ECC2K-130 on NVIDIA GPUs

Daniel J. Bernstein[1], Hsieh-Chung Chen[2], Chen-Mou Cheng[3], Tanja Lange[4], Ruben Niederhagen[3,4], Peter Schwabe[4], and Bo-Yin Yang[2]

[1] Department of Computer Science
University of Illinois at Chicago
851 S. Morgan Street, Chicago, IL 60607-7053, USA
djb@cr.yp.to
[2] Institute of Information Science
Academia Sinica
128, Section 2, Academia Road, Taipei 11529, Taiwan
{kc,by}@crypto.tw
[3] Department of Electrical Engineering
National Taiwan University
1, Section 4, Roosevelt Road, Taipei 10617, Taiwan
doug@crypto.tw, ruben@polycephaly.org
[4] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, 5600 MB Eindhoven, Netherlands
tanja@hyperelliptic.org, peter@cryptojedi.org

Abstract. A major cryptanalytic computation is currently underway on multiple platforms, including standard CPUs, FPGAs, PlayStations and GPUs, to break the Certicom ECC2K-130 challenge. This challenge is to compute an elliptic-curve discrete logarithm on a Koblitz curve over $\mathbb{F}_{2^{131}}$. Optimizations have reduced the cost of the computation to approximately 2^{77} bit operations in 2^{61} iterations.

GPUs are not designed for fast binary-field arithmetic; they are designed for highly vectorizable floating-point computations that fit into very small amounts of static RAM. This paper explains how to optimize the ECC2K-130 computation for this unusual platform. The resulting GPU software performs more than 63 million iterations per second, including 320 million $\mathbb{F}_{2^{131}}$ multiplications per second, on a $500 NVIDIA GTX 295 graphics card. The same techniques for finite-field arithmetic and elliptic-curve arithmetic can be reused in implementations of larger

Permanent ID of this document: 1957e89d79c5a898b6ef308dc10b0446. Date of this document: 2012.01.02. This work was sponsored in part by the National Science Foundation under grant ITR–0716498, in part by Taiwan's National Science Council under grant NSC-96-2221-E-001-031-MY3, and under grant NSC-96-2218-E-001-001, also through the Taiwan Information Security Center under grant NSC-97-2219-E-001-001, and under grant NSC-96-2219-E-011-008, in part by the Netherlands National Computing Facilities foundation under grant MP-185-10, and in part by the European Commission through the ICT Programme under Contract ICT–2007–216676 ECRYPT II and the ICT Programme under Contract ICT–2007–216499 CACE.

systems that are secure against similar attacks, making GPUs an interesting option as coprocessors when a busy Internet server has many elliptic-curve operations to perform in parallel.

Keywords: Graphics Processing Unit (GPU), Elliptic Curve Cryptography, Pollard rho, qhasm.

1 Introduction

The elliptic-curve discrete-logarithm problem (ECDLP) is the number-theoretic problem behind elliptic-curve cryptography (ECC): the problem of computing a user's ECC secret key from his public key. Pollard's rho method solves this problem in $O(\sqrt{\ell})$ iterations, where ℓ is the largest prime divisor of the order of the base point. A parallel version of the algorithm by van Oorschot and Wiener [20] provides a speedup by a factor of $\Theta(N)$ when running on N computers, if ℓ is larger than a suitable power of N. In several situations a group automorphism of small order m provides a further speedup by a factor of $\Theta(\sqrt{m})$. No further speedups are known for any elliptic curve chosen according to standard security criteria; this is the end of the story.

However, these asymptotic iteration counts ignore many factors that have an important influence on the cost of an attack. Understanding the hardness of a specific ECDLP requires a more thorough investigation. The publications summarized on `www.keylength.com`, giving recommendations for concrete cryptographic key sizes, all extrapolate from such investigations. To reduce extrapolation errors it is important to use as many data points as possible, and to push these investigations beyond the ECDLPs that have been carried out before.

Certicom published a list of ECDLP prizes in 1997 [12] in order to "increase the cryptographic community's understanding and appreciation of the difficulty of the ECDLP". These challenges range from very easy exercises, solved in 1997 and 1998, to serious cryptanalytic challenges. The last Certicom challenge that was publicly broken was a 109-bit ECDLP in 2004. Certicom had already predicted the lack of further progress: it had stated in [12, page 20] that the subsequent challenges were "expected to be infeasible against realistic software and hardware attacks, unless of course, a new algorithm for the ECDLP is discovered."

Since then new hardware platforms have become available to the attacker. Processor design has moved away from increasing the clock speed and towards increasing the number of cores. This means that implementations need to be parallelized in order to make full use of the processor. Running a serial implementation on a recent processor might take longer than 5 years ago, because the average clock speed has decreased, but if this implementation can be parallelized and occupy the entire processor then the implementation will run much faster. An extreme example of this high parallelism at reduced clock speed is the NVIDIA GTX 295 graphics card. This card contains two G200b Graphics Processing Units (GPUs); each GPU contains 30 cores; each core contains 8 ALUs; each ALU is capable of performing a 32-bit operation every cycle. Each ALU

operates at just 1.242 GHz, but 480 ALUs together have a tremendous amount of computational power.

Certicom's estimate a decade ago was that ECC2K-130, the first "infeasible" challenge, would require (on average) 2700000000 "machine days" of computation. Our main result is that a cluster of just 534 graphics cards running our software would solve ECC2K-130 in just 24 months.

In this paper we explain how we use Pollard's rho algorithm to compute ECDLP on GPUs. In particular, we give details on how we implement binary-field multiplication, a problem that at first seems quite poorly suited to GPUs, and on how we implement a complete ECDLP iteration function. Some of our optimizations are specific to ECC2K-130, but most of our implementation techniques can be reused in larger ECDLP computations. Furthermore, the binary-field arithmetic operations that we optimize are also the primary bottlenecks in various cryptographic computations; the same implementation techniques open up the interesting possibility of using GPUs as high-performance coprocessors to offload binary-field ECC computations from busy Internet servers.

This paper is part of a large collaborative project that has optimized ECDLP computations for several different platforms and that aims to break ECC2K-130. See [1] and `http://www.ecc-challenge.info` for more information about the project. All of the platforms use the same ECC2K-130 iteration function, allowing an objective comparison of the power of different platforms and putting our GPU speeds in perspective: finishing the computation in two years would require

- 1595 standard PCs (3.2 GHz AMD Phenom II X4 955 CPU) [1]; or
- 1231 PlayStation 3 computers (Cell CPU with 6 usable SPEs and 1 PPE) [11]; or
- 534 GTX 295 graphics cards, as shown in this paper; or
- 308 XC3S5000-4FG676 FPGAs [13];

or any combination of the above.

