ECC2K-130 on Cell CPUs

Joppe W. Bos[1,⋆], Thorsten Kleinjung[1,⋆], Ruben Niederhagen[2,3,⋆], and Peter Schwabe[3,⋆]

[1] Laboratory for Cryptologic Algorithms
EPFL, Station 14, CH-1015 Lausanne, Switzerland
{joppe.bos,thorsten.kleinjung}@epfl.ch

[2] Department of Electrical Engineering
National Taiwan University, 1 Section 4 Roosevelt Road, Taipei 106-70, Taiwan
ruben@polycephaly.org

[3] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
peter@cryptojedi.org

Abstract. This paper describes an implementation of Pollard's rho algorithm to compute the elliptic curve discrete logarithm for the Synergistic Processor Elements of the Cell Broadband Engine Architecture. Our implementation targets the elliptic curve discrete logarithm problem defined in the Certicom ECC2K-130 challenge. We compare a bitsliced implementation to a non-bitsliced implementation and describe several optimization techniques for both approaches. In particular, we address the question whether normal-basis or polynomial-basis representation of field elements leads to better performance. We show that using our software the ECC2K-130 challenge can be solved in one year using the Synergistic Processor Units of less than 2700 Sony Playstation 3 gaming consoles.

Keywords: Cell Broadband Engine Architecture, elliptic curve discrete logarithm problem, binary-field arithmetic, parallel Pollard rho.

1 Introduction

How long does it take to solve the elliptic curve discrete logarithm problem (ECDLP) on a given elliptic curve using given hardware? This question was addressed recently for the Koblitz curve defined in the Certicom challenge ECC2K-130 for a variety of hardware platforms [2]. This paper zooms into Section 6 of [2] and describes the implementation of the parallel Pollard rho algorithm [17] for the Synergistic Processor Elements of the Cell Broadband Engine Architecture (CBEA) in detail. We discuss our choice to use the technique of bitslicing [7] to accelerate the underlying binary-field arithmetic operations by comparing a bitsliced to a non-bitsliced implementation.

⋆ This work has been supported in part by the European Commission through the ICT Programme under Contract ICT–2007–216499 CACE, and through the ICT Programme under Contract ICT-2007-216646 ECRYPT II. Permanent ID of this document: bad46a78a56fdc3a44fcf725175fd253. Date: February 28, 2010.

Many optimization techniques for the non-bitsliced version do not require independent parallel computations (batching) and are therefore not only relevant in the context of cryptanalytical applications but can also be used to accelerate cryptographic schemes in practice.

To the best of our knowledge this is the first work to describe an implementation of high-speed binary-field arithmetic for the CBEA. We plan to put all code described in this paper into the public domain to maximize reusability of our results.

Organization of the paper. In Section 2 we describe the features of the CBEA which are relevant to this paper. To make this paper self contained we briefly recall the parallel version of Pollard's rho algorithm and summarize the choice of the iteration function in Section 3. Section 4 discusses different approaches to a high-speed implementation of the iteration function on the CBEA. In Sections 5 and 6 we describe the non-bitsliced and the bitsliced implementation, respectively. We summarize the results and conclude the paper in Section 7.

2 A Brief Description of the Cell Processor

The Cell Broadband Engine Architecture [12] was jointly developed by Sony, Toshiba and IBM. Currently, there are two implementations of this architecture, the Cell Broadband Engine (Cell/B.E.) and the PowerXCell 8i. The PowerXCell 8i is a derivative of the Cell/B.E. and offers enhanced double-precision floating-point capabilities and a different memory interface. Both implementations consist of a central *Power Processor Element* (PPE), based on the Power 5 architecture and 8 *Synergistic Processor Elements* (SPEs) which are optimized for high-throughput vector instructions. All units are linked by a high-bandwidth (204 GB/s) ring bus.

The Cell/B.E. can be found in the IBM blade servers of the QS20 and QS21 series, in the Sony Playstation 3, and several acceleration cards like the Cell Accelerator Board from Mercury Computer Systems. The PowerXCell 8i can be found in the IBM QS22 servers. Note that the Playstation 3 only makes 6 SPEs available to the programmer.

The code described in this paper runs on the SPEs directly and does not interact with the PPE or other SPEs during core computation. We do not take advantage of the extended capabilities of the PowerXCell 8i. In the remainder of this section we will describe only those features of the SPE which are of interest for our implementation and are common to both the Cell/B.E. and the PowerXCell 8i. Therefore we may address the Cell/B.E. and the PowerXCell 8i jointly as the *Cell processor* or *Cell CPU*. Further information on the current implementations of the Cell Broadband Engine Architecture can be found in [14].

Each SPE consists of a *Synergistic Processor Unit* (SPU) as its computation unit and a *Memory Flow Controller* (MFC) which grants access to the ring bus and therefore in particular to main memory.

2.1 SPU – Architecture and Instruction Set

The SPU is composed of three parts: The *Synergistic Execution Unit* (SXU) is the computational core of each SPE. It is fed with data either by the SPU *Register File Unit* (RFU) or by the *Local Storage* (LS) that also feeds the instructions into the SXU.

The RFU contains 128 general-purpose registers with a width of 128 bits each. The SXU has fast and direct access to the LS but the LS is limited to only 256 KB. The SXU does not have transparent access to main memory; all data must be transferred from main memory to LS and vice versa explicitly by instructing the DMA controller of the MFC. Due to the relatively small size of the LS and the lack of transparent access to main memory, the programmer has to ensure that instructions and the active data set fit into the LS and are transferred between main memory and LS accordingly.

Dual-issuing. The SXU has a pure RISC-like SIMD instruction set encoded into 32-bit instruction words; instructions are issued strictly in order to two pipelines called *odd* and *even pipeline*, which execute disjoint subsets of the instruction set. The even pipeline handles floating-point operations, integer arithmetic, logical instructions, and word SIMD shifts and rotates. The odd pipeline executes byte-granularity shift, rotate-mask, and shuffle operations on quadwords, and branches as well as loads and stores.

