Fast elliptic-curve cryptography on the Cell Broadband Engine

Neil Costigan[1] and Peter Schwabe[2] *

[1] School of Computing
Dublin City University, Glasnevin, Dublin 9, Ireland
neil.costigan@computing.dcu.ie
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
peter@cryptojedi.org

Abstract. This paper is the first to investigate the power of the Cell Broadband Engine for state-of-the-art public-key cryptography. We present a high-speed implementation of elliptic-curve Diffie-Hellman (ECDH) key exchange for this processor, which needs 777000 cycles on one Synergistic Processor Unit for a scalar multiplication on a 255-bit elliptic curve, including the costs for key verification and key compression. This cycle count is independent of inputs therefore protecting against timing attacks.

This speed relies on a new representation of elements of the underlying finite field suited for the unconventional instruction set of this architecture.

Furthermore we demonstrate that an implementation based on the multiprecision integer arithmetic functions provided by IBM's multi-precision math (MPM) library would take at least 9660640 cycles.

Comparison with implementations of the same function for other architectures shows that the Cell Broadband Engine is competitive in terms of cost-performance ratio to other recent processors such as the Core 2 for public-key cryptography.

Specifically, the state-of-the-art Galbraith-Lin-Scott ECDH software performs 27370 scalar multiplications per second using all four cores of a 2.5GHz Intel Core 2 Quad Q9300 inside a $400 computer, while the new software reported in this paper performs 24528 scalar multiplications per second on a Playstation 3 that costs just $279. Both of these speed reports are for high-security 256-bit elliptic-curve cryptography.

Keywords: Cell Broadband Engine, elliptic-curve cryptography (ECC), efficient implementation

* The first author was supported by the Irish Research Council for Science, Engineering and Technology (IRCSET). The second author was supported 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: a33572712a64958c0bf522e608f25f0d. Date: Jan 7, 2009

1 Introduction

This paper describes a high-speed implementation of state-of-the-art public-key cryptography for the Cell Broadband Engine (CBE). More specifically we describe an implementation of the `curve25519` function, an elliptic-curve Diffie-Hellman key exchange (ECDH) function introduced in [3].

Implementations of this function have been achieving speed records for high-security ECDH software on different platforms [3], [9]. Benchmarks of our implementation show that the CBE is competitive (in terms of cost-performance ratio) to other recent processors as the Intel Core 2 for public-key cryptography.

Our implementation needs 777000 cycles on one Synergistic Processor Unit (SPU) of the CBE. This number includes not only scalar multiplication on the underlying 255-bit elliptic curve, but furthermore key compression, key validation and protection against timing attacks. We plan to put our implementation into the public domain to maximize reusability of our results.

We wish to thank Dan Bernstein, Tanja Lange, Ruben Niederhagen, and Michael Scott for their invaluable feedback and encouragement. Neil Costigan would also like to thank Luleå University of Technology, Sweden.

1.1 How these speeds were achieved

Elliptic-curve cryptography (ECC) is usually implemented as a sequence of arithmetic operations in a finite field. IBM provides a multi-precision math (MPM) library developed especially for the Cell Broadband Engine as part of the standard Cell Software Development Kit (SDK). The obvious approach for the implementation of ECC on the CBE is thus to use this library for the underlying finite field arithmetic.

However, we will show that the targeted performance cannot be achieved following this approach, not even with optimizing some functions of the MPM library for arithmetic in fields of the desired size.

Instead, the speed of our implementation is achieved by

- Parting with the traditional way of implementing elliptic-curve cryptography which uses arithmetic operations in the underlying field as smallest building blocks,
- Representing finite field elements in a way that takes into account the special structure of the finite field and the unconventional SPU instruction set, and
- Careful optimization of the code on assembly level.

Related work Implementations of public-key cryptography for the Cell Broadband Engine have not yet been extensively studied. In particular we don't know of any previous implementation of ECC for the Cell Broadband Engine.

Costigan and Scott investigate in [5] the use of IBM's MPM library to accelerate OpenSSL on a Sony Playstation 3. The paper reports benchmarks for

RSA with different key lengths; RSA signature generation with a 2048 bit key is reported to take 0.015636s corresponding to 50035200 cycles on one SPU.

An implementation of the Digital Signature Algorithm (DSA) supporting key lengths up to 1024 bits is included in the SPE Cryptographic Library [15].