2 The GTX 295 graphics card

The most impressive feature of GPUs is their theoretical floating-point performance. Each of the 480 ALUs on a GTX 295 can dispatch a single-precision floating-point multiplication (with a free addition) every cycle. There are also 120 "special-function units" that can each dispatch two single-precision floating-point multiplications every cycle, for a total of 894 billion floating-point multiplications per second.

The most useful GPU arithmetic instructions for the ECC2K-130 computation are 32-bit logical instructions (ANDs and XORs) rather than floating-point multiplications, but 596 billion 32-bit logical instructions per second are still much more impressive than (e.g.) the 28.8 billion 128-bit logical instructions per second performed by a typical 2.4 GHz Intel Core 2 CPU with 4 cores and 3 128-bit ALUs per core.

However, the GPUs also have many bottlenecks that make most applications run slower, often one or two orders of magnitude slower, than the theoretical throughput figures would suggest. The most troublesome bottlenecks are discussed in the remainder of this section and include a heavy divergence penalty, high instruction latency, low SRAM capacity, high DRAM latency, and relatively low DRAM throughput per ALU.

2.1. The dispatcher. The 8 ALUs in a GPU core are fed by a single *dispatcher*. The dispatcher cannot issue more than one new instruction to the ALUs every 4 cycles. The dispatcher can send this one instruction to a *warp* containing 32 separate *threads* of computation, applying the instruction to 32 pieces of data in parallel and keeping all 8 ALUs busy for all 4 cycles; but the dispatcher cannot direct some of the 32 threads to follow one instruction while the remaining threads follow another.

Branching is allowed, but if threads in a warp take different branches ("diverge") then the threads taking one branch will no longer operate in parallel with the threads in the other branch. For example, if 32 threads are split among all three branches in the code

```
if(x)
  if(y)
    aaa
  else
    bbb
else
  ccc
```

then the first threads will execute `aaa` while the other threads remain idle; next the second threads will execute `bbb` while the other threads remain idle; and finally the third threads will execute `ccc` while the other threads remain idle. The total time is the sum of the times taken by `aaa`, `bbb`, `ccc`, rather than the maximum of those times.

2.2. Instruction latency. Each thread follows its instructions strictly in order. NVIDIA does not document the exact pipeline structure but states that "the delays introduced by read-after-write dependencies can be ignored as soon as there are at least 192 active threads per multiprocessor to hide them" [16, Section 5.1.2.6]. If 8 ALUs are fully occupied with 192 threads then each thread runs every 24 cycles; evidently the latency of an arithmetic instruction is below 24 cycles.

One might think that a single warp of 32 threads can keep the 8 ALUs fully occupied, if the instructions in each thread are scheduled for 24-cycle arithmetic latency (i.e., if an arithmetic result is not used until 6 instructions later). However, our experiments showed that even with careful scheduling more threads are required, in particular if instructions access shared memory. This does not mean that one can achieve the best performance by running the maximum number of threads that fit into the core. Threads share several critical resources, as dis-

cussed below, so increasing the number of threads means reducing the resources available to each thread.

The ECC2K-130 computation puts extreme pressure on shared memory and registers, as discussed later in this paper; to minimize this pressure we ended up using just 128 threads. Some parts of the computation would benefit from more threads but the efficient Karatsuba multiplier described in Section 5 does not allow us to use more than 128 threads. We found experimentally that the penalties of using 128 threads were somewhat smaller when we rescheduled instructions to separate shared-memory accesses from each other.

2.3. SRAM: registers and shared memory. Each core has 16384 32-bit registers; these registers are divided among the threads running on the core. For example, if the core is running 256 threads, then each thread is assigned 64 registers. If the core is running 128 threads, then each thread is assigned 128 registers, although the high 64 registers are somewhat limited: the architecture does not allow a high register as the second operand of an instruction. The number of registers per thread is limited to 128 even if there are fewer than 128 threads.

The core also has 16384 bytes of *shared memory* that provide variable array indexing and communication between threads. This memory is split into 16 banks, each of which can dispatch one 32-bit read or write every two cycles. If the 16 threads in a half-warp read from the same location or 16 different banks then there are no bank conflicts, but if the threads all read from different locations of the same bank then they take 16 times as long.

Threads also have fast access to an 8192-byte *constant cache*. This cache can broadcast a 32-bit value from one location to every thread simultaneously, but it cannot read more than one location per cycle.

2.4. DRAM: global memory and local memory. The CPU makes data available to the GPU by copying it into the DRAM on the graphics card outside the GPU. The cores on the GPU can then load data from this *global memory* and store results in global memory to be retrieved by the CPU. Global memory is also a convenient temporary storage area for data that does not fit into shared memory. However, global memory is limited to a throughput of just one 32-bit load from each GPU core per cycle, with a latency of 400–600 cycles.

Each thread also has access to *local memory*. The name "local memory" might suggest that this storage is fast, but in fact it is another area of DRAM, as slow as global memory. Instructions accessing local memory automatically incorporate the thread ID into the address being accessed, effectively partitioning the local memory among threads without any extra address-calculation instructions.

There are no hardware caches for global memory and local memory. Programmers can, and must, set up their own schedules for copying data to and from global memory.

2.5. Choice of GPU. We decided to focus on the GTX 295 because it provides an excellent price-performance ratio. The price of a GTX 295 was only about $500 when we began this project. We built several $2000 PCs, each containing

two GTX 295s and a standard CPU, following the advice of Bernstein, Chen, Chen, Cheng, Hsiao, Lange, Lin, and Yang in [7] and [6]; each PC is (considering the extra cost of power and cooling) more than twice as expensive as a PC containing a CPU alone, but it gives us several times better performance.

Similar GPUs are also widely available in computing clusters, which typically use self-contained 1U rackmount Tesla S1060 units. Each S1060 contains 4 G200 (or G200b) GPUs and is microarchitecturally identical to a pair of slightly overclocked GTX 295s. The United States TeraGrid network included two such GPU clusters when we began this project, namely Lincoln (384 GPUs) and Longhorn (512 GPUs).

There are two other GPU architectures of note: AMD's Evergreen (R8xx) GPUs, as in the Radeon HD 5970, and NVIDIA's very new Fermi (GF1xx) line of GPUs, as in the GTX 480. All of these GPUs pose similar parallelization challenges, but there are many differences in the details. The current dominance of G200-based Tesla GPUs in computer clusters makes these GPUs the most attractive target for now. We focus on the GTX 295 throughout this paper.