Up to two instructions can be issued each cycle, one in each pipeline, given that alignment rules are respected (i.e., the instruction for the even pipeline is aligned to a multiple of 8 bytes and the instruction for the odd pipeline is aligned to a multiple of 8 bytes plus an offset of 4 bytes), that there are no interdependencies to pending previous instructions for either of the two instructions, and that there are in fact at least two instructions available for execution. Therefore, a careful scheduling and alignment of instructions is necessary to achieve peak performance.

2.2 MFC – Accessing Main Memory

As mentioned before, the MFC is the gate for the SPU to reach main memory as well as other processor elements. Memory transfer is initiated by the SPU and afterwards executed by the DMA controller of the MFC in parallel to ongoing instruction execution by the SPU.

Since data transfers are executed in background by the DMA controller, the SPU needs feedback about when a previously initiated transfer has finished. Therefore, each transfer is tagged with one of 32 tags. Later on, the SPU can probe either in a blocking or non-blocking way if a subset of tags has any outstanding transactions. The programmer should avoid to read data buffers for incoming data or to write to buffers for outgoing data before checking the state of the corresponding tag to ensure deterministic program behaviour.

2.3 LS – Accessing Local Storage

The LS is single ported and has a *line interface* of 128 bytes width for DMA transfers and instruction fetch as well as a *quadword interface* of 16 bytes width for SPU load and store. Since there is only one port, the access to the LS is arbitrated using the following priorities:

1. DMA transfers (at most every 8 cycles),
2. SPU load/store,
3. instruction fetch.

Instructions are fetched in lines of 128 bytes, i.e., 32 instructions. In the case that all instructions can be dual issued, new instructions need to be fetched every 16 cycles. Since SPU loads/stores have precedence over instruction fetch, in case of high memory access there should be an `lnop` instruction for the odd pipeline every 16 cycles to avoid instruction starvation. If there are ongoing DMA transfers an `hbrp` instruction should be used giving instruction fetch explicit precedence over DMA transfers.

2.4 Determining Performance

The CBEA offers two ways to determine performance of SPU code: performance can either be statically analysed or measured during runtime.

Static analysis. Since all instructions are executed in order and dual-issue rules only depend on latencies and alignment, it is possible to determine the performance of code through static analysis. The "IBM SDK for Multicore Acceleration" [13] contains the tool `spu_timing` which performs this static analysis and gives quite accurate cycle counts.

This tool has the disadvantage that it assumes fully linear execution and does not model instruction fetch. Therefore the results reported by `spu_timing` are overly optimistic for code that contains loops or a high number of memory accesses. Furthermore, `spu_timing` can only be used inside a function. Function calls—in particular calling overhead on the caller side—can not be analyzed.

Measurement during runtime. Another way to determine performance is through an integrated decrementer (see [14, Sec. 13.3.3]). Measuring cycles while running the code captures all effects on performance in contrast to static code analysis.

The disadvantage of the decrementer is that it is updated with the frequency of the so-called timebase of the processor. The timebase is usually much smaller than the processor frequency. The Cell/B.E. in the Playstation 3 (rev. 5.1) for example changes the decrementer only every 40 cycles, the Cell/B.E. in the QS21 blades even only every 120 cycles. Small sections of code can thus only be measured on average by running the code several times repeatedly.

For cycle counts we report in this paper we will always state whether the count was obtained using `spu_timing`, or measured by running the code.

3 Preliminaries

The main task on solving the Certicom challenge ECC2K-130 is to compute a specific discrete logarithm on a given elliptic curve. Up to now, the most efficient algorithm known to solve this challenge is a parallel version of Pollard's rho algorithm running concurrently on a big number of machines.

In this section we give the mathematical and algorithmic background necessary to understand the implementations described in this paper. We will briefly explain the parallel version of Pollard's rho algorithm, review the iteration function described in [2], and introduce the general structure of our implementation.

3.1 The ECDLP and Parallel Pollard rho

The security of elliptic-curve cryptography relies on the believed hardness of the elliptic curve discrete logarithm problem (ECDLP): 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 $Q = [k]P$. Here, $[k]$ denotes scalar multiplication with k.

If the order of $\langle P \rangle$ is prime, the best-known algorithm to solve this problem (for most elliptic curves) is Pollard's rho algorithm [17]. In the following we describe the parallelized collision search as implemented in [2]. See [2] for credits and further discussion.

The algorithm uses a pseudo-random iteration function $f : \langle P \rangle \rightarrow \langle P \rangle$ and declares a subset of $\langle P \rangle$ as *distinguished points*. The parallelization is implemented in a client-server approach in which each client node generates an input point with known linear combination in P and Q, i.e. $R_0 = [a_0]P + [b_0]Q$ with a_0 and b_0 generated from a random seed s. It then iteratively computes $R_{i+1} = f(R_i)$ until the iteration reaches a distinguished point R_d. The random seed s and the distinguished point are then sent to a central server, the client continues by generating a new random input point.

The server searches for a *collision* in all distinguished points sent by the clients, i.e. two different input points reaching the same distinguished point R_d. The iteration function is constructed in such a way that the server can compute a_d and b_d such that $R_d = a_d P + b_d Q$ from a_0 and b_0 (which are derived from s). If two different input points yield a collision at a distinguished point R_d, the server computes the two (most probably different) linear combinations of the point R_d in P and Q: $R_d = [a_d]P + [b_d]Q$ and $R_d = [c_d]P + [d_d]Q$. The solution to the discrete logarithm of Q to the base P is then

$$Q = \left[\frac{a_d - c_d}{b_d - d_d}\right] P.$$

The expected number of distinguished points required to find a collision depends on the density of distinguished points in $\langle P \rangle$. The expected amount of iterations of f on all nodes in total is approximately $\sqrt{\frac{\pi|\langle P \rangle|}{2}}$ assuming the iteration function f is a random mapping of size $|\langle P \rangle|$ (see [11]).

