Cofactorization on Graphics Processing Units

Andrea Miele[1], Joppe W. Bos[2]*, Thorsten Kleinjung[1], and Arjen K. Lenstra[1]

[1] LACAL, EPFL, Lausanne, Switzerland
[2] NXP Semiconductors

Abstract. We show how the cofactorization step, a compute-intensive part of the relation collection phase of the number field sieve (NFS), can be farmed out to a graphics processing unit. Our implementation on a GTX 580 GPU, which is integrated with a state-of-the-art NFS implementation, can serve as a *cryptanalytic co-processor* for several Intel i7-3770K quad-core CPUs simultaneously. This allows those processors to focus on the memory-intensive sieving and results in more useful NFS-relations found in less time.

Keywords: Cofactorization, Graphics Processing Unit, Number Field Sieve

1 Introduction

Today, the asymptotically fastest publicly known integer factorization method is the number field sieve (NFS, [46, 30]). It has been used to set several integer factorization records, most recently a 768-bit RSA modulus as described in [27]. In the first of its two main steps, pairs of integers called *relations* are collected. This is done by iterating a two-stage approach: *sieving* to collect a large batch of promising pairs, followed by the identification of the relatively few relations among them. Sieving requires a lot of memory and is commonly done on CPUs. The follow-up stage requires little memory and can be parallelized in multiple ways. It may therefore be cost-effective to offload this follow-up stage to a coprocessor. Most previous work in this direction focussed on offloading the elliptic curve integer factoring (ECM, [31]), which is only part of this follow-up stage. For graphics processing units (GPUs) this is considered in [7, 5, 10] and for reconfigurable hardware such as field-programmable gate arrays in [53, 45, 17, 14, 19, 32, 58]. To allow the CPUs to keep sieving, thus optimally using their memory, in this paper the possibility is explored to offload the *entire* follow-up stage to GPUs. We describe our approach, with a focus on modular and elliptic curve arithmetic, to do so on the many-core, memory-constrained GPU platform. Our results demonstrate that GPUs can be used as an efficient *high-throughput co-processor* for this application.

Our design strategy exploits the inherent task parallelism of the stage that follows the actual sieving, namely the fact that collected pairs can be processed independently in parallel. Because the integers involved are relatively small (at most 384 bits for our target number), we have chosen not to parallelize the integer arithmetic, thereby avoiding performance penalties due to inter-thread synchronization while maximizing the compute-to-memory-access ratio [5]. We use a single thread to process a single pair from the input batch, aiming to maximize the number of pairs processed per second. Because this requires a large number of registers per thread and potentially reduces the GPU utilization, we use integer arithmetic algorithms that minimize register usage and apply native multiply-and-add instructions wherever possible.

For each pair the follow-up stage consists of checking if two integer values, obtained by evaluating two bivariate integer polynomials at the point determined by the pair, are both

[*] Part of this work has been performed while the second author was working for Microsoft Research, WA, USA.

smooth, i.e., divisible by primes up to certain bounds. This is done sequentially: a first kernel filters the pairs for which the first polynomial value is smooth, once enough pairs have been collected a second kernel does the same for the second polynomial value, and pairs that pass both filters correspond to relations. Each kernel first computes the relevant polynomial value and then subjects it to a sequence of occasional compositeness tests and factorization attempts aimed at finding small factors.

We have determined good parameters for two different approaches: to find as many relations as possible ($\approx 99\%$ in a batch) and a faster one to find most relations ($\approx 95\%$ in a batch). The effectiveness of these approaches is demonstrated by integrating the GPU software with state-of-the-art NFS software [16] tuned for the factorization of the 768-bit modulus from [27]. A single GTX 580 GPU can serve between 3 and 10 Intel i7-3770K *quad-core* CPUs.

Cryptologic applications of GPUs have been considered before: symmetric cryptography in [33, 20, 56, 21, 44, 11, 18], asymmetric cryptography in [39, 54, 22] for RSA and in [54, 1, 9] for ECC, and enhancing symmetric [8] and asymmetric [7, 5, 6, 10] cryptanalysis.

Our source code will be made available.

2 Preliminaries

The Number Field Sieve. For details on how NFS works, see [30, 50]. Its major steps are polynomial selection, relation collection, and the matrix step. For this paper, an operational description of relation collection for numbers in the current range of interest suffices. For those numbers relation collection is responsible for about 90% of the computational effort.

Here we call an integer B-smooth if there is no prime-power larger than B that divides it (elsewhere such numbers are called B-powersmooth). Relation collection uses smoothness bounds $B_{\mathrm{r}}, B_{\mathrm{a}} \in \mathbf{Z}_{>0}$ and polynomials $f_{\mathrm{r}}(X), f_{\mathrm{a}}(X) \in \mathbf{Z}[X]$ such that f_{r} is of degree one, f_{a} is irreducible of (small) degree $d > 1$, and f_{r} and f_{a} have a common root modulo the number to be factored. The polynomials f_{r} and f_{a} are commonly referred to as the *rational* and the *algebraic* polynomial, respectively. A relation is a pair of coprime integers (a, b) with $b > 0$ such that $bf_{\mathrm{r}}(a/b)$ is B_{r}-smooth and $b^d f_{\mathrm{a}}(a/b)$ is B_{a}-smooth.

Relations are determined by successively processing relatively large *special primes* until sufficiently many relations have been found. A special prime q defines an index-q sublattice in $\mathbf{Z}^2$ of pairs (a, b) such that q divides $bf_{\mathrm{r}}(a/b)b^d f_{\mathrm{a}}(a/b)$. Sieving in the sublattice results in a collection of pairs for which $bf_{\mathrm{r}}(a/b)$ and $b^d f_{\mathrm{a}}(a/b)$ have relatively many small factors. To identify the relations, for all collected pairs the values $bf_{\mathrm{r}}(a/b)$ and $b^d f_{\mathrm{a}}(a/b)$ are further inspected. This can be done by first simultaneously *resieving* the $bf_{\mathrm{r}}(a/b)$-values to remove their small factors, then doing the same for the $b^d f_{\mathrm{a}}(a/b)$-values, after which any cofactors are dealt with on a pair-by-pair basis. Alternatively, cofactoring can be preceded by a pair-by-pair search for the small factors in $bf_{\mathrm{r}}(a/b)$ and $b^d f_{\mathrm{a}}(a/b)$, thus simplifying the sieving step. The latter approach is adopted here, to offload as much as possible from the regular CPU cores, including the calculation of the relevant $bf_{\mathrm{r}}(a/b)$- and $b^d f_{\mathrm{a}}(a/b)$-values. The steps involved in this extended (and thus somewhat misnomered) cofactoring are described in Section 3.

Montgomery arithmetic. For arithmetic modulo a fixed odd modulus m *Montgomery arithmetic* [35] may be used because it avoids trials during the divisions and allows simple coding. Let r be the machine radix (here $r = 2^{32}$), let $k \in \mathbf{Z}_{>0}$ be minimal such that $r^k > m$, and let $\mu = -m^{-1} \bmod r$. The *Montgomery representation* of an integer $x \in \mathbf{Z}/m\mathbf{Z}$ is defined as $\tilde{x} = xr^k \bmod m$. Given Montgomery representations $\tilde{x}, \tilde{y}$ of $x, y \in \mathbf{Z}/m\mathbf{Z}$, it follows that $\tilde{t}$ such that $t = (x \pm y) \bmod m$ is calculated as $\tilde{t} = (\tilde{x} \pm \tilde{y}) \bmod m$, and that $\tilde{s}$ such that $s = xy \bmod m$ satisfies $\tilde{s} = \tilde{x}\tilde{y}r^{-k} \bmod m$. This Montgomery product $\tilde{s}$ can be computed by first calculating the ordinary integer product $u = \tilde{x}\tilde{y}$, and by next performing *Montgomery reduction*: modulo m divide u by r^k by replacing k times in succession u by $(u + [((u \bmod r)\mu) \bmod r]m)/r$, then $\tilde{s} = u - m$ if $u \geq m$ and $\tilde{s} = u$ otherwise. If $0 \leq \tilde{x}, \tilde{y} < m$, then the same bound hold for $\tilde{s}$.