3 The ECDLP and parallel Pollard rho

The standard method for solving the ECDLP in prime-order subgroups is Pollard's rho method [17]. For large instances of the ECDLP, one usually uses a parallelized rho method due to van Oorschot and Wiener [20]. This section briefly reviews the ECDLP and the parallel rho method.

The ECDLP is the following problem: Given an elliptic curve E over a finite field $\mathbb{F}_q$ and two points $P \in E(\mathbb{F}_q)$ and $Q \in \langle P \rangle$, find an integer k such that $Q = [k]P$.

Let ℓ be the order of P, and assume in the following that ℓ is prime. The parallel rho method is a client-server approach in which each client does the following:

1. Generate a pseudo-random starting point R_0 as a known linear combination of P and Q: $R_0 = a_0P + b_0Q$;
2. apply a pseudo-random iteration function f to obtain a sequence $R_{i+1} = f(R_i)$, where f is constructed to preserve knowledge about the linear combination of P and Q;
3. for each i, after computing R_i, check whether R_i belongs to an easy-to-recognize set D, the set of distinguished points, a subset of $\langle P \rangle$;
4. if at some moment a distinguished point R_i is reached, send (R_i, a_i, b_i) to the server and go to step 1.

The server receives all the incoming triples (R, a, b) and does the following:

1. Search the entries for a *collision*, i.e., two triples $(R, a, b), (R', a', b')$ with $R = R'$ and $b \not\equiv b' \pmod{\ell}$;
2. obtain the discrete logarithm of Q to the base P as $k = \frac{a'-a}{b-b'}$ modulo ℓ.

The expected running time of this parallel version of Pollard's rho algorithm is approximately $\sqrt{\pi\ell/2}$ calls to the iteration function f, assuming perfectly random behavior of f.

4 ECC2K-130 and the iteration function

The specific ECDLP addressed in this paper is given in the Certicom challenge list [12] as Challenge ECC2K-130. The given elliptic curve is the Koblitz curve $E : y^2 + xy = x^3 + 1$ over the finite field $\mathbb{F}_{2^{131}}$; the two given points P and Q have order ℓ, where ℓ is a 129-bit prime.

This section reviews the definition of distinguished points and the iteration function used in [1]. For a more detailed discussion, an analysis of communication costs, and a comparison to other possible implementation choices, the interested reader is referred to [1].

4.1. Definition of the iteration function. A point $R \in \langle P \rangle$ is *distinguished* if $\mathrm{HW}(x_R)$, the Hamming weight of the x-coordinate of R in normal-basis representation, is smaller than or equal to 34. The iteration function is defined as

$$R_{i+1} = f(R_i) = \sigma^j(R_i) + R_i,$$

where σ is the Frobenius endomorphism and

$$j = ((\mathrm{HW}(x_{R_i})/2) \bmod 8) + 3.$$

The restriction of σ to $\langle P \rangle$ corresponds to scalar multiplication with a particular easily computed scalar r. For an input $R_i = a_iP + b_iQ$, the output of f will be $R_{i+1} = (r^j a_i + a_i)P + (r^j b_i + b_i)Q$.

Each walk starts by picking a random 64-bit seed s which is then expanded deterministically into a linear combination $R_0 = a_0P + b_0Q$. To reduce bandwidth and storage requirements, the client does not report a distinguished triple (R, a, b) to the server but instead transmits only s and a 64-bit hash of R. On the occasions that a hash collision is found, the server recomputes the linear combinations in P and Q for $R = aP + bQ$ and $R' = a'P + b'Q$ from the corresponding seeds s and s'. This has the additional benefit that the client does not need to keep track of the coefficients a and b or counters for how often each Frobenius power is used. This speedup is particularly beneficial for highly parallel architectures such as GPUs, which otherwise would need a conditional addition to each counter in each step.

4.2. Computing the iteration function. The main task for each client is to repeatedly compute the function f defined above. Each iteration starts with a point $R = (x, y)$, computes a normal-basis Hamming weight $\mathrm{HW}(x)$ and thus $j = ((\mathrm{HW}(x)/2) \bmod 8) + 3$, computes $\sigma^j(R) = (x^{2^j}, y^{2^j})$, and adds (x, y) to (x^{2^j}, y^{2^j}) on E.

The addition on E requires 2 multiplications, one squaring, 6 additions, and 1 inversion in $\mathbb{F}_{2^{131}}$ in affine Weierstrass coordinates; see, e.g., [8]. Inversions are

significantly more expensive than multiplications but Montgomery's trick [15] reduces these expenses in a batch of inversions: it replaces m separate inversions with $3(m-1)$ multiplications and 1 inversion. For example, a 10-way batched inversion takes $3 \cdot (10-1) = 27$ multiplications and 1 inversion; each of the original inversions is thus replaced by 0.1 inversions and 2.7 multiplications.

The obvious way to compute x^{2^j} for $3 \leq j \leq 10$ is to first square 3 times, obtaining x^{2^3}, and then square repeatedly, at most 7 more times, until reaching x^{2^j}. However, this involves several expensive branches depending on the value of j. A branch-free strategy stated in [9] is to compute $r = x^{2^3}, s = r+b_0(r^2+r), t = s + b_1(s^4+s), u = t + b_2(t^{16}+t)$ where $j = 3 + b_0 + 2b_1 + 4b_2$, i.e., where $\mathrm{HW}(x) = 2b_0 + 4b_1 + 8b_2 + \cdots$. The computation $s = r + b_0(r^2+r)$ is a *conditional squaring*, computing $s = r = x^{2^3}$ if $b_0 = 0$ or $s = r^2 = x^{2^4}$ if $b_0 = 1$, i.e., $s = x^{2^{3+b_0}}$; similarly $t = x^{2^{3+b_0+2b_1}}$ and $u = x^{2^{3+b_0+2b_1+4b_2}} = x^{2^j}$.

In total, each iteration in a batch of 10 iterations takes 4.7 multiplications, 0.1 inversions, 1 squaring, 2 3-squarings (where an m-squaring means a sequence of m squarings), 2 conditional squarings, 2 conditional 2-squarings, 2 conditional 4-squarings, and 1 normal-basis Hamming-weight computation. As the batch size increases, the number of multiplications per iteration converges to 5 while the number of inversions per iteration converges to 0.

If each inversion is replaced by computing the $(2^{131}-2)$nd power using the sequence of multiplications and squarings shown in [9, Figure 5.3] then each iteration in a batch of 10 iterations takes 5.5 multiplications, 1.3 squarings, 2.6 m-squarings for various m, 2 conditional squarings, 2 conditional 2-squarings, 2 conditional 4-squarings, and 1 normal-basis Hamming-weight computation.

