1 = 50\,000$ and $B_2 = 5\,000\,000$.

Stage 1. The absolute throughput for the first case for different moduli sizes is given in Table 3. Interestingly, when comparing the throughput for 192-bit moduli between the high-performance GPU Tesla V100 with the consumer GPU RTX 2080 Ti, the consumer card processes more ECM trials per second by a factor of 1.44.

Table 3. Absolute throughput of ECM trials for stage one (in thousands per second) on different platforms with $B_1 = 8\,192$ and varying moduli sizes.

	128	160	192	224	256	288	320	352	384	416	448
Tesla P100	103.9	66.6	46.8	33.5	19.0	14.3	9.9	8.3	7.0	6.0	5.2
Tesla V100	228.9	188.8	149.1	141.3	117.6	73.4	61.9	52.4	35.4	29.4	24.7
RTX 2080 Ti	450.6	310.0	214.1	152.5	124.2	98.9	77.1	58.8	37.2	29.7	24.7
2×RTX 2080 Ti	542.6	481.3	377.1	285.5	232.9	191.4	150.2	113.6	73.0	58.3	48.2

Table 4. Absolute throughput of ECM trials for stage one and stage two (in thousands per seconds) on different platforms with $B_1 = 50\,000$, $B_2 = 5\,000\,000$ and varying moduli sizes.

	128	160	192	224	256	288	320	352	384	416	448
Tesla P100	10.79	7.15	4.97	3.52	1.91	1.42	1.11	0.91	0.77	0.65	0.55
Tesla V100	46.88	30.74	22.85	17.12	13.58	7.99	7.12	5.78	4.60	3.49	2.78
RTX 2080 Ti	40.86	27.39	20.21	14.77	11.62	9.34	6.85	6.30	4.11	3.32	2.78
2×RTX 2080 Ti	80.46	53.79	39.42	28.61	22.51	17.91	13.50	9.72	7.89	6.46	5.39

Stage 1 and Stage 2 Eventually, Table 4 states the absolute throughput of the entire ECM setting the bounds to $B_1 = 50\,000$ and $B_2 = 5\,000\,000$. For the exemplary application with a modulus size of 448 bit mentioned in Sect. 2.2, only one RTX 2080 Ti is capable of processing 2 781 ECM trials per second.

Multiple Devices Our implementation is designed to use multiple GPUs to increase throughput. Table 3 and Table 4 show that the throughput is almost doubled when utilizing two RTX 2080 Ti, and more so for larger moduli and larger ECM parameters, as the ratio of host side to GPU computation shifts towards more work on the GPU.

Table 5. Comparison of scaled throughput for Montgomery multiplication from the literature and this work. Throughput values are given in $\frac{\text{multiplications}}{\text{core} \times \text{cycle}} \times 10^{-3}$.

GPU	[25]	[31]	[16][d]	this work		
	GTX 480	GTX 580	GTX 980 Ti[a]	Tesla P100[b]	Tesla V100[c]	RTX 2080 Ti[c]
Cores	480	512	2816	3584	5120	4352
Clock*	1401	1544	1000	1316	1530	1665
Modulus†						
128	3.54063	7.34319	4.03125	2.65388	9.01974	8.65915
160	2.85956	4.75631		1.74424	6.40737	6.01596
192	2.32423	3.32816		1.24673	4.65732	4.62363
224	1.90638	2.45785		0.91325	3.61875	3.46953
256	1.53313	1.88861	1.32813	0.70659	2.92659	2.80919
320	1.04687	1.21691		0.44531	1.97959	1.88013
384	0.75839	0.84880	0.64063	0.30303	1.41461	1.36107

* in MHz † in bits [a] two-pass approach [b] CIOS [c] FIPS
[d] Values have been scaled from throughput per *Streaming Multiprocessor* per clock cycle

5.4 Comparison to Previous Work

Multiple factors make it hard to compare our results to previous work: Especially the fast changing GPU architectures make a comparison very difficult, but also no comparable set of parameters for B_1 and B_2 has been established. In lack of a better computation power estimate, we adopt the approach of [31] to scale the results accomplished on different GPU hardware by scoring results per cuda cores $\times$ clock rate.

Montgomery Multiplication. Comparing the most influential building block, the Montgomery multiplication algorithm to previous publications is a first step. Table 5 lists relevant work, the hardware in use and a score for the throughput scaled by the number of Compute Unified Device Architecture (CUDA) cores and their clock rate. The implementation of this work is the fastest of all implementations under comparison on the RTX 2080 Ti and more so for larger moduli, however comes in last place for the Pascal architecture platforms. Using our implementation and a modern GPU manufactured in the Turing architecture, clearly outperforms the previous results.