Jebelean's exact division. If n is known to be an integer multiple of an odd integer p, the quotient $\frac{n}{p}$ can be computed using an iteration very similar to Montgomery reduction: let $\mu = -p^{-1} \bmod r$, then $v = ((n \bmod r)(r - \mu)) \bmod r$ equals the least significant radix-r block $\frac{n}{p} \bmod r$ of $\frac{n}{p}$, after which n is replaced by $(n - vp)/r$ and the other radix-r blocks of $\frac{n}{p}$ are iteratively computed in the same way. This is known as *Jebelean's exact division method* [24].

3 Cofactoring Steps

This section lists the steps used to identify the relations among a collection of pairs of integers (a, b) that results from NFS sieving for one or more special primes. See [26] for related previous work. The notation is as in Section 2.

For all collected pairs (a, b) the values $bf_{\mathrm{r}}(a/b)$ and $b^d f_{\mathrm{a}}(a/b)$ can be calculated by observing that $b^k f(a/b) = \sum_{i=0}^{k} f_i a^i b^{k-i}$ for $f(X) = \sum_{i=0}^{k} f_i X^i \in \mathbf{Z}[X]$. The value $z = b^k f(a/b)$ is trivially calculated in $k(k-1)$ multiplications by initializing z as 0, and by replacing, for $i = 0$, 1, ..., k in succession, z by $z + f_i a^i b^{k-i}$, or, at the cost of an additional memory location, in $3k - 1$ multiplications by initializing $z = f_0$ and $t = a$ and by replacing, for $i = 1$, 2, ..., k in succession, z by $zb + f_i t$ and, if $i < k$, t by ta. Even with the most naive approach (as opposed to asymptotically faster methods), this is a negligible part of the overall calculation. The resulting values need to be tested for smoothness, with bound B_{r} for the $bf_{\mathrm{r}}(a/b)$-values and bound B_{a} for the $b^d f_{\mathrm{a}}(a/b)$-values.

For all pairs (a, b) both $bf_{\mathrm{r}}(a/b)$ and $b^d f_{\mathrm{a}}(a/b)$ have relatively many small factors (because the pairs are collected during NFS sieving). After shifting out all factors of two, other very small factors may be found using trial division, somewhat larger ones by Pollard $p - 1$ [47], and the largest ones using ECM [31]. These three methods are further described below. In our experiment (cf. 5.2) it turned out to be best to skip trial division for $bf_{\mathrm{r}}(a/b)$ and let Pollard $p - 1$ and ECM take care of the very small factors as well. Based on the findings reported in [28] or their GPU-incompatibility, other integer factorization methods like Pollard rho [48] or quadratic sieve [49] are not considered. It is occasionally useful to make sure that remaining cofactors are composite. An appropriate compositeness test is therefore described first.

Compositeness test. Let $m - 1 = 2^t u$ for $t \in \mathbf{Z}_{\geq 0}$ and odd $u \in \mathbf{Z}$. If for some $a \in (\mathbf{Z}/m\mathbf{Z})^*$ it is the case that $a^u \not\equiv 1 \bmod m$ and $a^{u2^i} \not\equiv -1 \bmod m$ for $0 \leq i < t$, then m is composite and a is a *witness* to m's compositeness. As shown in [34, 51], for composite m more than 75% of the integers in $\{1, 2, \ldots, m - 1\}$ are witnesses to m's compositeness.

This test is used as follows to process an m-value that is found as an as yet unfactored part of a polynomial value $bf_{\mathrm{r}}(a/b)$ or $b^d f_{\mathrm{a}}(a/b)$. If 2 is a witness to m's compositeness, then m is subjected to further factoring attempts; if not, the polynomial value is declared fully factored and the corresponding pair (a, b) is cast aside if $m > B_{\mathrm{r}}$ for $m \mid bf_{\mathrm{r}}(a/b)$ or $m > B_{\mathrm{a}}$ for $m \mid b^d f_{\mathrm{a}}(a/b)$. This carries the risk that a non-prime factor may appear in a supposedly fully factored polynomial value, or that a pair (a, b) is wrongly discarded. With a small probability to occur, either type of failure is of no concern in our cryptanalytic context.

Trial division. Given an odd integer n, all its prime factors up to some small trial division bound are removed using trial division. For each small odd prime p (possibly tabulated, if memory is available) first $\pi = (-p)^{-1} \bmod r$ is calculated (per thread, at negligible overhead), with $r = 2^{32}$ as in Section 2. Next, n is tested for divisibility by p: with u initialized as n and k the least integer such that $u < r^k$, the integer u is modulo p divided by r^k (using Montgomery reduction, with p and π in the roles of m and μ, respectively). If the resulting 32-bit value u satisfies $u \bmod p \equiv 0$, then n is divisible by p and the divisibility test is repeated with n replaced by $\frac{n}{p}$ (computed using Jebelean's method).

Pollard $p - 1$. The prime factors p of n for which $p - 1$ is B_1-smooth can be found at a cost of $O(B_1)$ multiplications modulo n by means of "stage 1" of Pollard's $p-1$ method [47]: with $t = a^k \bmod n$, for some $a \neq \pm 1 \bmod n$, $a \neq 0 \bmod n$ and k the product of all prime powers $\leq B_1$, the product of all such p divides $\gcd(t - 1, n)$. In practice the value $a = 2$ is used for efficiency reasons. If the order modulo n of t is at most B_2, for some bound $B_2 > B_1$, this can be exploited in "stage 2" [36], thereby allowing in $p - 1$ one additional prime factor between B_1 and B_2. Naively, $\gcd(t^\ell - 1, n)$ could be computed for all primes ℓ in $(B_1, B_2]$. A much faster but memory-consuming method uses the fast Fourier transform (cf. [38]). On GPUs a baby-step giant-step approach is more suitable and is used here. It follows from the description below and the optimizations described in [36].

Elliptic Curve Method. Stage 1 of Pollard's $p-1$ method uses $O(B_1)$ multiplications modulo n to find prime factors p of n for which the groups $(\mathbf{Z}/p\mathbf{Z})^*$ have B_1-smooth order. Thus, p can be found in time mostly linear in the largest prime factor of $p-1$. The elliptic curve method (ECM) for integer factorization [31] works analogously but replaces the fixed group $(\mathbf{Z}/p\mathbf{Z})^*$ of order $p - 1$ by a number of groups with orders behaving like random integers *close to* p: given one such group with B_1-smooth order, p can be found in $O(B_1)$ multiplications and additions modulo n. Trading off the number of groups attempted and the smoothness bound, finding p can heuristically be argued to take $\exp((\sqrt{2} + o(1))(\sqrt{\log p \log\log p}))$ elementary operations modulo n, where $p \to \infty$.

Like Pollard's $p - 1$ method, each ECM attempt operates on a group element and the product k of all prime powers $\leq B_1$, mimics the "$\bmod\ p$" operations by doing them "$\bmod\ n$", and hopes to run into the identity element $\bmod\, p$ but not $\bmod\, n$, if not in stage 1 then in stage 2. Where Pollard's method is based on arithmetic in the group of integers modulo the composite multiple n of p, ECM is based on arithmetic with "points" belonging to groups associated with elliptic curves over prime fields, mimicking those operations by doing them modulo the composite multiple n of those primes. Because the operations may not be well-defined, they may fail, thereby revealing a factor of n.