4.3. Bitslicing. The next step in optimizing multiplications in $\mathbb{F}_{2^{131}}$, squarings in $\mathbb{F}_{2^{131}}$, etc. is to decompose these arithmetic operations into the operations available on the target platform.

Modern microprocessors operate on words of many bits, say n bits. The fast XOR logical instruction reads two n-bit words, computes n bit XORs (additions in $\mathbb{F}_2$) in parallel, and produces an n-bit output word. Similar comments apply to other logical instructions: AND (multiplication in $\mathbb{F}_2$), OR, etc. For GPUs a typical instruction is often described as operating on a 32-bit word, so one might think that $n = 32$; but the same instruction is issued to many threads, effectively increasing n to 32 times the number of threads.

If one n-bit word can be viewed as n independent elements of $\mathbb{F}_2$, then k n-bit words can be viewed as n independent elements of the vector space $\mathbb{F}_2^k$, for example representing n independent elements of the field $\mathbb{F}_{2^k}$ on a suitable basis. If an operation in $\mathbb{F}_{2^k}$ can be carried out with $T(k)$ additions and multiplications in $\mathbb{F}_2$ then the same operation can be carried out in parallel on n elements in $\mathbb{F}_{2^k}$ with $T(k)$ XOR instructions and AND instructions.

This technique is called *bitslicing*. It was introduced by Biham [10] in an implementation of the Data Encryption Standard. The speedups that can be achieved from bitslicing binary-field arithmetic on a modern CPU were demonstrated by Bernstein in [4]. We decided to use bitslicing on a GPU as it allows very efficient use of logical operations in implementing binary-field operations.

The final cost of our optimized bitsliced multiplier turned out to be smaller than a lower bound for the cost of a traditional non-bitsliced table-based multiplier.

The most obvious challenge in efficient bitsliced arithmetic is to optimize $T(k)$, the number of bit operations required for an operation in $\mathbb{F}_{2^k}$. However, the requirement to work on nk bits in parallel creates additional challenges that are particularly troublesome for GPUs: an n-bit GPU instruction is efficient only if n is very large (32 bits times $\geq$128 threads), but if n is very large then the cost of accessing nk bits becomes a bottleneck.

The rest of this section reviews the techniques that produce the smallest number of bit operations known for the ECC2K-130 iteration function. The rest of this paper explains how we made the iteration function run quickly on the GPU: Section 5 analyzes 131-bit polynomial multiplication, and Section 6 analyzes the complete iteration function.

4.4. Choice of basis. There is no irreducible trinomial of degree 131 in the polynomial ring $\mathbb{F}_2[z]$. The standard representation of $\mathbb{F}_{2^{131}}$ is as $\mathbb{F}_2[z]$ modulo an irreducible pentanomial, such as $z^{131} + z^{13} + z^2 + z + 1$. Multiplication on the basis $1, z, z^2, \ldots, z^{130}$ of $\mathbb{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$ involves one 131-bit polynomial multiplication and just 455 extra bit operations.

However, the ECC2K-130 iteration involves not only several multiplications but also also many squarings, including m-squarings, making normal bases particularly attractive. A squaring in normal basis corresponds to a simple cyclic shift of the coefficients. An m-squaring corresponds to a cyclic shift by m positions. Working with x in normal basis also removes the need to convert x from pentanomial basis to normal basis.

The main problem with normal bases is the cost of multiplication. However, $\mathbb{F}_{2^{131}}$ has a type-II optimal normal basis, and Shokrollahi's type-II multiplier [18] (see also [14]) has a surprisingly small overhead above the cost of polynomial multiplication. Bernstein and Lange [9] recently reduced this overhead further and showed that combining the optimal normal basis with an *optimal polynomial basis* achieves a significantly lower cost than a pentanomial basis for the ECC2K-130 iteration.

Our final GPU implementation uses this multiplier, as described in Section 6. Here we give a short summary of the mathematics necessary for understanding the implementation. For full details see [9].

Field elements such as x and y are represented on the permuted optimal normal basis $\left(\zeta+\zeta^{-1}, \zeta^2+\zeta^{-2}, \zeta^3+\zeta^{-3}, \ldots, \zeta^{131}+\zeta^{-131}\right)$ of $\mathbb{F}_{2^{131}}$, where $\zeta \in \mathbb{F}_{2^{262}}$ is a primitive 263rd root of 1. Squaring is a permutation of coefficients in this basis, although no longer a simple cyclic shift. Multiplication has four steps:

- Convert each input to the *optimal polynomial basis* $\left(\zeta+\zeta^{-1}, (\zeta+\zeta^{-1})^2, (\zeta+\zeta^{-1})^3, \ldots, (\zeta+\zeta^{-1})^{131}\right)$. A streamlined recursive conversion takes just 325 bit operations.
- Apply 131-bit polynomial multiplication, obtaining a product of the form $a_2(\zeta+\zeta^{-1})^2 + a_3(\zeta+\zeta^{-1})^3 + \cdots + a_{131}(\zeta+\zeta^{-1})^{131} + \cdots + a_{262}(\zeta+\zeta^{-1})^{262}$.

- Apply a double-size inverse conversion, obtaining the same product as a linear combination of $\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^{262} + \zeta^{-262}$. This transformation takes just 779 bit operations.
- Reduce the product to the original basis $\left(\zeta+\zeta^{-1}, \zeta^2+\zeta^{-2}, \zeta^3+\zeta^{-3}, \ldots, \zeta^{131}+\zeta^{-131}\right)$ using the identities $\zeta^{262} + \zeta^{-262} = \zeta + \zeta^{-1}$, $\zeta^{261} + \zeta^{-261} = \zeta^2 + \zeta^{-2}$, etc. This transformation takes just 130 bit operations.

We use the name `multprep` for the initial conversion of an input from permuted optimal normal basis to optimal polynomial basis. We use the name `ppn` for the remaining steps, taking two inputs in optimal polynomial basis and delivering an output in normal basis.

Sometimes the output of a multiplication is used again as input to another multiplication, and is not used for any squarings. In such cases Bernstein and Lange keep the lower half of the polynomial product in polynomial-basis representation and use the conversion routine only to compute the polynomial reduction, ending up in polynomial-basis representation and skipping a subsequent `multprep`. We use the name `ppp` for this operation, taking two inputs in optimal polynomial basis and delivering an output in optimal polynomial basis.