ECM Throughput. Comparing the achieved throughput of the developed software with previously published results suffers from various problems: different hardware, varying modulus sizes and varying settings for both first and second stage bounds across different publications.

Especially, as to the authors' knowledge, apart from Miele et al. in [31], no other publication of ECM on GPUs implemented the second stage. Additionally, in [31] only very small bounds of $B_1 = 256$ and $B_2 = 16\,384$ were chosen. Note that the implemented w-NAF approach in stage one in this work benefits from larger B_1 as precomputation costs amortize.

Table 6. Comparison of this implementation with [12] and their parameter sets for 192-bit moduli. Values are given in $\frac{\text{ECM trials}}{\text{core}\times\text{cycle}} \times 10^{-5}$.

	Bos et al. [12]		this work		
	no-storage	windowing			
GPU	GTX 580		Tesla P100	Tesla V100	RTX 2080 Ti
cores/clock*	512/1544		3584/1316	5120/1530	4352/1665
$B_1 = 960$	2.1692	1.0014	0.64070	0.20936	0.49398
$B_1 = 8\,192$	0.2513	0.1151	0.09917	0.20134	0.29373
$B_1 = 50\,000$	N/A	N/A	.01650	0.04609	0.05168

* in MHz

For bounds this small our implementation is actually significantly slower, as host-side and precomputation overhead dominate the runtime.

Albeit already published in 2012, the comparison with [12] is the most interesting for the stage one implementation, as they also use a somewhat larger bound of $B_1 = 8\,192$, but do not implement stage two. However, the comparison lacks modulus sizes other than 192 bit, as [12] only published these results. The comparison to our implementation is shown in Table 6 and perfectly shows the advantage of our approach for larger bounds. Considering $B_1 = 8\,192$, our implementation slightly outperforms the *no-storage* approach by Bos et al. although we do not use highly optimized addition chains.

Even less recent, published in 2009, is the implementation by Bernstein et al. [8]. A comparison is somewhat unfair, as Bernstein developed *a = -1* Edwards curves after this paper was published. However, their GPU implementation uses the bound $B_1 = 8\,192$, and in comparison the proposed implementation is significantly faster. However, this comparison is unfair as multiple generations of hardware architectures aimed at GPGPU have been released within the last ten years, and the authors of [8] decided to use a floating point representation.

6 Conclusion

In this work we present a highly optimized and scalable implementation of the entire ECM algorithm for modern GPUs. On algorithmic level, we demonstrated that a w-NAF representation seems to be the most promising optimization technique realizing the scalar multiplication in the first stage. For the second stage we rely on an optimized baby-step giant-step approach. For the underlying Montgomery multiplication, we implemented three difference strategies where against our expectations FIPS performs best. Eventually, we demonstrate that the throughput of previous literature is achieved – and actually exceeded – on the most recent Turing architecture. We hope that the scalability, flexibility and free availability of our ECM implementation will support other researchers in achieving new factorization and DL records, reducing costs and reassessing the security of some algorithms used in PKC.

References

1. Antao, S., Bajard, J.C., Sousa, L.: RNS-based elliptic curve point multiplication for massive parallel architectures. Comput. J. **55**(5), 629–647 (2012)
2. Antao, S., Bajard, J.C., Sousa, L.: Elliptic curve point multiplication on GPUs. In: ASAP 2010–21st IEEE International Conference on Application-specific Systems, Architectures and Processors. IEEE, July 2010
3. Barker, E.B., Dang, Q.H.: Recommendation for Key Management Part 3: Application-Specific Key Management Guidance. Technical Report NIST SP 800–57Pt3r1, National Institute of Standards and Technology, January 2015
4. Bernstein, D.J., et al.: The billion-mulmod-per-second PC. In: SHARCS 2009 Workshop Record (Proceedings 4th Workshop on Special-purpose Hardware for Attacking Cryptograhic Systems, Lausanne, Switserland, September 9–10, 2009) (2009)
5. Bernstein, D., Birkner, P., Lange, T., Peters, C.: ECM using edwards curves. Math. Comput. **82**(282), 1139–1179 (2013)
6. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted edwards curves. In: Vaudenay, S. (ed.) AFRICACRYPT 2008. LNCS, vol. 5023, pp. 389–405. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68164-9_26
7. 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). https://doi.org/10.1007/978-3-642-14712-8_4
8. 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). https://doi.org/10.1007/978-3-642-01001-9_28
9. Bernstein, D.J., Chuengsatiansup, C., Lange, T.: Double-base scalar multiplication revisited. Cryptology ePrint Archive, Report 2017/037 (2017). https://eprint.iacr.org/2017/037
10. Bernstein, D.J., Lange, T.: Explicit-Formulas Database. https://hyperelliptic.org/EFD/index.html
11. Bos, J.W.: Low-latency elliptic curve scalar multiplication. Int. J. Parallel Prog. **40**(5), 532–550 (2012)
12. Bos, J.W., Kleinjung, T.: ECM at work. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 467–484. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34961-4_29
13. Boudot, F., Gaudry, P., Guillevic, A., Heninger, N., Thomé, E., Zimmermann, P.: Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment. Cryptology ePrint Archive, Report 2020/697 (2020). https://eprint.iacr.org/2020/697
14. Bouvier, C., Imbert, L.: Faster cofactorization with ECM using mixed representations. Cryptology ePrint Archive, Report 2018/669 (2018). https://eprint.iacr.org/2018/669
15. 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). https://doi.org/10.1007/3-540-47555-9_16
16. Emmart, N., Luitjens, J., Weems, C., Woolley, C.: Optimizing Modular Multiplication for NVIDIA's Maxwell GPUs. In: 2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH), pp. 47–54. IEEE, Silicon Valley, CA, USA, July 2016