The current best approach to implement ECM, as used here, is *"$a = -1$" twisted Edwards curves* (based on [15, 4, 23, 3]) with extended twisted Edwards coordinates (improving on *Montgomery curves* [36] and methods from [57]). Below points are represented as pairs of projective points $((x : z), (y : t))$ for $x, z, y, t \in \mathbf{Z}/n\mathbf{Z}$, with *zero point* $((0 : 1), (1 : 1))$. Applying the additively written "group operation" requires a total of eight multiplications and squarings in $\mathbf{Z}/n\mathbf{Z}$. With initial point P the point kP can thus be calculated in $O(B_1)$ multiplications in $\mathbf{Z}/n\mathbf{Z}$, after which the gcd of n and the x-coordinate of kP is computed. Because the same k is often used, good addition-subtraction chains can be prepared (cf. [10]): for $B_1 = 256$, the point kP can be computed in 1400 multiplications and 1444 squarings modulo n. Due to the significant memory reduction this approach is particularly efficient for memory constrained devices like GPUs. We also select curves for which 16 divides the group order, further enhancing the success probability of ECM (cf. [2, Thm. 3.4 and 3.6] and [3]). More specifically we use "$a = -1$" twisted Edwards curve ($E : -x^2 + y^2 = 1 + dx^2y^2$) over $\mathbf{Q}$ with $d = -((g - 1/g)/2)^4$ such that $d(d + 1) \neq 0$ and $g \in \mathbf{Q} \setminus \{\pm 1, 0\}$.

Related work on stage 1 of ECM for cofactoring on constrained devices can be found in [53, 45, 17, 14, 19, 32, 58, 7, 5, 10]. Unlike these publications, the GPU-implementation presented here includes stage 2 of ECM, as it significantly improves the performance of ECM.

ECM Stage 2 on GPUs. The fastest known methods to implement stage 2 of ECM are FFT-based [12, 36, 37] and rather memory-hungry, which may explain why earlier constrained device ECM-cofactoring work did not consider stage 2. These methods are also incompatible with the memory restrictions of current GPUs. Below a baby-step giant-step approach [52] to stage 2 is described that is suitable for GPUs. Let $Q = kP$ be as above. Similar to the naive approach to stage 2 of Pollard's $p - 1$ method, the points ℓQ for the primes ℓ in $(B_1, B_2]$ can be computed and be compared to the zero point modulo a prime dividing p but not modulo n. The latter amounts to computing the gcd of n and the product of the x-coordinates of the points ℓQ. With N primes ℓ, computing all points requires about $8N$ multiplications in $\mathbf{Z}/n\mathbf{Z}$, assuming a few precomputed small even multiples of Q. Balancing the computational efforts of the two stages with $B_1 = 256$ as above, leads to $B_2 = 2803$ (and $N = 354$).

The baby-step giant step approach from [36] speeds up the calculation at the cost of more memory, while also exploiting that for Edwards curves and any point P it is the case that

$$\frac{y(P)}{t(P)} = \frac{y(-P)}{t(-P)}, \tag{1}$$

with $y(P)$ and $t(P)$ the y- and t-coordinate, respectively, of P.

For a giant-step value $w < B_1$, any ℓ as above can be written as $vw \pm u$ where $u \in U = \{u \in \mathbf{Z} : 1 \leq u \leq \frac{w}{2}, \gcd(u, w) = 1\}$, and $v \in V = \{v \in \mathbf{Z} : \lceil \frac{B_1}{w} - \frac{1}{2} \rceil \leq v \leq \lfloor \frac{B_2}{w} + \frac{1}{2} \rfloor\}$. Comparing $(vw - u)Q$ to the zero point modulo p but not modulo n amounts to checking if $\gcd(t(uQ)y(vwQ) - t(vwQ)y(uQ), n) \neq 1$. Because of (1), this compares $(vw + u)Q$ to the zero point as well. Hence, computation of $\gcd(m, n)$ for $m = \prod_{v \in V} \prod_{u \in U} (t(uQ)y(vwQ) - t(vwQ)y(uQ))$ suffices to check if Q has prime order in $(B_1, B_2]$. Optimal parameters balance the costs of the preparation of the $\frac{\varphi(w)}{2}$ tabulated baby-step values $(y(uQ) : t(uQ))$ (where φ is Euler's totient function) and on the fly computation of the giant-step values $(y(vwQ) : t(vwQ))$. Suboptimal, smaller w-values may be used to reduce storage requirements. For instance, the choice $w = 2 \cdot 3 \cdot 5 \cdot 7$ and $B_2 = 7770$ leads to 24 tabulated values and a total of 2904 multiplications and squarings modulo n, which matches the computational effort of stage 1 with $B_1 = 256$. Although $\gcd(u, w) = 1$ already avoids easy composites, the product can be restricted to those u, v for which one of $vw \pm u$ is prime if storage for about $\frac{B_2 - B_1}{w} \times \frac{\varphi(w)}{2}$ bits is available. With w and tabulated baby-step values as above, this increases B_2 to 8925 for a similar computational effort, but requires about 125 bytes of storage. A more substantial improvement is to define

$$y_v = \Big(\prod_{\tilde{v} \in V - \{v\}} t(\tilde{v}wQ)\Big)\Big(\prod_{\tilde{u} \in U} t(\tilde{u}Q)\Big)y(vwQ) \text{ and } y_u = \Big(\prod_{\tilde{u} \in U - \{u\}} t(\tilde{u}Q)\Big)\Big(\prod_{\tilde{v} \in V} t(\tilde{v}wQ)\Big)y(uQ),$$

and to replace m by $\prod_{v \in V} \prod_{u \in U} (y_v - y_u)$. This saves $2|V||U|$ of the $3|V||U|$ multiplications in the calculation of m at a cost that is linear in $|U| + |V|$ to tabulate the y_v and y_u values. For instance, it allows usage of $B_2 = 16\,384$ at an effort of 3368 modular multiplications.

4 GPU Implementation Details

In this section we outline our approach to implement the algorithms from Section 3 with a focus on the many-core GPU architecture. We used a quad-core Intel i7-3770K CPU running at 3.5 GHz with 16 GB of memory and an NVIDIA GeForce GTX 580 GPU, with 512 CUDA cores running at 1544 MHz and 1.5 GB of global memory, as further described below.

4.1 Compute unified device architecture

We focus on the GeForce x-series families for $x \in \{8, 9, 100, 200, 400, 500, 600, 700\}$, of the NVIDIA GPU architecture with the compute unified device architecture (CUDA) [41]. Our NVIDIA GeForce GTX 580 GPU belongs to the GeForce 400- and 500-series ([40]) of the Fermi architecture family. These GPUs support $32 \times 32 \to 32$-bit multiplication instructions, for both the least and most significant 32 bits of the result.

Each GPU contains a number of streaming multiprocessors (SMs), with each SM consisting of multiple scalar processor cores (SP). On a Fermi architecture GPU there are typically about 16 SMs and 32 SPs per SM, but numbers vary per model. Figure 1 depicts a high-level overview of a CUDA GPU architecture.

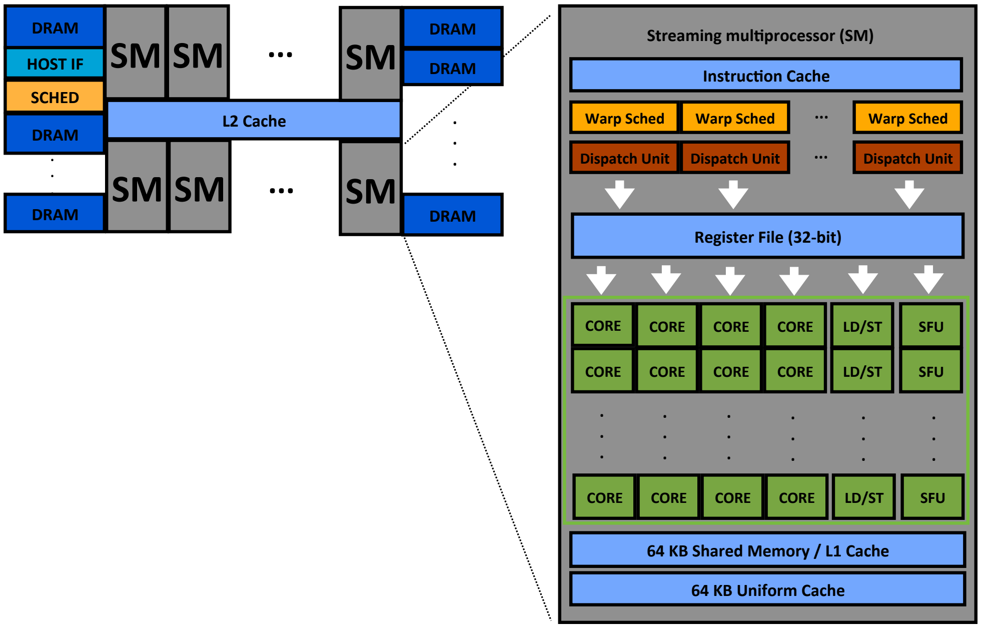

Fig. 1. High-level overview of a CUDA GPU architecture.