3.2 ECC2K-130 and Our Choice of the Iteration Function

The specific ECDLP addressed in this paper is given in the Certicom challenge list [9] as challenge ECC2K-130. The given elliptic curve is a 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. The challenge is to find an integer k such that $Q = [k]P$. Here we will only give the definition of distinguished points and the iteration function used in our implementation. For a detailed description please refer to [2], for a discussion and comparison to other possible choices also see [1]:

We define a point R_i as *distinguished* if the Hamming weight of the x-coordinate in normal basis representation $\mathrm{HW}(x_{R_i})$ is smaller than or equal to 34. Our iteration function is defined as

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

where σ is the Frobenius endomorphism and

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

The restriction of σ to $\langle P\rangle$ corresponds to scalar multiplication with some scalar r. For an input $R_i = a_iP + b_iQ$ the output of f will be $R_{i+1} = (r^j a_i + a_i)P + (r^j b_i + b_i)Q$. When a collision has been detected, it is possible to recompute the two according iterations and update the coefficients a_i and b_i following this rule. This gives the coefficients to compute the discrete logarithm.

3.3 Computing the Iteration Function

Computing the iteration function requires one application of σ^j and one elliptic-curve addition. Furthermore we need to convert the x-coordinate of the resulting point to normal basis, if a polynomial-basis representation is used, and check whether it is a distinguished point.

Many applications use so-called inversion-free coordinate systems to represent points on elliptic curves (see, e.g., [10, Sec. 3.2]) to speed up the computation of point multiplications. These coordinate systems use a redundant representation for points. Identifying distinguished points requires a unique representation, this is why we use the affine Weierstrass representation to represent points on the elliptic curve. Elliptic-curve addition in affine Weierstrass coordinates on the given elliptic curve requires 2 multiplications, one squaring, 6 additions, and 1 inversion in $\mathbb{F}_{2^{131}}$ (see, e.g. [6]). Application of σ^j means computing the 2^j-th powers of the x- and the y-coordinate. In total, one iteration takes 2 multiplications, 1 squaring, 2 computations of the form r^{2^m}, with $3 \leq m \leq 10$, 1 inversion, 1 conversion to normal-basis, and one Hamming-weight computation. In the following we will refer to computations of the form r^{2^m} as *m-squaring*.

A note on the inversion. To speed up the relatively costly inversion we can batch several inversions and use Montgomery's trick [16]: m batched inversions can be computed with $3(m-1)$ multiplications and one inversion. For example, $m = 64$ batched elliptic curve additions take $2 \cdot 64 + 3 \cdot (64 - 1) = 317$ multiplications, 64 squarings and 1 inversion. This corresponds to 4.953 multiplications, 1 squaring and 0.016 inversions for a single elliptic-curve addition.

4 Approaches for Implementing the Iteration Function

In the following we discuss the two main design decisions for the implementation of the iteration function: 1.) Is it faster to use bitslicing or a standard approach and 2.) is it better to use normal-basis or polynomial-basis representation for elements of the finite field.

4.1 Bitsliced or Not Bitsliced?

Binary-field arithmetic was commonly believed to be more efficient than prime-field arithmetic for hardware but less efficient for software implementations. This is due to the fact that most common microprocessors spend high effort on accelerating integer- and floating-point multiplications. Prime-field arithmetic can benefit from those high-speed multiplication algorithms, binary-field arithmetic cannot. However, Bernstein showed recently that for *batched* multiplications, binary fields can provide better performance than prime fields also in software [3]. In his implementation of batched Edwards-curve arithmetic the bitslicing technique [7] is used to compute (at least) 128 binary-field multiplications in parallel on an Intel Core 2 processor.

Bitslicing is a matter of transposition: Instead of storing the coefficients of an element of $\mathbb{F}_{2^{131}}$ as sequence of 131 bits in 2 128-bit registers, we can use 131 registers to store the 131 coefficients of an element, one register per bit. Algorithms are then implemented by simulating a hardware implementation – gates become bit operations such as `AND` and `XOR`. For one element in 131 registers this is highly inefficient, it may become efficient if all 128 bits of the registers are used for 128 independent (batched) operations. The lack of registers—most architectures including the SPU do not support 131 registers—can easily be compensated for by *spills*, i.e. storing currently unused values on the stack and loading them when they are required.

The results of [3] show that for batched binary-field arithmetic on the Intel Core 2 processor bitsliced implementations are faster than non-bitsliced implementations. However, the question whether this is also the case on the SPU of the Cell processor is hard to answer a priori for several reasons:

– The Intel Core 2 can issue up to 3 bit operations on 128-bit registers per cycle, an obvious lower bound on the cycles per iteration is thus given as the number of bit operations per cycle divided by 3. The SPU can issue only one bit operation per cycle, the lower bound on the performance is thus three times as high.
– Bernstein in [3, Sec. 3] describes that the critical bottleneck for batched multiplications are in fact loads instead of bit operations. The Core 2 has only 16 architectural 128-bit registers and can do only 1 load per cycle, i.e. one load per 3 bit operations.