The costs of either type of field multiplication, `ppn` or `ppp`, are dominated by the costs of the 131-bit polynomial multiplication. The next section optimizes polynomial multiplication for the GPU, and Section 6 optimizes the entire iteration function for the GPU.

5 Polynomial multiplication on the GPU

With optimal polynomial bases (see Section 4.4), each iteration involves slightly more than five 131-bit polynomial multiplications and only about 10000 extra bit operations. We are not aware of any 131-bit polynomial multiplier using fewer than 11000 bit operations; in particular, Bernstein's multiplier [4, Section 2] uses 11961 bit operations.

These figures show that polynomial multiplication consumes more than 80% of the bit operations in each iteration. We therefore placed a high priority on making multiplication run quickly on the GPU, preferably below 200 cycles in a single core. This section explains how we achieved this goal.

5.1. The importance of avoiding DRAM. We began by exploring an embarrassingly vectorized approach: T threads in a core work on $32T$ independent multiplication problems in bitsliced form. The $32T \times 2$ inputs are stored as 262 vectors of $32T$ bits, and the $32T$ outputs are stored as 261 vectors of $32T$ bits.

The main difficulty with this approach is that, even if the outputs are perfectly overlapped with the inputs, even if no additional storage is required, the inputs cannot fit into SRAM. For $T = 128$ the inputs consume 134144 bytes, while shared memory and registers together have only 81920 bytes. Reducing T to 64 (and tolerating a limit of 50% ALU utilization) would fit the inputs into 67072 bytes, but would also make half of the registers inaccessible (since each

thread can access at most 128 registers), reducing the total capacity of shared memory and registers to 49152 bytes.

There is more than enough space in DRAM, even with very large T, but DRAM throughput then becomes a serious bottleneck. A single pass through the input vectors, followed by a single pass through the output vectors, keeps the DRAM occupied for $523T$ cycles (i.e., more than 16 cycles per multiplication), and any low-memory multiplication algorithm requires many such passes.

We implemented and optimized several multiplication algorithms and complete iteration functions using this approach, but our best result using this approach was only 26 million iterations per second on a GTX 295. The remainder of this section describes a faster approach.

5.2. How to fit into shared memory. The SIMD programming model of GPUs highly relies on the exploitation of data-level parallelism. However, data-level parallelism does not require having each thread work on a completely independent computation: parallelism is also available within computations. For example, the addition of two 32-way-bitsliced field elements is nothing but a sequence of 131 32-bit xor operations; it naturally contains 131-way data-level parallelism. Similarly, there are many ways to break 131-bit binary-polynomial multiplication into several smaller-degree polynomial multiplications that can be carried out in parallel.

Threads cannot communicate through registers, so having several threads cooperate on a single computation requires the active data for the computation to fit into shared memory. On the other hand, registers have more space than shared memory; during multiplication we use some registers as spill locations for data not involved in the multiplication, reversing the traditional direction of data spilling from registers to memory.

Our final software carries out 128 independent 131-bit multiplications (i.e., four 32-way bitsliced 131-bit multiplications) inside shared memory and registers, with no DRAM access. This means that each multiplication has to fit within 1024 bits of shared memory. This would not have been a problem for schoolbook multiplication, but it was a rather tight fit for the fast Karatsuba-type multiplication algorithm that we use (see below); more simultaneous multiplications would have meant compromises in the multiplication algorithm.

We decided to use 128 threads. This means that 32 threads are cooperating on each of the four 32-way bitsliced 131-bit multiplications. We expected, and ran experiments to confirm, that this would be enough threads to hide most latencies in the most time-consuming parts of the iteration function, particularly multiplication. Our 131-bit multiplication algorithm allows close to 32-way parallelization, as discussed below, although the parallelization is not perfect.

We would have had fewer latency problems from 192 or 256 threads, but the overall benefit is small and overwhelmed by increased parallelization requirements within each multiplication. In the opposite direction, we could have reduced the parallelization requirements by running 96 or 64 threads, but below 128 threads the GPU performance drops almost linearly.

5.3. Vectorized 128-bit multiplication. Our main task is now to multiply 131-bit polynomials, at each step using using 32 parallel bit operations to the maximum extent possible. We repeat the resulting algorithm on 128 independent inputs to obtain what the code actually does with 128 threads: namely, 128 separate multiplications of 131-bit polynomials, stored in bitsliced form as $4 \cdot 131$ 32-bit words, using 128 parallel 32-bit operations to the maximum extent possible.

First consider the simpler task of multiplying 128-bit polynomials. This can efficiently be performed by applying three levels of Karatsuba expansion. Each level uses $2n$ xor instructions to expand a $2n$-bit multiplication into three n-bit multiplications, and then $5n - 3$ xor instructions to collect the results (with Bernstein's "refined Karatsuba" from [3, page 7] and [4]).

Three levels of Karatsuba result in 27 times 16-bit polynomial multiplications. The inputs to these multiplications occupy a total of 864 bits, consuming most but not all of the 1024 bits of shared memory available to each 131-bit multiplication. The code from [5] for a 16-bit polynomial multiplication can be scheduled to fit into 67 registers. It is applied to the 27 multiplications in parallel, leaving 5 threads idle out of 32. In total $27 \cdot 4 = 108$ 16-bit polynomial multiplications on 32-bit words are carried out by 108 threads in this subroutine leaving 20 threads idle. Each thread executes 413 instructions (350 bit operations and 63 load/store instructions).

The initial expansion can be parallelized trivially. Operations on all three levels can be joined and performed together on blocks of 16 bits per operand using 8 loads, 19 xor instructions, and 27 stores per thread.

Karatsuba collection is more work: On the highest level (level 3), each block of 3 times 32-bit results (with leading coefficient zero) is combined into a 64-bit intermediate result for level 2. This takes 5 loads (2 of these conditional), 3 xor operations and 3 stores per thread on each of the 9 blocks. Level 2 operates on blocks of 3 64-bit intermediate results leading to 3 128-bit blocks of intermediate results for level 1. This needs 6 loads and 5 xor operations for each of the 3 blocks. The 3 blocks of intermediate results of this step do not need to be written to shared memory and remain in registers for the following final step on level 1. Level 1 combines the remaining three blocks of 128 bits to the final 256-bit result by 12 xor operations per thread.

5.4. Vectorized 131-bit multiplication. To multiply 131-bit polynomials, the inputs are split into a 128-bit low part and a 3-bit high part. The 128-bit multiplications of the two low parts are handled by a 128-bit multiplier as described above. The 3×3-bit product of the high parts and the two 3×128-bit mixed products can be handled in a straightforward way: the 3×3-bit multiplication can be carried out almost for free by an otherwise idle 16-bit multiplication thread; the 3×128-bit multiplications can be implemented straightforwardly by schoolbook multiplication.