C for CUDA is an extension to the C language that employs the *single-instruction multiple-thread* (SIMT) model of massively parallel programming. The programmer defines *kernel functions*, which are compiled for and executed in parallel on the SPs such that each light-weight thread executes the same instructions but on its own data. A number of threads is grouped into a *thread block* which is scheduled on a single SM, the threads of which time-share the SPs.

Threads inside a thread block are executed in groups of 32 called *warps*. On Fermi architecture GPUs each SM has two warp schedulers and two instruction dispatch units. This means that two instructions, from separate warps, can be scheduled and dispatched at the same time. Switching between warps, filling the pipeline as much as possible, a high throughput rate can be sustained. The distinct possibilities of a conditional branch are executed serially by the threads inside a warp, with threads active only when their branch is executed. Multiple execution paths within a warp are thus best avoided.

Threads in the same block can communicate via on-chip shared memory and may synchronize their execution using barriers (a synchronization method which makes threads wait until all reach a certain point). There is a large but relatively slow amount of global memory that is accessible to all threads. Fermi architecture GPUs have an L1-cache for each SM, and a unified L2-cache together with fast constant (read-only) memory initialized by the CPU.

4.2 Modular arithmetic on GPUs

We used the parallel thread execution (PTX) instruction set and inline assembly wherever possible to simplify (cf. carry-handling) and speed-up (cf. multiply-and-add) our code; Table 7 in the Appendix lists the arithmetic assembly routines used. “Warp divergent” code was reduced to a minimum by converting most branches into *straight line code* to avoid different execution paths within a warp: branch-free code that executes both branches and uses a bit-mask to select the correct value was often found to be more efficient than “if-else” statements.

Practical performance. Our decision not to use parallel integer arithmetic dictates the use of algorithms with minimal register usage. For Montgomery multiplication, the most critical operation, we therefore preferred the plain interleaved schoolbook method to Karatsuba [25]; Algorithm 6 in the Appendix gives the CUDA pseudo-code for moduli of at least 96 bits.

Table 1 compares our results both with the state-of-the-art implementation from [29] benchmarked on an NVIDIA GTX 480 card (480 cores, 1401Mhz) and with the ideal peak throughput attainable on our GTX 580 GPU. Compared to [29] our throughput is up to twice better, especially for smaller (128-bit) moduli, even after the figures from [29] are scaled by a factor of $\frac{512}{480} \cdot \frac{1544}{1401}$ to account for our larger number of cores (512) and higher frequency (1544 MHz). For 32ℓ-bit moduli, with $\ell \in [3, 12]$ (i.e. moduli ranging from 96 to 384 bits), we counted the total number of multiplication and multiply-and-add instructions required by Algorithm 6 (including all calls to the auxiliary algorithms in the Appendix). The throughput of those instructions on our GPU is 0.5 per clock cycle per core, whereas the throughput of the addition instructions is 1 per clock cycle per core. Since we use fewer addition than multiplication instructions, our throughput count considers only the latter. Thus, our estimate for the Montgomery multiplication peak throughput is obtained as $\frac{1544 \cdot 10^6 \cdot 16 \cdot 32}{2m(\ell)}$ where $m(\ell) = \ell(4\ell + 1)$ is the number of multiplication instructions performed by Algorithm 6. In our benchmarks we transfer to the GPU two (distinct) operands and a modulus for each thread, and then compute one million modular multiplications using Algorithm 6 (using each output

Table 1. Benchmark results for the NVIDIA GTX 580 GPU for number of Montgomery multiplications per second and ECM trials per second for various modulus sizes. The Montgomery multiplication throughput reported in [29] was scaled as explained in the text. The estimated peak throughput based on an instruction count is also included together with the total number of dispatched threads. ECM used bounds $B_1 = 256$ and $B_2 = 16384$ (for a total of $2844 + 3368 = 6212$ Montgomery multiplications per trial).

	Leboeuf [29]	this work				
	Montgomery multiplications			#threads	ECM (8192 threads for all sizes)	
moduli	measured	measured	peak		trials	Montgomery multiplications
bitsize	(scaled, millions)	(millions)			(thousands)	measured (millions)
96		10119	10135	16384	1078	6697
128	2799	5805	5813	16384	674	4187
160	2261	3760	3764	16384	453	2814
192	1837	2631	2635	16384	309	1920
224	1507	1943	1947	15360	243	1510
256	1212	1493	1497	10240	180	1118
320	828	962	964	10240	107	665
384	600	671	672	9216	86	534

as one of the next inputs) before transferring the results back to the CPU. Our throughput turns out to be very close to the peak value.

4.3 Elliptic curve arithmetic on GPUs

When running stage 1 of ECM on memory constrained devices like GPUs, the large number of precomputed points required for windowing methods cannot be stored in fast memory. Thus, one is forced to settle for a (much) smaller window size, thereby reducing the advantage of using twisted Edwards curves. For example, in [7] windowing is not used at all because, citing [7], “Besides the base point, we cannot cache any other points”. Memory is also a problem in [5], where the faster curve arithmetic from Hisil et al. [23] is not used since this requires storing a fourth coordinate per point. These concerns were the motivation behind [10], the approach we adopted for stage 1 of ECM (as indicated in Section 3). For stage 2 we use the baby-step giant-step approach, optimized as described at the end of Section 3 for $B_2 \leq 32768$. Using bounds that balance the number of stage 1 and 2 multiplications does not necessarily balance the GPU running time of the two stages (this varies with the modulus size), but it is a good starting point for further optimization.

Table 1 lists the resulting performance figures, in terms of thousands of trials per second for various modulus sizes. Two jobs each consisting of 8192 threads were launched simultaneously, with each job per thread doing an ECM trial with the bounds as indicated, and with at the start a unique modulus per thread transferred to the GPU. The relatively high register usage of ECM reduces the number of threads that can be launched per SM before running out of registers. Nevertheless, and despite its large number of modular additions and subtractions, ECM manages to sustain a high Montgomery multiplication throughput. Except for the comparison to the work reported in [29], we have not been able to put our results in further perspective because we did not have access to other multiplication or ECM results or implementations in a comparable context.

5 Cofactorization on GPUs

This section describes our GPU approach to cofactoring, i.e., recognizing among the pairs (a, b) resulting from NFS sieving those pairs for which $bf_{\rm r}(a/b)$ is $B_{\rm r}$-smooth and $b^d f_{\rm a}(a/b)$ is $B_{\rm a}$-smooth. Approaches common on regular cores (resieving followed by sequential processing of the remaining candidates) allow pair-by-pair optimization with respect to the highest overall yield or yield per second while exploiting the available memory, but are incompatible with the memory and SIMT restrictions of current GPUs.

5.1 Cofactorization overview

Given our application, where throughput is important but latency almost irrelevant, it is a natural choice to process each pair in a single thread, eliminating the need for inter-thread communication, minimizing synchronization overhead, and allowing the scheduler to maximize pipelining by interleaving instructions from different warps. On the negative side, the large memory footprint per thread reduces the number of simultaneously active threads per SM.