  The SPUs have 128 architectural 128-bit registers and can do one load per bit operation. However, unlike on the Core 2, the load operations have to compete with store operations. Due to the higher number of registers and the lower

arithmetic/load ratio it seems easier to come close to the lower bound of cycles imposed by the number of bit operations on the SPUs than on the Core 2. How much easier and how close exactly was very hard to foresee.
- Bitsliced operations rely heavily on parallelism and therefore require much more storage for inputs, outputs and intermediate values. As described in Section 2, all data of the active set of variables has to fit into 256 KB of storage alongside the code. In order to make code run fast on the SPU which executes all instructions in order, optimization techniques such as loop unrolling and function inlining are crucial; these techniques increase the code size and make it harder to fit all data into the local storage.
- Montgomery inversions require another level of parallelism, inverting for example 64 values in parallel requires $64 \cdot 128$ field elements (131 KB) of inputs when using bitsliced representation. This amount of data can only be handled using DMA transfers between the main memory and the local storage. In order to not suffer from performance penalties due to these transfers, they have to be carefully interleaved with computations.

We decided to evaluate which approach is best by implementing both, the bitsliced and the non-bitsliced version, independently by two groups in a friendly competition.

4.2 Polynomial or Normal Basis?

Another choice to make for both bitsliced and non-bitsliced implementations is the representation of elements of $\mathbb{F}_{2^{131}}$: Polynomial bases are of the form $(1, z, z^2, z^3, \ldots, z^{130})$, so the basis elements are increasing powers of some element $z \in \mathbb{F}_{2^{131}}$. Normal bases are of the form $(\alpha, \alpha^2, \alpha^4, \ldots, \alpha^{2^{130}})$, so each basis element is the square of the previous one.

Performing arithmetic in normal-basis representation has the advantage that squaring elements is just a rotation of coefficients. Furthermore we do not need any basis transformation before computing the Hamming weight in normal basis. On the other hand, implementations of multiplications in normal basis are widely believed to be much less efficient than those of multiplications in polynomial basis.

In [19], von zur Gathen, Shokrollahi and Shokrollahi proposed an efficient method to multiply elements in type-2 normal basis representation. Here we review the multiplier shown in [2]; see [2] for further discussion and history:

An element of $\mathbb{F}_{2^{131}}$ in type-2 normal basis representation is of the form

$$f_0(\zeta + \zeta^{-1}) + f_1(\zeta^2 + \zeta^{-2}) + f_2(\zeta^4 + \zeta^{-4}) + \cdots + f_{130}(\zeta^{2^{130}} + \zeta^{-2^{130}}),$$

where ζ is a 263rd root of unity in $\mathbb{F}_{2^{131}}$. This representation is first permuted to obtain coefficients of

$$\zeta + \zeta^{-1}, \zeta^2 + \zeta^{-2}, \zeta^3 + \zeta^{-3}, \ldots, \zeta^{131} + \zeta^{-131},$$

and then transformed to coefficients in polynomial basis

$$\zeta + \zeta^{-1}, (\zeta + \zeta^{-1})^2, (\zeta + \zeta^{-1})^3, \ldots, (\zeta + \zeta^{-1})^{131}.$$

Applying this transform to both inputs allows us to use a fast polynomial-basis multiplier to retrieve coefficients of

$$(\zeta+\zeta^{-1})^2, (\zeta+\zeta^{-1})^3, \dots, (\zeta+\zeta^{-1})^{262}.$$

Applying the inverse of the input transformation yields coefficients of

$$\zeta^2+\zeta^{-2}, \zeta^3+\zeta^{-3}, \dots, \zeta^{262}+\zeta^{-262}.$$

Conversion to permuted normal basis just requires adding appropriate coefficients, for example ζ^{200} is the same as ζ^{-63} and thus $\zeta^{200}+\zeta^{-200}$ is the same as $\zeta^{63}+\zeta^{-63}$. To obtain the normal-basis representation we only have to apply the inverse of the input permutation.

This multiplication still incurs overhead compared to modular multiplication in polynomial basis, but it needs careful analysis to understand whether this overhead is compensated for by the above-described benefits of normal-basis representation. Observe that all permutations involved in this method are free for hardware and bitsliced implementations while they are quite expensive in non-bitsliced software implementations.

5 The Non-bitsliced Implementation

For the non-bitsliced implementation, we decided not to implement arithmetic in a normal-basis representation. The main reason is that the required permutations, splitting and reversing of the bits, as required for the conversions in the Shokrollahi multiplication algorithm (see Section 4.2) are too expensive to outweigh the gain of having no basis change and faster m-squarings.

The non-bitsliced implementation uses a polynomial-basis representation of elements in $\mathbb{F}_{2^{131}} \cong \mathbb{F}_2[z]/(z^{131}+z^{13}+z^2+z+1)$. Field elements in this basis can be represented using 131 bits, on the SPE architecture this is achieved by using two 128-bit registers, one containing the three most significant bits. As described in Section 3 the functionality of addition, multiplication, squaring and inversion are required to implement the iteration function. Since the distinguished-point property is defined on points in normal basis, a basis change from polynomial to normal basis is required as well. In this section the various implementation decisions for the different (field-arithmetic) operations are explained.

The implementation of an addition is trivial and requires two `XOR` instructions. These are instructions going to the even pipeline; each of them can be dispatched together with one instruction going to the odd pipeline. The computation of the Hamming weight is implemented using the `CNTB` instruction, which counts the number of ones per byte for all 16 bytes of a 128-bit vector concurrently, and the `SUMB` instruction, which sums the four bytes of each of the four 32-bit parts of the 128-bit input. The computation of the Hamming weight requires four cycles (measured).