However, one can save some of the additions to obtain the final result, and further streamline the code, by distributing the computations as follows. The 5 most significant bits of the final result only depend on the 5 most significant bits

of each input. Thus they can be obtained by computing the product of these bits (using the 16-bit multiplier) and by cutting off the 4 least significant bits of the 9 resulting bits. Now the 2 most significant bits of the results from the 3×128-bit multiplications do not need to be computed and summed up anymore; the 128 least significant bits of the 3×128-bit multiplications can be obtained each by 6 loads for the 3 highest bits of each input, $3 \cdot 128$ combined load-and instructions per input, and $2 \cdot 128$ xor instructions (some of them masked for the 2 least significant bits).

Overall the multiplier uses 13087 bit operations, and about 40% of the ALU cycles are spent on these bit operations rather than on loads, stores, address calculations, and other overhead. An extra factor of about 1.1 is lost from 32-way parallelization, since the 32 threads are not always all active. For comparison, the Toom-type techniques from [5] use only 11961 bit operations, saving about 10%, but appear to be more difficult to parallelize.

6 ECC2K-130 iterations on the GPU

Recall that polynomial multiplication, the topic of the previous section, consumes more than 80% of the bit operations in the ECC2K-130 computation. This does not mean that the 20% overhead can be ignored! Imagine, for example, that the polynomial-multiplication code is carrying out useful bit operations in 40% of its cycles, while the remaining code fits much less smoothly into the GPU and is carrying out useful bit operations in only 5% of its cycles. The total time would then be *triple* the polynomial-multiplication time.

This section discusses several aspects of the overhead in the ECC2K-130 computation. Our main goal, as in the previous section, is to identify 32-way parallelism in the bit operations inside each 131-bit operation. This is often more challenging for the "overhead" operations than it is for multiplication, and in some cases we change algorithms to improve parallelism. All of these operations work entirely in shared memory, except where global memory is explicitly mentioned below.

6.1. Basis conversion (`multprep`). As explained in Section 4.4 we keep most elements of $\mathbb{F}_{2^{131}}$ in (permuted) normal basis. Before those elements are multiplied we convert them from normal basis to polynomial basis.

Consider an element a of $\mathbb{F}_{2^{131}}$ in (permuted) normal basis:

$$a = a_0(\zeta + \zeta^{-1}) + a_1(\zeta^2 + \zeta^{-2}) + \cdots + a_{130}(\zeta^{131} + \zeta^{-131}).$$

On the first two levels of the basis conversion algorithm the following sequence of operations is executed on bits $a_0, a_{62}, a_{64}, a_{126}$:

$$\begin{aligned} a_{62} &\leftarrow a_{62} + a_{64}, && \text{then} \\ a_0 &\leftarrow a_0 + a_{126}, && \text{then} \\ a_{64} &\leftarrow a_{64} + a_{126}, && \text{then} \\ a_0 &\leftarrow a_0 + a_{62}. \end{aligned}$$

Meanwhile the same operations are performed on bits $a_1, a_{61}, a_{65}, a_{125}$; on bits $a_2, a_{60}, a_{66}, a_{124}$; and so on through $a_{30}, a_{32}, a_{94}, a_{96}$. We assign these 31 groups of bits to 32 threads, keeping almost all of the threads busy.

Merging levels 2 and 3 and levels 4 and 5 works in the same way. This assignment keeps 24 out of 32 threads busy on levels 2 and 3, and 16 out of 32 threads busy on levels 4 and 5. This assignment of operations to threads also avoids almost all memory-bank conflicts (see Section 2).

6.2. Multiplication with reduction (`ppp` and `ppn`). Recall that a `ppp` operation produces a product in polynomial basis, suitable for input to a subsequent multiplication. A `ppn` operation produces a product in normal basis, suitable for input to a squaring.

The main work in `ppn`, beyond polynomial multiplication, is a conversion of the product from polynomial basis to normal basis. This conversion is almost identical to `multprep` above, except that it is double-size and in reverse order. The main work in `ppp` is a more complicated double-size conversion, with similar parallelization.

6.3. Squaring and m-squaring (`sq`, `msq` and `sqseq`). Squaring (subroutine `sq`) and m-squaring (subroutine `msq`) are simply permutations in normal basis, costing 0 bit operations, but this does not mean that they cost 0 cycles.

The obvious method for 32 threads to permute 131 bits is for them to pick up the first 32 bits, store them in the correct locations, pick up the next 32 bits, store them in the correct locations, etc.; each thread performs 5 loads and 5 stores, with most of the threads idle for the final load and store. The addresses determined by the permutation for different m-squarings can be kept in constant memory. However, this approach triggers two GPU bottlenecks.

The first bottleneck is shared-memory bank throughput. Recall from Section 2 that threads in the same half-warp cannot simultaneously store values to the same memory bank. To almost completely eliminate this bottleneck we wrote a greedy search tool that decides on a good order to pick up 131 bits, trying to avoid all memory bank conflicts for both the loads and the stores. For almost all values of m, including the most frequently used ones, this tool found a conflict-free assignment. For two values of m the assignment involves a few bank conflicts, but these values are used only in inversion, not in the main loop.

The second bottleneck is constant-cache throughput. If thread i loads from a constant array at position i then the constant cache serves only one thread per cycle. To eliminate this bottleneck we move these loads out of the main loop and dedicate 10 registers per thread to hold 20 load and 20 store positions for the 4 most-often used m-squarings, packing 4 1-byte positions in one 32-bit register. Unpacking the positions costs just one shift and one mask instruction for the two middle bytes, a mask instruction for the low byte, and a shift instruction for the high byte.

6.4. Hamming-weight computation (`hamming` and `below`). The Hamming-weight computation (the `hamming` subroutine) receives a bitsliced input and computes a bitsliced output. More specifically, the first 8 bits of the input value x

are overwritten with bits $h_0, \ldots, h_7$ such that the Hamming weight of the input value x is $\sum_{i=0}^{7} h_i 2^i$. During the parallel computation of these 8 bits also other bits of x are overwritten. The basic building block for the parallel computation is a full adder, which has three input bits b_1, b_2, b_3 and uses 5 bit operations to compute 2 output bits c_0, c_1 such that $b_1 + b_2 + b_3 = c_1 2 + c_0$. When the full adder overwrites one of the input bits with c_1 this bit gets a weight of 2. If three such bits with a weight of 2 are input to a full adder, one of the output bits will have a weight of 4. More generally: If three bits with a weight of 2^i enter a full adder, one output bit will have a weight of 2^i, the other one a weight of 2^{i+1}. At the beginning of the computation all 131 bits have a weight of 2^0.