The cofactorization stage is split into two GPU kernel functions that receive pairs (a, b) as input: the rational kernel outputs pairs for which $bf_{\rm r}(a/b)$ is $B_{\rm r}$-smooth to the algebraic kernel that outputs those pairs for which $b^d f_{\rm a}(a/b)$ is $B_{\rm a}$-smooth as well. The two kernels have the same code structure: all that distinguishes them is that the algebraic one usually has to handle larger values and a higher degree polynomial. To make our implementation flexible with respect to the polynomial selection, the maximum size of the polynomial values is a kernel parameter that is fixed at compile time and that can easily be changed together with the polynomial degree and coefficient size and the size of the inputs.

Kernel structure. Given a pair (a, b), a kernel-thread first evaluates the relevant polynomial, storing the odd part n of the resulting value along with identifying information i as a pair (i, n); if applicable the special prime is removed from n. The value n is then updated in the following sequence of steps, with all parameters set at run-time using a configuration file. First trial division may be applied up to a small bound. The resulting pairs (i, n) are regrouped depending on their radix-2^{32} sizes. The cost of the resulting inter-thread communication and synchronization is outweighed by the advantage of being able to run size-specific versions of the other steps. All threads in a warp then grab a pair (i, n) of the same size and each thread attempts to factor its n-value using Pollard's $p - 1$ method or ECM. If the resulting n is at most the smoothness bound, the kernel outputs the ith pair (a, b). If n's compositeness cannot be established or if n is larger than some user-defined threshold, the ith pair (a, b) is discarded. Pairs (i, n) with small enough composite n are regrouped and reprocessed. Figure 2 shows a pictorial example of a kernel execution flow.

This approach treats every pair (i, n) in the same group in the same way, which makes it attractive for GPUs. However, unnecessary computations may be performed: for instance, if a factoring attempt fails, compositeness does not need to be reascertained. Avoiding this requires divergent code which, as it turned out, degrades the performance. Also, factoring attempts may chance upon a factor larger than the smoothness bound, an event that goes by unnoticed as only the unfactored part is reported back. We have verified that the CPU easily discards such mishaps at negligible overhead.

Interaction between CPU and GPU. The CPU uses two programs to interact with the GPU. The first one adds batches of (a, b) pairs produced by the siever (which may be running on the CPU too) to a FIFO buffer and keeps track of special primes. The second program

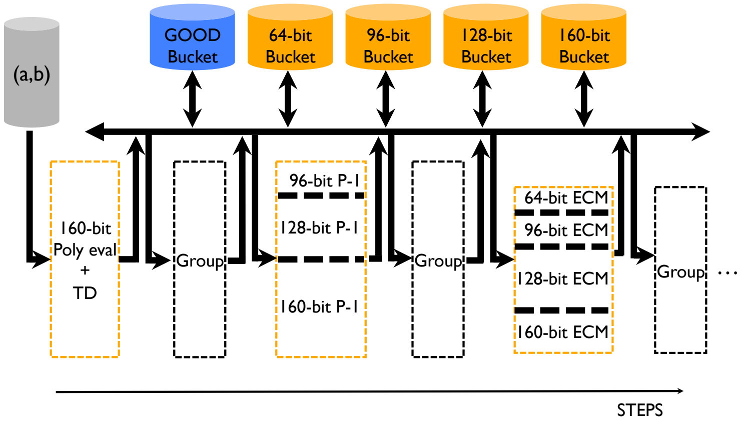

Fig. 2. An example of kernel execution flow where the values are assumed to be at most 160 bits. The height of the dashed rectangles is proportional to the number of values that are processed at a given step.

Table 2. Time in seconds to process a single special prime on all cores of a quad-core Intel i7-3770K CPU.

large primes	number of pairs after sieving	relations found	sieving time	cofactoring time	total time	% of time spent on cofactoring	relations per second
3	$\approx 5 \cdot 10^5$	125	25.6	4.0	29.6	13.5	4.22
4	$\approx 10^6$	137	25.9	6.1	32.0	19.1	4.28

controls the GPU by iterating the following steps (where the roles of the kernels may be reversed and the batch sizes depend on the GPU memory constraints and the kernel):

1. copy a batch from the FIFO buffer to the GPU;
2. launch the rational kernel on the GPU;
3. store the pairs output by the rational kernel in an intermediate buffer;
4. if the intermediate buffer does not contain enough pairs, return to Step 1;
5. copy a batch from the intermediate buffer to the GPU;
6. launch the algebraic kernel on the GPU (providing it with the proper special primes);
7. store the pairs output by the algebraic kernel in a file and return to Step 1.

Exploiting the GPU memory hierarchy. GPU performance strongly depends on where intermediate values are stored. We use constant memory for fixed data precomputed by the CPU and accessed by *all threads at the same time*: primes for trial division, polynomial coefficients, and baby-step giant-step table-indices for the second stages of factoring attempts. To lower register pressure, the fast shared memory per SM acts as a "user-defined cache" for the values most frequently accessed, such as the moduli n to be factored and the values $-n^{-1} \bmod 2^{32}$. The slower but much larger global memory stores the batch of (a, b) pairs along with their current n-values. To reduce memory overhead, the n-values are moved back and forth to shared memory after regrouping.

Table 3. Parameters choices for cofactoring. Later ECM attempts use larger bounds in the specified ranges.

desired yield	algorithm	rational kernel			algebraic kernel		
		attempts	B_1	B_2	attempts	B_1	B_2
95%	Pollard $p-1$	1	$[256, 2048]$	$[8192, 16384]$	1	$[256, 4096]$	$[16384, 32768]$
	ECM	$[5, 10]$	256	$[4096, 8192]$	10	$[256, 512]$	$[4096, 32768]$
99%	Pollard $p-1$	1	$[1024, 4096]$	$[8192, 32768]$	1	$[256, 2048]$	$[8192, 16384]$
	ECM	$[10, 12]$	$[256, 512]$	$[4096, 32768]$	$[10, 20]$	$[256, 512]$	$[4096, 32768]$

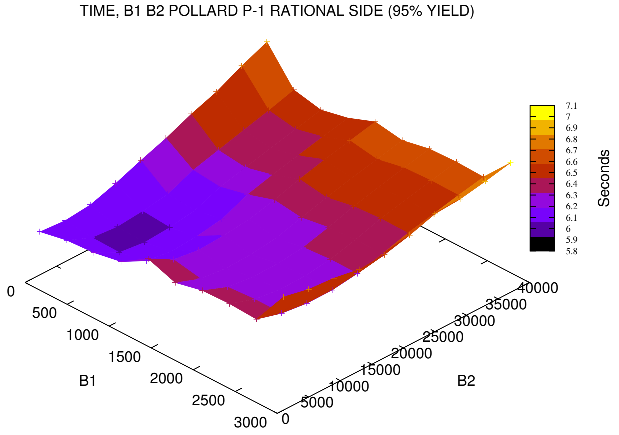

Fig. 3. Rational kernel cofactoring run times as a function of the Pollard $p-1$ bounds with desired yield 95%.

5.2 Parameter selection

For our experiments we applied the CPU NFS siever from [16] (obviously, with multi-threading enabled) to produce relations for the 768-bit number from [27]. Except for the special prime, three so-called *large primes* (i.e., primes not used for sieving but bounded by the applicable smoothness bound) are allowed in the rational polynomial value, whereas on the algebraic side the number of large primes is limited to three or four. Table 2 lists typical figures obtained when processing a single special prime in either setting; the percentages are indicative for NFS factorizations in general. The relatively small amount of time spent by the CPU on cofactoring suggests various ways to select optimal GPU parameters. One approach is aiming for as many relations per second as possible. Another approach is to aim for a certain fixed percentage of the relations among the pairs produced by NFS sieving, and then to select parameters that minimize the GPU time (thus maximizing the number of CPUs that can be served by a GPU). Although in general a fixed percentage cannot be ascertained, it can be done for experimental runs covering a fixed set of special prime ranges, and the resulting parameters can be used for production runs covering all special primes. Here we report on this latter approach in two settings: aiming for all (denoted by "99%") or for 95% of all relations.