In order to eliminate (or reduce) stalls due to data dependencies we interleave different iterations. Our experiments show that interleaving a maximum of eight

Algorithm 1. The reduction algorithm for the ECC2K-130 challenge used in the non-bitsliced version. The algorithm is optimized for architectures with 128-bit registers.

```
Input: $C = A \cdot B = a + b \cdot z^{128} + c \cdot z^{256}$, such that $A, B \in \mathbb{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$
    and $a, b, c$ are 128-bit strings representing polynomial values.
Output: $D = C \bmod (z^{131} + z^{13} + z^2 + z + 1)$.
1: $c \leftarrow (c \ll 109) + (b \gg 19)$
2: $b \leftarrow b$ AND $(2^{19} - 1)$
3: $c \leftarrow c + (c \ll 1) + (c \ll 2) + (c \ll 13)$
4: $a \leftarrow a + (c \ll 16)$
5: $b \leftarrow b + (c \gg 112)$
6: $x \leftarrow (b \gg 3)$
7: $b \leftarrow b$ AND 7
8: $a \leftarrow a + x + (x \ll 1) + (x \ll 2) + (x \ll 13)$
9: return $(D = a + b \cdot z^{128})$
```

iterations maximizes performance. We process 32 of such batches in parallel, computing on 256 iterations in order to reduce the cost of the inversion (see Section 3). All 256 points are converted to normal basis, we keep track of the lowest Hamming weight of the x-coordinate among these points. This can be done in a branch-free way eliminating the need for 256 expensive branches. Then, before performing the simultaneous inversion, only one branch is used to check if one of the points is distinguished. If one or more distinguished points are found, we have to process all 256 points again to determine and output the distinguished points. Note that this happens only very infrequently.

5.1 Multiplication

If two polynomials $A, B \in \mathbb{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$ are multiplied in a straight-forward way using 4-bit lookup tables, the table entries would be 134-bit wide. Storing and accumulating these entries would require operations (`SHIFT` and `XOR`) on two 128-bit limbs. In order to reduce the number of required operations we split A as

$$A = A_l + A_h \cdot z^{128} = \widetilde{A}_l + \widetilde{A}_h \cdot z^{121}.$$

This allows us to build a 4-bit lookup table from $\widetilde{A}_l$ whose entries fit in 124 bits (a single 128-bit limb). Furthermore, the product of $\widetilde{A}_l$ and an 8-bit part of B fits in a single 128-bit limb. While accumulating such intermediate results we only need byte-shift instructions. In this way we calculate the product $\widetilde{A}_l \cdot B$.

For calculating $\widetilde{A}_h \cdot B$ we split B as

$$B = B_l + B_h \cdot z^{128} = \widetilde{B}_l + \widetilde{B}_h \cdot z^{15}.$$

Then we calculate $\widetilde{A}_h \cdot \widetilde{B}_l$ and $\widetilde{A}_h \cdot \widetilde{B}_h$ using two 2-bit lookup tables from $\widetilde{B}_l$ and $\widetilde{B}_h$. We choose to split 15 bits from B in order to facilitate the accumulation of partial products in

$$C = A \cdot B = \widetilde{A}_l \cdot B_l + \widetilde{A}_l \cdot B_h \cdot z^{128} + \widetilde{A}_h \cdot \widetilde{B}_l \cdot z^{121} + \widetilde{A}_h \cdot \widetilde{B}_h \cdot z^{136}$$

since $121 + 15 = 136$ which is divisible by 8.

The reduction can be done efficiently by taking the form of the irreducible polynomial into account. Given the result C from a multiplication or squaring, $C = A \cdot B = C_H \cdot z^{131} + C_L$, the reduction is calculated using the trivial observation that

$$C_H \cdot z^{131} + C_L \equiv C_L + (z^{13} + z^2 + z^1 + 1)C_H \bmod (z^{131} + z^{13} + z^2 + z + 1).$$

Algorithm 1 shows the reduction algorithm optimized for architectures which can operate on 128-bit operands. This reduction requires 10 `XOR`, 11 `SHIFT` and 2 `AND` instructions. On the SPU architecture the actual number of required `SHIFT` instructions is 15 since the bit-shifting instructions only support values up to 7. Larger bit-shifts are implemented combining both a byte- and a bit-shift instruction. When interleaving two independent modular multiplication computations, parts of the reduction and the multiplication of both calculations are interleaved to reduce latencies, save some instructions and take full advantage of the available two pipelines.

When doing more than one multiplication containing the same operand, we can save some operations. By doing the simultaneous inversion in a binary-tree style we often have to compute the products $A \cdot B$ and $A' \cdot B$. In this case, we can reuse the 2-bit lookup tables from $\tilde{B}_l$ and $\tilde{B}_h$. We can also save operations in the address generation of the products $\tilde{A}_l \cdot B_l$, $\tilde{A}_l \cdot B_h$, $\tilde{A}'_l \cdot B_l$ and $\tilde{A}'_l \cdot B_h$. Using these optimizations in the simultaneous inversion a single multiplication takes 149 cycles (`spu_timing`) averaged over the five multiplications required per iteration.

5.2 Squaring

The squaring is implemented by inserting a zero bit between each two consecutive bits of the binary representation of the input. This can be efficiently implemented using the `SHUFFLE` and `SHIFT` instructions. The reduction is performed according to Algorithm 1. Just as with the multiplication two squaring computations are interleaved to reduce latencies. A single squaring takes 34 cycles (measured).

5.3 Basis Conversion and m-Squaring

The repeated Frobenius map σ^j requires at least 6 and at most 20 squarings, both the x- and y-coordinate of the current point need at most $3 + 7 = 10$ squarings each (see Section 3), when computed as a series of single squarings. This can be computed in at most $20 \times 34 = 680$ cycles ignoring loop overhead using our single squaring implementation.

To reduce this number a time-memory tradeoff technique is used. We precompute all values

$$T[k][j][i_0 + 2i_1 + 4i_2 + 8i_3] = (i_0 \cdot z^{4j} + i_1 \cdot z^{4j+1} + i_2 \cdot z^{4j+2} + i_3 \cdot z^{4j+3})^{2^{3+k}}$$

for $0 \leq k \leq 7$, $0 \leq j \leq 32$, $0 \leq i_0, i_1, i_2, i_3 \leq 1$. These precomputed values are stored in two tables, for both limbs needed to represent the number, of $8 \times 33 \times 16$ elements of 128-bit each. This table requires 132 KB which is more than half of the available space of the local store.