Because there are many input bits, it is easy to keep many threads active in parallel. In the first addition round 32 threads perform 32 independent full-adder operations, 96 bits with weight 2^0 are transformed into 32 bits with weight 2^0 and 32 bits with weight 2^1. This leaves $131 - 96 + 32 = 67$ bits of weight 2^0 and 32 bits of weight 2^1.

In the second round, 22 threads pick up 66 bits of weight 2^0 and produce 22 bits of weight 2^0 and 22 bits of weight 2^1. At the same time 10 other threads pick up 30 bits of weight 2^1 and produce 10 bits of weight 2^1 and 10 bits of weight 2^2. This leaves $67 - 66 + 22 = 23$ bits of weight 2^0, $32 - 30 + 22 + 10 = 34$ bits of weight 2^1, and 10 bits of weight 2^2.

In the third round 7 threads perform full-adder operations on 21 input bits with weight 2^0, 11 threads perform full-adder operations on 33 input bits of weight 2^1, and 3 threads perform full-adder operations on 9 input bits of weight 2^2.

This parallel computation needs 13 rounds to compute the bits $h_0, \ldots, h_7$, i.e. 8 bits with weight $2^0, \ldots, 2^7$. The implementation actually uses a somewhat less parallel approach with 21 rounds, two of these rounds being half-adder operations which receive only 2 input bits and take only 2 bit operations. This has the benefit of simplifying computation of the input positions as a function of the thread ID.

Once the Hamming weight is computed the subroutine `below` tests whether the weight is below 34, i.e., whether the point is distinguished.

6.5. Kernel-launch overhead, register spills, etc. GPU code is organized into *kernels* called from the CPU. Calling (*launching*) a kernel takes several microseconds on top of any time needed to copy data between global memory and the CPU. To eliminate these costs we run a single kernel for several seconds. The kernel consists of a loop around a complete iteration; it performs the iteration repeatedly without contacting the CPU. Any distinguished points are masked out of subsequent updates; distinguished points are rare, so negligible time is lost computing unused updates.

We stream a batch of iterations in a simple way between global memory and shared memory; this involves a small number of global-memory copies in each iteration. See Section 4.2 for further discussion of batching. We avoid spilling any additional data to DRAM; in particular, we avoid all use of local memory.

All of this sounds straightforward but in fact required completely redesigning our programming environment. To explain this we briefly review NVIDIA's standard programming environment and the problems that we encountered with it.

Non-graphical applications for NVIDIA GPUs are usually programmed in CUDA, a C-like language designed by NVIDIA. NVIDIA's `nvcc` compiler translates a `*.cu` CUDA file into a `*.ptx` file in a somewhat machine-independent language called PTX. NVIDIA's `ptxas` compiler translates this `*.ptx` file into a machine-specific binary `*.cubin` file. The `*.cubin` file is loaded onto the GPU and run.

NVIDIA's register allocators were designed to handle small kernels consisting of hundreds of instructions; their memory-use scaling appears to be quadratic with the kernel size, and their time scaling appears to be even worse. For medium-size kernels we found NVIDIA's compilers intolerably slow; even worse, the resulting code involved frequent spills to local memory, dropping performance by an order of magnitude. For larger kernels the compilers ran out of memory and crashed.

We experimented with writing code in the PTX language, but this language still requires compilation by `ptxas`; even though `ptxas` is labelled as an "assembler" it turns out to be the culprit in NVIDIA's register-allocation problems. To control register allocation we eventually resorted to the reverse-engineered assembler `cudasm` by van der Laan [19]. We fixed some bugs in `cudasm` and, to improve usability, extended Bernstein's `qhasm` programming language [2] to support `cudasm`. We used `qhasm` on top of `cudasm` to implement the whole kernel: more than 90000 instructions after macro processing for batch size 32. We then designed our own assembly-level function-call convention and merged large stretches of instructions into functions, reducing instruction-cache misses and making the code size independent of the batch size. This allowed us to increase the batch size to 128, making the per-iteration inversion cost negligible.

6.6. Overall results. Table 1 reports timings of all major building blocks in our software. For example, in the `multprep` row of the table, the cycles/iteration column is 52, indicating that `multprep` was responsible for 52 cycles in each iteration. The calls/iteration column is 4.12, indicating that an average iteration involved 4.12 calls to `multprep`; in fact, a batch of 128 iterations involved 527 calls to `multprep`. The cycles/call column is 12.

We collected these numbers by performing the following experiment. On a typical pass through the main loop (specifically, the 10000th pass), inside each thread, we checked the GTX 295's hardware cycle half-counter before and after each use of `multprep`, and tallied the cycles spent inside `multprep`. We repeated this experiment 20 times, with 128 threads in each experiment, yielding a total of $20 \cdot 128 = 2560$ cycle counts. We divided the cycle counts by 16384 because each pass through the main loop performs $16384 = 128 \cdot 128$ iterations: our implementation always handles 128 iterations in parallel, and on top of this we chose a batch size of 128 as mentioned above. The average of these cycle counts was 52. Table 1 also reports standard deviations: e.g., 52 ± 0.21.

After measuring one operation we removed the cycle counters and placed them around each occurrence of another operation. Only one operation is measured at a time. Cycle counting is not free, so the sum of times measured for the individual operations is slightly more than the time measured for the entire main loop. The operations shown in Table 1 are as follows:

- `ppp` multiplies two field elements in polynomial basis, producing output in polynomial basis.
- `ppn` multiplies two field elements in polynomial basis, producing output in normal basis.
- `multprep` converts from normal basis to polynomial basis.
- `sqseq` is the sequence of squarings used to compute x^{2^j}: a 3-squaring, a conditional squaring, a conditional 2-squaring, and a conditional 4-squaring.
- `msq` is an m-squaring for any $m \in \{1, 2, 3, 4, 8, 16, 32, 65\}$.
- `add` adds two field elements.
- `cadd` is conditional field addition, i.e., addition masked by an extra bit.
- `hamming` computes Hamming weight.
- `below` checks for a distinguished point.
- `readfen` copies a field element from global memory to shared memory.
- `writefen` copies a field element the other way.
- `readbit` copies a bit from global memory to shared memory.
- `writebit` copies a bit the other way.
- `copy` copies a field element within shared memory.

There are some additional rows in the table showing total time spent in 131-bit polynomial multiplication; total time spent in the `27xmult16` subroutine; total time spent on global-memory access; and total time spent in inversion.