Experiments. For a fixed set of special prime ranges and both large prime settings we determined all (a, b) pairs generated by NFS sieving and counted all relations resulting from those (a, b) pairs. Next, we processed the (a, b) pairs for either setting using our GPU cofactoring program, while widely varying all possible choices and aiming for 95% or 99% of all relations.

Table 4. Approximate timings in seconds of cofactoring steps to process approximately 50 million (a, b) pairs, measured using the CUDA `clock64` instruction. The wall clock time (measured with the unix `time` utility) includes the kernel launch overhead the CPU/GPU memory transfer and all CPU book-keeping operations.

number of large primes	desired yield	kernel	polynomial evaluation	trial division	Pollard $p-1$	ECM	regrouping	total	wall clock
3	95%	rational	0.05	-	56.42	149.49	5.97	211.94	263
		algebraic	0.10	0.36	6.21	39.05	0.44	46.16	
	99%	rational	0.05	-	79.19	213.15	7.75	300.16	367
		algebraic	0.10	0.36	10.84	48.93	0.68	60.91	
4	95%	rational	0.06	-	57.50	122.66	7.22	187.45	324
		algebraic	0.18	0.88	15.75	110.75	1.11	128.68	
	99%	rational	0.06	-	57.48	158.49	8.53	224.57	479
		algebraic	0.18	0.89	27.47	212.47	1.79	242.80	

This led to the observations below. Although other input numbers (than our 768-bit modulus) may lead to other choices our results are indicative for generic large composites.

We found that the rational kernel should be executed first, that it is best to skip trial division in the rational kernel, and that a small trial division bound (say, 200) in the algebraic kernel leads to a slight speed-up compared to not using algebraic trial division. For all other steps the two kernels behave similarly, though with somewhat different parameters that also depend on the desired yield (but not on the large prime setting). The details are listed in Table 3. Not shown there are the discarding thresholds that slightly decrease with the number of ECM attempts. Actual run times of the cofactoring steps are given in Table 4. Rational batches contain 3.5 times more pairs than algebraic ones (because the algebraic kernel has to handle larger values). For 3 large primes the rational kernel is called 5 times more often than the algebraic one, for 4 large primes 2.2 times more often.

Varying the bounds of the Pollard $p-1$ factoring attempt on the rational side within reasonable ranges does not noticeably affect the yield because almost all missed prime factors are found by the subsequent ECM attempts. However, early removal of small primes may reduce the sizes, thus reducing the ECM run time and, if not too much time is spent on Pollard $p-1$, also the overall run time. This is depicted in Figure 3. Note that in record breaking ECM work the number of trials is much larger; however, according to [55] the empirically determined numbers reported in Table 3 are in the theoretically optimal range.

5.3 Performance results

Table 5 summarizes the results when the same special prime as in Table 2 is processed, but now with GPU-assistance. The figures clearly show that farming out cofactoring to a GPU is

Table 5. GPU cofactoring for a single special prime. The number of quad-core CPUs that can be served by a single GPU is given in the second to last column.

large primes	number of pairs after sieving	desired yield	seconds	CPU/GPU ratio	relations found
3	$\approx 5 \cdot 10^5$	95%	2.6	9.8	132
		99%	3.7	6.9	136
4	$\approx 10^6$	95%	6.5	4.0	159
		99%	9.6	2.7	165

Table 6. Processing multiple special primes with desired yield 99%.

large primes	special primes	number of pairs after sieving	setting	total seconds	relations found	relations per second
3	100	$\approx 5 \cdot 10^7$	CPU only	2961	12523	4.23
			CPU and GPU	2564	13761	5.37
4	50	$\approx 5 \cdot 10^7$	CPU only	1602	6855	4.28
			CPU and GPU	1300	8302	6.39

advantageous from an overall run time point of view and that, depending on the yield desired, a single GPU can keep up with multiple quad-core CPUs. Remarkably, more relations may be found given the same collection of (a, b) pairs: with an adequate number of GPUs each special prime can be processed faster and produces more relations. Based on more extensive experiments the overall performance gain measured in "relations per second" found with and without GPU assistance is 27% in the 3 large primes case and 50% in the 4 large primes case (cf. Table 6).

Including equipment and power expenses in the analysis is much harder, as illustrated by (unrelated) experiments in [43]. Relative power and purchase costs vary constantly, and the power consumption of a GPU running CUDA applications depends on the configuration and the operations performed [13]. For instance, global memory accesses account for a large fraction of the power consumption and the effect on the power consumption of arithmetic instructions depends more on their throughput than on their type. We have not carried out actual power consumption measurements comparing the settings from Table 6.

6 Conclusion

It was shown that modern GPUs can be used to accelerate a compute-intensive part of the relation collection step of the number field sieve integer factorization method. Strategies were outlined to perform the entire cofactorization stage on a GPU. Integration with state-of-the-art lattice siever software indicates that a performance gain of up to 50% can be expected for the relation collection step of factorization of numbers in the current range of interest, if a single GPU can assist a regular multi-core CPU. Because relation collection for such numbers is responsible for about 90% of the total factoring effort the overall gain may be close to 45%; we have no experience with other sizes yet.

It is a subject of further research if a speed-up can be obtained using other types of graphic cards (to which we did not have access). In particular it would be interesting to explore if and how lower-end CUDA enabled GPUs can still be used for the present application and if the larger memory of more recent cards such as the GeForce GTX 780 Ti or GeForce GTX Titan can be exploited. Given our results we consider it unlikely that it would be advantageous to combine multiple GPUs using NVIDIA's scalable link interface.

Acknowledgements. This work was supported by the Swiss National Science Foundation under grant number 200020-132160. We gratefully acknowledge comments by the anonymous referees.

References

1. S. Antao, J.-C. Bajard, and L. Sousa. Elliptic curve point multiplication on GPUs. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 192–

199, 2010.