In [17] Shimizu et al. report 4074000 cycles for 1024-bit-RSA encryption or decryption and 1331000 cycles for 1024-bit-DSA key generation. Furthermore they report 2250000 cycles for 1024-bit-DSA signature generation and 4375000 cycles for 1024-bit-DSA signature verification.

The Cell Broadband Engine has recently demonstrated its power for cryptanalysis of symmetric cryptographic primitives [19], [18],

Organization of the paper In Section 2 we will briefly review the features of the CBE which are relevant for our implementations. Section 3 describes the `curve25519` function including some necessary background on elliptic-curve arithmetic. Section 4 describes IBM's MPM library including optimizations we applied to accelerate arithmetic in finite fields of the desired size. We show that an implementation based on this library cannot achieve the targeted performance. In Section 5 we detail our implementation of `curve25519`. We conclude the paper with a discussion of benchmarking results and a comparison to ECDH implementations for other architectures in Section 6.

2 The Cell Broadband Engine

When it became apparent that multi-core chip design rather than increased frequency was the gateway to more efficient CPUs, IBM, Sony and Toshiba took a novel approach: Instead of developing a chip with every core being the same, they came up with the Cell Broadband Engine Architecture (CBEA). Currently two different CBEA processors are available: the Cell Broadband Engine (CBE) and the PowerXCell 8i processor. Both are multi-core processors consisting of a traditional central processor and 8 specialized high performance processors called Synergistic Processor Units (SPUs).

These units are combined across a high bandwidth (204 GB/s) [11] bus to offer a multi-core environment with two instruction sets and enormous processing power. Compared with the CBE the PowerXCell 8i processor has highly increased double precision floating point performance. The implementation described in this paper is optimized for the CBE, we will therefore in the following focus on the description of this processor.

The Cell Broadband Engine can be found in the Sony Playstation 3 and the IBM QS20 and QS21 blade server series. Note that the CBE in the Playstation 3 makes just 6 out of 8 SPUs available for general purpose computations. Toshiba equips several laptops of the Qosmio series with the SpursEngine consisting of 4 SPUs intended for media processing. This SpursEngine can also be found in a PCI Express card called WinFast pxVC1100 manufactured by Leadtek which is currently available only in Japan.

The primary processor of the Cell Broadband Engine is a 64-bit Power Processor Unit (PPU). This PPU works as a supervisor for the other cores. Currently operating at 3.2GHz, The PPU is a variant of the G5/PowerPC product line, a RISC driven processor found in IBM's servers and Apple's PowerMac range.

2.1 The Cell's SPU

The real power of the CBE is in the additional SPUs. Each SPU is a specialist processor with a RISC-like SIMD instruction set and a 128-element array of 128-bit registers. It has two pipelines (pipeline 0 and pipeline 1); each cycle it can dispatch one instruction per pipeline. Whether or not the SPU really dispatches two instructions in a given cycle is highly dependent on instruction scheduling and alignment. This is subject to the following conditions:

- Execution of instructions is purely in-order.
- The two pipelines execute disjoint sets of instructions (i.e. each instruction is either a pipeline-0 or a pipeline-1 instruction).
- The SPU has a fetch queue that can contain at most two instructions.
- Instructions are fetched into the fetch queue only if the fetch queue is empty.
- Instructions are fetched in pairs; the first instruction in such a pair is from an even word address, the second from an odd word address.
- The SPU executes two instructions in one cycle only if two instructions are in the fetch queue, the first being a pipeline-0 instruction and the second being a pipeline-1 instruction and all inputs to these instructions being available and not pending due to latencies of previously executed instructions.

Hence, instruction *scheduling* has to ensure that pipeline-0 and pipeline-1 instructions are interleaved and that latencies are hidden; instruction *alignment* has to ensure that pipeline-0 instructions are at even word addresses and pipeline-1 instructions are at odd word addresses.

Both our implementation and the MPM library build the finite field arithmetic on the integer arithmetic instructions of the SPU. This is due to the fact that single-precision floating-point arithmetic offers a too small mantissa and that double-precision floating-point arithmetic causes excessive pipeline stalls on the SPU and is therefore very inefficient.

All integer arithmetic instructions (except shift and rotate instructions) are SIMD instructions operating either on 4 32-bit word elements or on 8 16-bit halfword elements or on 16 8-bit byte elements of a 128-bit register.

Integer multiplication constitutes an exception to this scheme: The integer multiplication instructions multiply 4 16-bit halfwords in parallel and store the 32-bit results in the 4 word elements of the results register.

The following instructions are the most relevant for our implementation; for a detailed description of the SPU instruction set see [13], for a list of instruction latencies and associated pipelines see [12, Appendix B].