Each of the instructions in our software handles 128 iterations, but is also followed by 128 threads, keeping the GPU core busy for at least 16 cycles. The number of cycles spent per iteration is therefore at least 0.125 times the number of instructions. For example, `27xmult16` occurs 5.04 times per iteration and involves 413 instructions, accounting for $0.125 \cdot 5.04 \cdot 413 \approx 260$ cycles in each iteration. The actual number of cycles spent on `27xmult16` is about 25% higher than this; the gap is explained in part by instruction-cache misses and in part by the delays for complex instructions discussed in Section 2.

The complete kernel uses 1164 cycles per iteration on average on a single core on a GTX 295 graphics card. Therefore we achieve 63 million iterations per second on a single card (60 cores, 1.242 GHz). The whole ECC2K-130 computation would be finished in two years (on average; the rho method is probabilistic) using 534 GTX 295 graphics cards.

References

1. Daniel V. Bailey, Lejla Batina, Daniel J. Bernstein, Peter Birkner, Joppe W. Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier van Damme, Giacomo de Meulenaer, Luis Julian Dominguez Perez, Junfeng Fan, Tim Güneysu, Frank Gurkaynak,

operation	calls/iteration	cycles/call	cycles/iteration	
complete iteration			1164.43 ± 12.05	100.00%
`ppp`	2.9766	158.08 ± 0.23	470.55 ± 0.68	40.41%
`ppn`	2.0625	159.54 ± 1.64	329.05 ± 3.38	28.26%
`sqseq`	2.0000	44.99 ± 0.83	89.97 ± 1.67	7.73%
`readfen`	7.9844	9.03 ± 0.10	72.12 ± 0.82	6.19%
`multprep`	4.1172	12.63 ± 0.05	52.00 ± 0.21	4.47%
`writefen`	5.9844	7.91 ± 0.27	47.35 ± 1.63	4.07%
`hamming`	1.0000	41.60 ± 0.11	41.60 ± 0.11	3.57%
`add`	7.0000	4.01 ± 0.06	28.04 ± 0.40	2.41%
`readbit`	5.0000	2.90 ± 0.84	14.52 ± 4.22	1.25%
`cadd`	2.0000	6.81 ± 0.77	13.62 ± 1.53	1.17%
`below`	1.0000	11.26 ± 0.16	11.26 ± 0.16	0.97%
`msq`	1.0703	9.60 ± 0.12	10.27 ± 0.13	0.88%
`copy`	2.0000	3.73 ± 0.87	7.45 ± 1.74	0.64%
`writebit`	4.0000	1.25 ± 0.02	5.00 ± 0.09	0.43%
131-bit poly mult	5.0391	119.11 ± 0.13	600.21 ± 0.65	51.55%
`27xmult16`	5.0391	65.02 ± 0.19	327.66 ± 0.98	28.14%
DRAM access			139.01 ± 5.73	11.94%
inversion			13.74 ± 0.16	1.18%

Table 1. Timings of major building blocks within an iteration.

Thorsten Kleinjung, Tanja Lange, Nele Mentens, Ruben Niederhagen, Christof Paar, Francesco Regazzoni, Peter Schwabe, Leif Uhsadel, Anthony Van Herrewege, and Bo-Yin Yang. Breaking ECC2K-130. Cryptology ePrint Archive, Report 2009/541, 2009. `http://eprint.iacr.org/2009/541`.

2. Daniel J. Bernstein. qhasm: tools to help write high-speed software. `http://cr.yp.to/qhasm.html`.
3. Daniel J. Bernstein. Fast multiplication, 2000. `http://cr.yp.to/talks.html#2000.08.14`.
4. Daniel J. Bernstein. Batch binary Edwards. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 317–336. Springer, 2009. Document ID: 4d7766189e82c1381774dc840d05267b, `http://cr.yp.to/papers.html#bbe`.
5. Daniel J. Bernstein. Minimum number of bit operations for multiplication, 2009. `http://binary.cr.yp.to/m.html` (accessed 2009-12-07).
6. Daniel J. Bernstein, Hsieh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Chin Lin, and Bo-Yin Yang. The billion-mulmod-per-second PC. In *Workshop Record of SHARCS 2009: Special-purpose Hardware for Attacking Cryptographic Systems*, pages 131–144, 2009. `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`.
7. Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, and Bo-Yin Yang. ECM on graphics cards. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 483–501. Springer, 2009. Document ID: 6904068c52463d70486c9c68ba045839, `http://eprint.iacr.org/2008/480/`.

8. Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. http://www.hyperelliptic.org/EFD/. (accessed 2010-09-25).
9. Daniel J. Bernstein and Tanja Lange. Type-II optimal polynomial bases. In M. Anwar Hasan and Tor Helleseth, editors, *Arithmetic of Finite Fields*, volume 6087 of *Lecture Notes in Computer Science*, pages 41–61, 2010. Document ID: 90995f3542ee40458366015df5f2b9de, http://binary.cr.yp.to/opb-20100209.pdf.
10. Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.
11. Joppe W. Bos, Thorsten Kleinjung, Ruben Niederhagen, and Peter Schwabe. ECC2K-130 on Cell CPUs. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology – AFRICACRYPT 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 225–242. Springer Berlin / Heidelberg, 2010. Document ID: bad46a78a56fdc3a44fcf725175fd253, http://eprint.iacr.org/2010/077/.
12. Certicom. Certicom ECC challenge. http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf, 1997.
13. Junfeng Fan, Daniel V. Bailey, Lejla Batina, Tim Güneysu, Christof Paar, and Ingrid Verbauwhede. Breaking elliptic curves cryptosystems using reconfigurable hardware. 20th International Conference on Field Programmable Logic and Applications (FPL 2010), Milano, Italy, August 31st–September 2nd, 2010.
14. Joachim von zur Gathen, Amin Shokrollahi, and Jamshid Shokrollahi. Efficient multiplication using type 2 optimal normal bases. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields*, volume 4547 of *Lecture Notes in Computer Science*, pages 55–68, 2007.
15. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48:243–264, 1987.
16. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide Version 2.2*, 2009. developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf.
17. John M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32:918–924, 1978.
18. Jamshid Shokrollahi. *Efficient implementation of elliptic curve cryptography on FPGAs*. Ph.D. thesis, Rheinische Friedrich-Wilhelms Universität, 2007. http://nbn-resolving.de/urn:nbn:de:hbz:5N-09601.
19. Wladimir J. van der Laan. Cubin utilities, 2007. http://wiki.github.com/laanwj/decuda/.
20. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.