2. R. Barbulescu, J. W. Bos, C. Bouvier, T. Kleinjung, and P. L. Montgomery. Finding ECM-friendly curves through a study of Galois properties. In E. W. Howe and K. S. Kedlaya, editors, *Algorithmic Number Theory Symposium – ANTS 2012*, volume 1 of *The Open Book Series*, pages 63–86. Mathematical Sciences Publishers, 2013.
3. D. J. Bernstein, P. Birkner, and T. Lange. Starfish on strike. In M. Abdalla and P. S. L. M. Barreto, editors, *Latincrypt*, volume 6212 of *Lecture Notes in Computer Science*, pages 61–80. Springer, Heidelberg, 2010.
4. D. J. Bernstein, P. Birkner, T. Lange, and C. Peters. ECM using Edwards curves. *Mathematics of Computation*, 82(282):1139–1179, 2013.
5. D. J. Bernstein, H.-C. Chen, M.-S. Chen, C.-M. Cheng, C.-H. Hsiao, T. Lange, Z.-C. Lin, and B.-Y. Yang. The billion-mulmod-per-second PC. In *Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2009*, pages 131–144, 2009.
6. D. J. Bernstein, H.-C. Chen, C.-M. Cheng, T. Lange, R. Niederhagen, P. Schwabe, and B.-Y. Yang. ECC2K-130 on NVIDIA GPUs. In G. Gong and K. C. Gupta, editors, *Progress in Cryptology – INDOCRYPT 2010*, volume 6498 of *Lecture Notes in Computer Science*, pages 328–346. Springer-Verlag Berlin Heidelberg, 2010.
7. D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on graphics cards. In A. Joux, editor, *Eurocrypt 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer, Heidelberg, 2009.
8. M. Bevand. MD5 Chosen-Prefix Collisions on GPUs. Black Hat, 2009. Whitepaper.
9. J. W. Bos. Low-latency elliptic curve scalar multiplication. *International Journal of Parallel Programming*, 40(5):532–550, 2012.
10. J. W. Bos and T. Kleinjung. ECM at work. In X. Wang and K. Sako, editors, *Asiacrypt 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 467–484. Springer Berlin Heidelberg, 2012.
11. J. W. Bos and D. Stefan. Performance analysis of the SHA-3 candidates on exotic multi-core architectures. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 279–293. Springer, Heidelberg, 2010.
12. R. P. Brent. Some integer factorization algorithms using elliptic curves. *Australian Computer Science Communications*, 8:149–163, 1986.
13. S. Collange, D. Defour, and A. Tisserand. Power consumption of GPUs from a software perspective. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 914–923, Berlin, Heidelberg, 2009. Springer-Verlag.
14. G. de Meulenaer, F. Gosset, G. M. de Dormale, and J.-J. Quisquater. Integer factorization based on elliptic curve method: Towards better exploitation of reconfigurable hardware. In *Field-Programmable Custom Computing Machines – FCCM 2007*, pages 197–206. IEEE Computer Society, 2007.
15. H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44:393–422, July 2007.
16. J. Franke and T. Kleinjung. GNFS for linux. Software, 2012.
17. K. Gaj, S. Kwon, P. Baier, P. Kohlbrenner, H. Le, M. Khaleeluddin, and R. Bachimanchi. Implementing the elliptic curve method of factoring in reconfigurable hardware. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 119–133. Springer, Heidelberg, 2006.
18. J. Gilger, J. Barnickel, and U. Meyer. GPU-acceleration of block ciphers in the OpenSSL cryptographic library. In D. Gollmann and F. Freiling, editors, *Information Security*, volume 7483 of *Lecture Notes in Computer Science*, pages 338–353. Springer Berlin Heidelberg, 2012.
19. T. Güneysu, T. Kasper, M. Novotny, C. Paar, and A. Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57:1498–1513, 2008.
20. O. Harrison and J. Waldron. AES encryption implementation and analysis on commodity graphics processing units. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 209–226. Springer, Heidelberg, 2007.
21. O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th conference on Security symposium*, pages 195–209. USENIX Association, 2008.
22. O. Harrison and J. Waldron. Efficient acceleration of asymmetric cryptography on graphics hardware. In B. Preneel, editor, *Africacrypt 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 350–367. Springer, Heidelberg, 2009.

23. H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *Asiacrypt 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer, Heidelberg, 2008.
24. T. Jebelean. An algorithm for exact division. *Journal of Symbolic Computation*, 15(2):169–180, 1993.
25. A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. Number 145 in Proceedings of the USSR Academy of Science, pages 293–294, 1962.
26. T. Kleinjung. Cofactorisation strategies for the number field sieve and an estimate for the sieving step for factoring 1024-bit integers. In *Special-purpose Hardware for Attacking Cryptographic Systems – SHARCS 2006*, 2006.
27. T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In T. Rabin, editor, *Crypto 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350. Springer, Heidelberg, 2010.
28. A. Kruppa. A software implementation of ECM for NFS. Research Report RR-7041, INRIA, 2009. `http://hal.inria.fr/inria-00419094/PDF/RR-7041.pdf`.
29. K. Leboeuf, R. Muscedere, and M. Ahmadi. A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2593–2596, 2013.
30. A. K. Lenstra and H. W. Lenstra, Jr. *The Development of the Number Field Sieve*, volume 1554 of *Lecture Notes in Mathematics*. Springer-Verslag, 1993.
31. H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.
32. D. Loebenberger and J. Putzka. Optimization strategies for hardware-based cofactorization. In M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography – SAC*, volume 5867 of *Lecture Notes in Computer Science*, pages 170–181. Springer, Heidelberg, 2009.
33. S. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68, 2007.
34. G. L. Miller. Riemann's hypothesis and tests for primality. In *Proceedings of seventh annual ACM symposium on Theory of computing*, STOC '75, pages 234–239. ACM, 1975.
35. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
36. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
37. P. L. Montgomery. *An FFT extension of the elliptic curve method of factorization.* PhD thesis, University of California, 1992.
38. P. L. Montgomery and R. D. Silverman. An FFT extension to the p-1 factoring algorithm. *Mathematics of Computation*, 54(190):839–854, 1990.
39. A. Moss, D. Page, and N. P. Smart. Toward acceleration of RSA using 3D graphics hardware. In S. D. Galbraith, editor, *Proceedings of the 11th IMA international conference on Cryptography and coding*, Cryptography and Coding 2007, pages 364–383. Springer-Verlag, 2007.
40. NVIDIA. Fermi architecture whitepaper. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2010.
41. NVIDIA. Cuda programming guide 5. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, 2013.
42. NVIDIA. Parallel thread execution isa version 3.2. `http://docs.nvidia.com/cuda/parallel-thread-execution/index.html`, 2013.
43. NVIDIA Developer Zone. https://devtalk.nvidia.com/default/topic/491799/gtx-590-cuda-power-tests/. 2011.
44. D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software AES encryption. In S. Hong and T. Iwata, editors, *Fast Software Encryption – FSE 2010*, volume 6147 of *Lecture Notes in Computer Science*, pages 75–93. Springer, Heidelberg, 2010.
45. J. Pelzl, M. Šimka, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer, and C. Paar. Area-time efficient hardware architecture for factoring integers with the elliptic curve method. *Information Security, IEE Proceedings on*, 152(1):67–78, 2005.
46. J. M. Pollard. The lattice sieve. pages 43–49 in [30].
47. J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974.
48. J. M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

49. C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarsson, editors, *Eurocrypt 1984*, volume 209 of *Lecture Notes in Computer Science*, pages 169–182. Springer, Heidelberg, 1985.
50. C. Pomerance. A tale of two sieves. *Biscuits of Number Theory*, 85, 2008.
51. M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.
52. D. Shanks. Class number, a theory of factorization, and genera. In D. J. Lewis, editor, *Symposia in Pure Mathematics*, volume 20, pages 415–440. American Mathematical Society, 1971.
53. M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, and V. Fischer. Hardware factorization based on elliptic curve method. In *Field-Programmable Custom Computing Machines – FCCM 2005*, pages 107–116. IEEE Computer Society, 2005.
54. R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 79–99. Springer, Heidelberg, 2008.
55. G. Xin. Fast smoothness test. Semester project report, June 2013.
56. J. Yang and J. Goodman. Symmetric key cryptography on modern graphics hardware. In K. Kurosawa, editor, *Asiacrypt*, volume 4833 of *Lecture Notes in Computer Science*, pages 249–264. Springer, Heidelberg, 2007.
57. P. Zimmermann and B. Dodson. 20 years of ECM. In F. Hess, S. Pauli, and M. E. Pohst, editors, *Algorithmic Number Theory – ANTS-VII*, volume 4076 of *Lecture Notes in Computer Science*, pages 525–542. Springer, Heidelberg, 2006.
58. R. Zimmermann, T. Güneysu, and C. Paar. High-performance integer factoring with reconfigurable devices. In *Field Programmable Logic and Applications – FPL 2010*, pages 83–88. IEEE, 2010.

Appendix

Let $r = 2^{32}$.

Table 7. Pseudo-code notation for CUDA PTX assembly instructions [42] used in our implementation. Function parameters are 32-bit unsigned integers and the suffixes are analogous to the actual CUDA PTX suffixes. We denote by f the single-bit carry flag set by instructions with suffix ".cc".