Given a coordinate a of an elliptic-curve point and an integer $0 \leq m \leq 7$ the computation of the m-squaring $a^{2^{3+m}}$ can be computed as

$$\sum_{j=0}^{32} T[m][j][\lfloor (a/2^{4j}) \rfloor \bmod 2^4].$$

This requires 2×33 `LOAD` and 2×32 `XOR` instructions, due to the use of two tables, plus the calculation of the appropriate address to load from. Our assembly implementation of the m-squaring function requires 96 cycles (measured), this is 1.06 and 3.54 times faster compared to performing 3 and 10 sequential squarings respectively.

For the basis conversion we used a similar time-memory tradeoff technique. We enlarged the two tables by adding $1 \times 33 \times 16$ elements which enables us to reuse the m-squaring implementation to compute the basis conversion. For the computation of the basis conversion we proceed exactly the same as for the m-squarings, only the initialization of the corresponding table elements is different.

5.4 Modular Inversion

From Fermat's little theorem it follows that the modular inverse of $a \in \mathbb{F}_{2^{131}}$ can be obtained by computing $a^{2^{131}-2}$. This can be implemented using 8 multiplications, 6 m-squarings (using $m \in \{2, 4, 8, 16, 32, 65\}$) and 3 squarings. When processing many iterations in parallel the inversion cost per iteration is small compared to the other main operations such as multiplication. Considering this, and due to code-size considerations, we calculate the inversion using the fast routines we already have at our disposal: multiplication, squaring and m-squaring, for $3 \leq m \leq 10$. In total the inversion is implemented using 8 multiplications, 14 m-squarings and 7 squarings. All these operations depend on each other; hence, the interleaved (faster) implementations cannot be used. Our implementation of the inversion requires 3784 cycles (measured).

We also implemented the binary extended greatest common divisor [18] to compute the inverse. This latter approach turned out to be roughly 2.1 times slower.

6 The Bitsliced Implementation

This section describes implementation details for the speed-critical functions of the iteration function using bitsliced representation for all computations. As explained in Section 4, there is no overhead from permutations when using bitslicing and therefore the overhead for normal-basis multiplication is much lower than for a non-bitsliced implementation. We implemented all finite-field operations in polynomial- and normal-basis representation and then compared the performance. The polynomial-basis implementation uses $\mathbb{F}_{2^{131}} \cong \mathbb{F}_2[z]/(z^{131} + z^{13} + z^2 + z + 1)$ just as the non-bitsliced implementation.

Due to the register width of 128, all operations in the bitsliced implementation processes 128 inputs in parallel. The cycle counts in this section are therefore for 128 parallel computations.

6.1 Multiplication

Polynomial basis. The smallest known number of bit operations required to multiply two degree-130 polynomials over $\mathbb{F}_2$ is 11961 [4]. However, converting the sequence of bit operations in [4] to C syntax and feeding it to spu-gcc does not compile because the size of the resulting function exceeds the size of the local storage. After reducing the number of variables for intermediate results and some more tweaks the compiler produced functioning code, which had a code size of more than 100 KB and required more than 20000 cycles to compute a multiplication.

We decided to sacrifice some bit operations for code size and better-scheduled code and composed the degree-130 multiplication of 9 degree-32 multiplications using two levels of the Karatsuba multiplication technique [15]. One of these multiplications is actually only a degree-31 multiplication; in order to keep code size small we use degree-32 multiplication with leading coefficient zero. We use improvements to classical Karatsuba described in [3] to combine the results of the 9 multiplications.

The smallest known number of bit operations for degree-32 binary polynomial multiplication is 1286 [4]. A self-written scheduler for the bit operation sequence from [4] generates code that takes 1303 cycles (`spu_timing`) for a degree-32 binary polynomial multiplication. In total our degree-130 multiplication takes 14503 cycles (measured). This includes 11727 cycles for 9 degree-32 multiplications, cycles required for combination of the results, and function-call overhead.

Reduction modulo the pentanomial $z^{131} + z^{13} + z^2 + z + 1$ takes 520 bit operations, our fully unrolled reduction function takes 590 cycles (measured), so multiplication in $\mathbb{F}_{2^{131}}$ takes $14503 + 590 = 15093$ cycles.

Normal basis. The normal-basis multiplication uses the conversion to polynomial basis as described in Section 4.2. For both, conversion of inputs to polynomial basis and conversion of the result to normal basis (including reduction) we use fully unrolled assembly functions. As for multiplication we implemented scripts to schedule the code optimally. One input conversion takes 434 cycles (measured), output conversion including reduction takes 1288 cycles (measured), one normal-basis multiplication including all conversions takes 16653 cycles (measured).

6.2 Squaring

Polynomial basis. In polynomial-basis representation, a squaring consists of inserting zero-bits between all bits of the input and modular reduction. The first part does not require any instructions in bitsliced representation because we do not have to store the zeros anywhere, we only have to respect the zeros during reduction. For squarings, the reduction is cheaper than for multiplications because we know that every second bit is zero. In total it needs 190 bit operations, hence, squaring is bottlenecked by loading 131 inputs and storing 131 outputs. One call to the squaring function takes 400 cycles (measured).

Normal basis. In normal basis a squaring is a cyclic shift of bits, so we only have to do 131 loads and 131 stores to cyclically shifted locations. A call to the squaring function in normal-basis representation takes 328 cycles (measured).

6.3 *m*-Squaring

Polynomial basis. In polynomial basis we decided to implement m-squarings as a sequence of squarings. A fully unrolled code can hide most of the 131 load and 131 store operations between the 190 bit operations of a squaring – implementing dedicated m-squaring functions for different values of m would mostly remove the overhead of $m-1$ function calls but on the other hand significantly increase the overall code size.