17. Gélin, A., Kleinjung, T., Lenstra, A.K.: Parametrizations for families of ecm-friendly curves. In: Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25–28, 2017, pp. 165–171 (2017)
18. Großschädl, J., Kamendje, G.-A.: Optimized RISC architecture for multiple-precision modular arithmetic. In: Hutter, D., Müller, G., Stephan, W., Ullmann, M. (eds.) Security in Pervasive Computing. LNCS, vol. 2802, pp. 253–270. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-39881-3_22
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). https://doi.org/10.1007/978-3-540-89255-7_20
20. Kaya Koc, C., Acar, T., Kaliski, B.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996)
21. Kleinjung, T., et al.: Factorization of a 768-Bit RSA modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_18
22. Kleinjung, T., et al.: A heterogeneous computing environment to solve the 768-bit RSA challenge. Cluster Comput. **15**(1), 53–68 (2012)
23. Kleinjung, T., Diem, C., Lenstra, A.K., Priplata, C., Stahlke, C.: Computation of a 768-Bit prime field discrete logarithm. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 185–201. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_7
24. Kruppa, A.: A Software Implementation of ECM for NFS. Research Report RR-7041, INRIA (2009). https://hal.inria.fr/inria-00419094
25. Leboeuf, K., Muscedere, R., Ahmadi, M.: A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography. In: 2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), pp. 2593–2596, May 2013
26. Lenstra, A.K.: Integer factoring. Des. Codes Crypt. **19**(2–3), 101–128 (2000). https://doi.org/10.1023/A:1008397921377
27. Lenstra, A.K.: General purpose integer factoring. Cryptology ePrint Archive, Report 2017/1087 (2017). https://eprint.iacr.org/2017/1087
28. Lenstra, H.W.: Factoring integers with elliptic curves. Ann. Math. **126**(3), 649–673 (1987). https://doi.org/10.2307/1971363
29. Mahé, E.M., Chauvet, J.M.: Fast GPGPU-based elliptic curve scalar multiplication. Cryptology ePrint Archive, Report 2014/198 (2014). https://eprint.iacr.org/2014/198
30. Miele, A.: On the analysis of public-key cryptologic algorithms (2015). https://infoscience.epfl.ch/record/207710
31. Miele, A., Bos, J.W., Kleinjung, T., Lenstra, A.K.: Cofactorization on graphics processing units. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 335–352. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44709-3_19
32. Neves, S., Araujo, F.: On the performance of GPU public-key cryptography. In: ASAP 2011–22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 133–140. September 2011
33. Savas, E., Koc, C.K.: Montgomery inversion. J. Cryptographic Eng. **8**(3), 201–210 (2018)
34. Szerwinski, R., Güneysu, T.: Exploiting the power of GPUs for asymmetric cryptography. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 79–99. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85053-3_6

35. Valenta, L., Cohney, S., Liao, A., Fried, J., Bodduluri, S., Heninger, N.: Factoring as a service. In: Grossklags, J., Preneel, B. (eds.) FC 2016. LNCS, vol. 9603, pp. 321–338. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54970-4_19
36. Yu, W., Musa, S.A., Li, B.: Double-base chains for scalar multiplications on elliptic curves. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12107, pp. 538–565. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45727-3_18
37. 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). https://doi.org/10.1007/11792086_37