Pseudo-code notation	Operation	Carry flag effect
addc(c, a, b)	$c \leftarrow a + b + f \bmod r$	
addc.cc(c, a, b)	$c \leftarrow a + b + f \bmod r$	$f \leftarrow \lfloor (a + b + f)/r \rfloor$
subc(c, a, b)	$c \leftarrow a - b - f \bmod r$	
subc.cc(c, a, b)	$c \leftarrow a - b - f \bmod r$	$f \leftarrow \lfloor (a - b - f)/r \rfloor$
mul.lo(c, a, b)	$c \leftarrow a \cdot b \bmod r$	
mul.hi(c, a, b)	$c \leftarrow \lfloor (a \cdot b)/r \rfloor$	
mad.lo.cc(d, a, b, c)	$d \leftarrow a \cdot b + c \bmod r$	$f \leftarrow \lfloor ((a \cdot b) \bmod r + c)/r \rfloor$
madc.lo.cc(d, a, b, c)	$d \leftarrow a \cdot b + c + f \bmod r$	$f \leftarrow \lfloor ((a \cdot b) \bmod r + c + f)/r \rfloor$
mad.hi.cc(d, a, b, c)	$d \leftarrow (\lfloor (a \cdot b)/r \rfloor + c) \bmod r$	$f \leftarrow \lfloor (\lfloor (a \cdot b)/r \rfloor + c)/r \rfloor$
madc.hi.cc(d, a, b, c)	$d \leftarrow (\lfloor (a \cdot b)/r \rfloor + c + f) \bmod r$	$f \leftarrow \lfloor (\lfloor (a \cdot b)/r \rfloor + c + f)/r \rfloor$

Algorithm 1 Mul(Z, x, Y)

```
**Input:** Integers $x$ and $Y = \sum_{i=0}^{n-1} Y_i r^i$ such that $0 \leq x, Y_i < r$ for $0 \leq i < n$.
**Output:** $Z = x \cdot Y = \sum_{i=0}^{n} Z_i r^i$.
  mul.lo$(Z_0, x, Y_0)$
  mul.hi$(Z_1, x, Y_0)$
  mad.lo.cc$(Z_1, x, Y_1, Z_1)$
  mul.hi$(Z_2, x, Y_1)$
  **for** $i = 2$ to $n - 2$ **do**
    madc.lo.cc$(Z_i, x, Y_i, Z_i)$
    mul.hi$(Z_{i+1}, x, Y_i)$
  madc.lo.cc$(Z_{n-1}, x, Y_{n-1}, Z_{n-1})$
  madc.hi$(Z_n, x, Y_{n-1}, 0)$
  **return** $Z \quad (= \sum_{i=0}^{n} Z_i r^i)$
```

Algorithm 2 Sub(Z, Y)

```
**Input:** Integers $Z = \sum_{i=0}^{n} Z_i r^i$ and $Y = \sum_{j=0}^{n-1} Y_j r^j$ such that $0 \leq Z_i, Y_j < r$ for $0 \leq i \leq n$, $0 \leq j < n$, and
  $0 \leq Z < 2Y$.
**Output:** If $Z \geq Y$ then $Z = Z - Y = \sum_{i=0}^{n} Z_i r^i$ with $Z_n = 0$. Otherwise $Z = r^{n+1} - (Y - Z) \bmod r^{n+1} =$
  $\sum_{i=0}^{n} Z_i r^i$ with $Z_n = r - 1$.
  sub.cc$(Z_0, Z_0, Y_0)$
  **for** $i = 1$ to $n - 1$ **do**
    subc.cc$(Z_i, Z_i, Y_i)$
  subc$(Z_n, Z_n, 0)$
  **return** $Z \quad (= \sum_{i=0}^{n} Z_i r^i)$
```

Algorithm 3 PredicateAdd(Z, Y, p) (where $a \wedge b$ computes the bitwise logical AND operation on each pair of corresponding bits in a and b)

```
Input: Integers $Z = \sum_{i=0}^{n-1} Z_i r^i$, $Y = \sum_{i=0}^{n-1} Y_i r^i$, and $p \in \{0, r-1\}$ such that $0 \leq Z_i, Y_i < r$ for $0 \leq i < n$,
    and $0 \leq Z < r^n$.
Output: $Z = Z + 0$ if $p = 0$ and $Z = Z + Y$ if $p = r - 1$.
    add.cc$(Z_0, Z_0, Y_0 \wedge p)$
    for $i = 1$ to $n - 2$ do
        addc.cc$(Z_i, Z_i, Y_i \wedge p)$
    addc$(Z_{n-1}, Z_{n-1}, Y_{n-1} \wedge p)$
    return $Z \quad (= \sum_{i=0}^{n-1} Z_i r^i)$
```

Algorithm 4 MulAddShift(Z, x, Y, c)

```
Input: Integers $Z = \sum_{i=0}^{n} Z_i r^i$, $Y = \sum_{j=0}^{n-1} Y_j r^j$, $x$ and $c$ such that $0 \leq x, Z_i, Y_j < r$ for $0 \leq i \leq n$, $0 \leq j < n$
    and $c \in \{0, 1\}$.
Output: $Z = \lfloor (Z + x \cdot Y + c r^{n+1})/r \rfloor = \sum_{i=0}^{n} Z_i r^i$
    mad.lo.cc$(Z_0, x, Y_0, Z_0)$
    for $i = 1$ to $n - 1$ do
        madc.lo.cc$(Z_i, x, Y_i, Z_i)$
    addc$(Z_n, Z_n, 0)$
    mad.hi.cc$(Z_0, x, Y_0, Z_1)$
    for $i = 2$ to $n$ do
        madc.hi.cc$(Z_{i-1}, x, Y_{i-1}, Z_i)$
    addc$(Z_n, c, 0)$
    return $Z \quad (= \sum_{i=0}^{n} Z_i r^i)$
```

Algorithm 5 MulAdd(Z, c, x, Y)

```
Input: Integers $Z = \sum_{i=0}^{n} Z_i r^i$, $Y = \sum_{j=0}^{n-1} Y_j r^j$, and $x$ such that $0 \leq x, Z_i, Y_j < r$ for $0 \leq i \leq n$, $0 \leq j < n$,
    and $0 \leq Z < 2r^n$.
Output: $Z = (Z + x \cdot Y) \bmod r^{n+1} = \sum_{i=0}^{n} Z_i r^i$, $c = \lfloor (Z + x \cdot Y)/r^{n+1} \rfloor$ $(c \in \{0, 1\})$.
    mad.lo.cc$(Z_0, x, Y_0, Z_0)$
    for $i = 1$ to $n - 1$ do
        madc.lo.cc$(Z_i, x, Y_i, Z_i)$
    addc$(Z_n, Z_n, 0)$
    mad.hi.cc$(Z_1, x, Y_0, Z_1)$
    for $i = 2$ to $n - 1$ do
        madc.hi.cc$(Z_i, x, Y_{i-1}, Z_i)$
    $c \leftarrow Z_n$
    madc.hi.cc$(Z_n, x, Y_{n-1}, Z_n)$
    $c \leftarrow (c > Z_n)$ // $c \in \{0, 1\}$
    return $Z \quad (= \sum_{i=0}^{n} Z_i r^i)$
```

Algorithm 6 Radix-2^{32} interleaved Montgomery multiplication (we assume $n > 2$).

```
Input: Integers $A, B, M, \mu$ such that $A = \sum_{i=0}^{n-1} A_i r^i$ with $0 \leq A_i < r$, $0 \leq B < M < r^n$, and $\mu =$
    $(-M^{-1}) \bmod r$.
Output: Integer $C = \frac{A \cdot B}{r}^{n} \bmod M = \sum_{i=0}^{n-1} C_i r^i$ with $0 \leq C_i < r$ and $0 \leq C < M$.
  Mul$(C, A_0, B)$
  mul.lo$(q, C_0, \mu)$
  MulAddShift$(C, q, M)$
  for $i = 1$ to $n - 1$ do
      MulAdd$(C, c, A_i, B)$ // $c$ is a temporary unsigned integer variable
      mul.lo$(q, C_0, \mu)$
      MulAddShift$(C, q, M, c)$
  Sub$(C, M)$
  PredicateAdd$(C, M, C_n)$ // $C_n \in \{0, r - 1\}$
```