Normal basis. For the normal-basis implementation we implemented m-squarings for all relevant values of m as separate fully unrolled functions. The only difference between these functions is the shifting distance of the store locations. Each m-squaring therefore takes 328 cycles (measured), just like a single squaring.

Conditional *m*-Squaring. The computation of σ^j cannot just simply be realized as a single m-squaring with $m = j$, because the value of j is most likely different for the 128 bitsliced values in one batch. Therefore the computation of $r = \sigma^j(x_{R_i})$ is carried out using 3 conditional m-squarings as follows:

```
$r \leftarrow x^{2^3}$
**if** $x_{R_i}[1]$ **then** $r \leftarrow r^2$
**if** $x_{R_i}[2]$ **then** $r \leftarrow r^{2^2}$
**if** $x_{R_i}[3]$ **then** $r \leftarrow r^{2^4}$
**return** $r$,
```

where $x_{R_i}[k]$ denotes the bit at position k of x_{R_i}. The computation of $\sigma^j(y_{R_i})$ is carried out in the same way.

When using bitsliced representation, conditional statements have to be replaced by equivalent arithmetic computations. We can compute the k-th bit of the result of a conditional m-squaring of r depending on a bit b as

$$r[k] \leftarrow (r[k] \texttt{ AND } \neg b) \texttt{ XOR } (r^{2^m}[k] \texttt{ AND } b).$$

The additional three bit operations per output bit can be interleaved with loads and stores needed for squaring. In particular when using normal-basis squaring (which does not involve any bit operations) this speeds up the computation: A conditional m-squaring in normal-basis representation takes 453 cycles (measured).

For the polynomial-basis implementation we decided to first compute an m-squaring and then a separate conditional move. This `cmov` function requires 262 loads, 131 stores and 393 bit operations and thus balances instructions on the two pipelines. One call to the `cmov` function takes 518 cycles.

6.4 Addition

Addition is the same for normal-basis and polynomial-basis representation. It requires loading 262 inputs, 131 `XOR`s and storing of 131 outputs. Just as squaring, the function is bottlenecked by loads and stores rather than bit operations. One call to the addition function takes 492 cycles (measured).

6.5 Inversion

For both polynomial and normal basis the inversion is implemented using Fermat's little theorem. It involves 8 multiplications, 3 squarings and 6 m-squarings (with $m = 2, 4, 8, 16, 32, 65$). It takes 173325 cycles using polynomial basis and 136132 cycles using normal basis (both measured). Observe that with a sufficiently large batch size for Montgomery inversion this does not have big impact on the cycle count of one iteration.

6.6 Conversion to Normal Basis

Polynomial basis. For the polynomial-basis implementation we have to convert the x-coordinate to normal basis to check whether we found a distinguished point. This basis conversion is generated using the techniques described in [5] and uses 3380 bit operations. The carefully scheduled code takes 3748 cycles (measured).

6.7 Hamming-Weight Computation

The bitsliced Hamming-weight computation of a 131-bit number represented in normal basis can be done in a divide-and-conquer approach (producing bitsliced results) using 625 bit operations. We unrolled this algorithm to obtain a function that computes the Hamming weight using 844 cycles (measured).

6.8 Control Flow Overhead

For both, polynomial-basis and normal-basis representation there is additional overhead from additions, loop control, and reading new input points after a distinguished point has been found. This overhead accounts for only about 8 percent of the total computation time. Reading a new input point after a distinguished point has been found takes about 2,009,000 cycles. As an input point takes on average $2^{25.7} \approx 40,460,197$ iterations to reach a distinguished point, these costs are negligible and are ignored in our overall cycle counts for the iteration function.

6.9 Complete Iteration

To make the computation of the iteration function as fast as possible we used the largest batch size for Montgomery inversions that allows us to fit all data into the local storage. Our polynomial-basis implementation uses a batch size of 12 and needs 113844 cycles (measured) to compute the iteration function. The normal-basis implementation uses a batch size of 14 and requires 100944 cycles (measured). Clearly, the overhead caused by the conversions for multiplications in the normal-basis implementation is outweighed by the benefits in faster m-squarings, conditional m-squarings, and the saved basis conversion.

6.10 Using DMA Transfers to Increase the Batch Size

To be able to use larger numbers for the batch size we modified the normal-basis implementation to make use of main memory. The batches are stored in main memory and are fetched into LS temporarily for computation.

Since the access pattern to the batches is totally deterministic, it is possible to use multi-buffering to prefetch data while processing previously loaded data and to write back data to main memory during ongoing computations. Even though 3 slots—one for outgoing data, one for computation, and one for incoming data—are sufficient for the buffering logic, we use 8 slots in local memory as ringbuffer to hide indeterministic delays on the memory bus. We assign one DMA tag to each of these slots to monitor ongoing transactions.

Before computation, one slot is chosen for the first batch and the batch is loaded to LS. During one step of the iteration function, the SPU iterates multiple times over the batches. Each time, first the SPU checks whether the last write back from the next slot has finished using a blocking call to the MFC on the assigned tag. Then it initiates a prefetch for the next required batch into this next slot. Now—again in a blocking manner—it is checked whether the data for the current batch already has arrived. If so, data is processed and finally the SPU initiates a DMA transfer to write changed data back to main memory.

Due to this access pattern, all data transfers can be performed with minimal overhead and delay. Therefore it is possible to increase the batch size to 512 improving the runtime per iteration for the normal basis implementation by about 5 percent to 95428 cycles (measured). Measurements on IBM blade servers QS21 and QS22 showed that neither processor bus nor main memory are a bottleneck even if 8 SPEs are doing independent computations and DMA transfers in parallel.

7 Conclusions

To the best of our knowledge there were no previous attempts to implement fast binary-field arithmetic on the Cell. The closest work that we are aware of is [8], in which Bos, Kaihara and Montgomery solved an elliptic-curve discrete-logarithm problem over a 112-bit prime field using a PlayStation 3 cluster of 200 nodes. Too many aspects of both the iteration function and the underlying

Table 1. Cycle counts per input for all operations on one SPE of a 3192 MHz Cell Broadband Engine, rev. 5.1. For the bitsliced implementations, cycle counts for 128 inputs are divided by 128. The value B in the last row denotes the batch size for Montgomery inversions.

	Non-bitsliced, polynomial basis	Bitsliced, polynomial basis	Bitsliced, normal basis
Squaring	34	3.164	2.563
m-squaring	96	$m \times 3.164$	2.563
Conditional m-squaring	—	$m \times 3.164 + 4.047$	3.539
Multiplication	149	117.914	130.102
Addition	2	3.844	
Inversion	3784	1354.102	1063.531
Conversion to normal basis	96	29.281	—
Hamming-weight computation	4	6.594	
Pollard's rho iteration	1148 ($B = 256$)	889.406 ($B = 12$)	788.625 ($B = 14$) 745.531 ($B = 512$)

field arithmetic are different from the implementation in this paper to allow a meaningful comparison.

From the two implementations described in this paper it is clear that on the Cell processor bitsliced implementations of highly parallel binary-field arithmetic are more efficient than standard implementations. Furthermore we show that normal-basis representation of finite-field elements outperforms polynomial-basis representation when using a bitsliced implementation. For applications that do not process large batches of different independent computations the non-bitsliced approach remains of interest. The cycle counts for all field operations are summarized in Table 1 for both approaches.

Using the bitsliced normal-basis implementation—which uses DMA transfers to main memory to support a batch size of 512 for Montgomery inversions—on all 6 SPUs of a Sony Playstation 3 in parallel, we can compute 25.57 million iterations per second. The expected total number of iterations required to solve the ECDLP given in the ECC2K-130 challenge is $2^{60.9}$ (see [2]). Using the software described in this paper, this number of iterations can be computed in 2654 Playstation 3 years.

References

1. Bailey, D.V., Baldwin, B., Batina, L., Bernstein, D.J., Birkner, P., Bos, J.W., van Damme, G., de Meulenaer, G., Fan, J., Güneysu, T., Gurkaynak, F., Kleinjung, T., Lange, T., Mentens, N., Paar, C., Regazzoni, F., Schwabe, P., Uhsadel, L.: The Certicom challenges ECC2-X. In: Workshop Record of SHARCS 2009: Special-purpose Hardware for Attacking Cryptographic Systems, pp. 51–82 (2009), `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`

2. Bailey, D.V., Batina, L., Bernstein, D.J., Birkner, P., Bos, J.W., Chen, H.-C., Cheng, C.-M., Van Damme, G., de Meulenaer, G., Dominguez Perez, L.J., Fan, J., Güneysu, T., Gürkaynak, F., Kleinjung, T., Lange, T., Mentens, N., Niederhagen, R., Paar, C., Regazzoni, F., Schwabe, P., Uhsadel, L., Van Herrewege, A., Yang, B.-Y.: Breaking ECC2K-130 (2009), `http://eprint.iacr.org/2009/541`
3. Bernstein, D.J.: Batch binary Edwards. In: Halevi, S. (ed.) Advances in Cryptology – CRYPTO 2009. LNCS, vol. 5677, pp. 317–336. Springer, Heidelberg (2009)
4. Bernstein, D.J.: Minimum number of bit operations for multiplication (May 2009), `http://binary.cr.yp.to/m.html` (accessed 2009-12-07)
5. Bernstein, D.J.: Optimizing linear maps modulo 2. In: Workshop Record of SPEED-CC: Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers, pp. 3–18 (2009), `http://www.hyperelliptic.org/SPEED/record09.pdf`
6. Bernstein, D.J., Lange, T.: Explicit-formulas database, `http://www.hyperelliptic.org/EFD/` (accessed 2010-01-05)
7. Biham, E.: A fast new DES implementation in software. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 260–272. Springer, Heidelberg (1997)
8. Bos, J.W., Kaihara, M.E., Montgomery, P.L.: Pollard rho on the PlayStation 3. In: Workshop Record of SHARCS 2009: Special-purpose Hardware for Attacking Cryptographic Systems, pp. 35–50 (2009), `http://www.hyperelliptic.org/tanja/SHARCS/record2.pdf`
9. Certicom. Certicom ECC Challenge (1997), `http://www.certicom.com/images/pdfs/cert_ecc_challenge.pdf`
10. Hankerson, D., Menezes, A., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer, New York (2004)
11. Harris, B.: Probability distributions related to random mappings. The Annals of Mathematical Statistics 31, 1045–1062 (1960)
12. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: HPCA 2005, pp. 258–262. IEEE Computer Society, Los Alamitos (2005)
13. IBM. IBM SDK for multicore acceleration (version 3.1), `http://www.ibm.com/developerworks/power/cell/downloads.html?S_TACT=105AGX16&S_CMP=LP`
14. IBM DeveloperWorks. Cell Broadband Engine programming handbook (version 1.11), (May 2008), `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D`
15. Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. In: Proceedings of the USSR Academy of Science, vol. 145, pp. 293–294 (1962)
16. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48, 243–264 (1987)
17. Pollard, J.M.: Monte Carlo methods for index computation (mod p). Mathematics of Computation 32, 918–924 (1978)
18. Stein, J.: Computational problems associated with Racah algebra. Journal of Computational Physics 1(3), 397–405 (1967)
19. von zur Gathen, J., Shokrollahi, A., Shokrollahi, J.: Efficient multiplication using type 2 optimal normal bases. In: Carlet, C., Sunar, B. (eds.) WAIFI 2007. LNCS, vol. 4547, pp. 55–68. Springer, Heidelberg (2007)