`a`: Adds each 32-bit word element of a register a to the corresponding word element of a register b and stores the results in a register r.

`mpy`: Multiplies the 16 least significant bits of each 32-bit word element of a register a with the corresponding 16 bits of each word element of a register b and stores the resulting four 32-bit results in the four word elements of a register r.

`mpya`: Multiplies 16-bit halfwords as the `mpy` instruction but adds the resulting four 32-bit word elements to the corresponding word elements of a register c and stores the resulting sum in a register r.

`shl`: Shifts each word element of a register a to the left by the number of bits given by the corresponding word element of a register b and stores the result in a register r.

`rotmi`: Shifts of each word element of a register a to the right by the number of bits given in an immediate value and stores the result in a register r.

`shufb`: Allows to set each of the result register r to either the value of an arbitrary byte of one of two input registers a and b or to a constant value of `0`, `0x80` or `0xff`.

2.2 Computation micro-kernels for the SPU

Alvaro, Kurzak and Dongarra [1] introduce the idea of a *computation micro-kernel* for the SPU where the restricted code and data size of the SPU become important design criteria but issues such as inter-chip communication and synchronization are not considered. The kernel focuses on utilization of the wide registers and the instruction level parallelism. Furthermore, for security aware applications such as those using ECC, there is an interesting security architecture where an SPU can run in *isolation mode*, where inter-chip communications, loading and unloading program code incur significant overhead. In this paper we describe such a computation micro-kernel implementation running on one SPU of the CBE.

3 ECDH and the `curve25519` function

3.1 Elliptic-Curve Diffie-Hellman Key Exchange (ECDH)

Let $\mathbb{F}$ be a finite field and $E/\mathbb{F}$ an elliptic curve defined over $\mathbb{F}$. Let $E(\mathbb{F})$ denote the group of $\mathbb{F}$-rational points on E. For any $P \in E(\mathbb{F})$ and $k \in \mathbb{Z}$ we will denote the k-th scalar multiple of P as $[k]P$.

The Diffie-Hellman key exchange protocol [6] can now be carried out in the group $\langle P \rangle \subseteq E(\mathbb{F})$ as follows: User A chooses a random $a \in \{2, \dots, |\langle P \rangle| - 1\}$, computes $[a]P$ and sends this to user B. User B chooses a random $b \in \{2, \dots, |\langle P \rangle| - 1\}$, computes $[b]P$ and sends this to user A. Now both users can compute $Q = [a]([b]P) = [b]([a]P) = [(a \cdot b)]P$. The joint key for secret key cryptography is then extracted from Q; a common way to do this is to compute a hash value of the x-coordinate of Q.

3.2 Montgomery Arithmetic

For elliptic curves defined by an equation of the form $By^2 = x^3 + Ax^2 + x$, Montgomery introduced in [16] a fast method to compute the x-coordinate of a point $R = P + Q$, given the x-coordinates of two points P and Q and the x-coordinate of their difference $P - Q$.

These formulas lead to an efficient algorithm to compute the x-coordinate of $Q = [k]P$ for any point P. This algorithm is often referred to as the Montgomery ladder. In this algorithm the x-coordinate x_P of a point P is represented as (X_P, Z_P), where $x_P = X_P/Z_P$; for the representation of the point at infinity see the discussion in Appendix B of [3]. See Algorithms 1 and 2 for a pseudocode description of the Montgomery ladder.

Algorithm 1 The Montgomery ladder for x-coordinate-based scalar multiplication on the elliptic curve $E : By^2 = x^3 + Ax^2 + x$

```
Input: A scalar $0 \leq k \in \mathbb{Z}$ and the $x$-coordinate $x_P$ of some point $P$
Output: $(X_{[k]P}, Z_{[k]P})$ fulfilling $x_{[k]P} = X_{[k]P}/Z_{[k]P}$
  $t = \lceil \log_2 k + 1 \rceil$
  $X_1 = x_P$; $X_2 = 1$; $Z_2 = 0$; $X_3 = x_P$; $Z_3 = 1$
  for $i \leftarrow t - 1$ downto 0 do
      if bit $i$ of $k$ is 1 then
          $(X3, Z3, X2, Z2) \leftarrow$ LADDERSTEP$(X1, X3, Z3, X2, Z2)$
      else
          $(X2, Z2, X3, Z3) \leftarrow$ LADDERSTEP$(X1, X2, Z2, X3, Z3)$
      end if
  end for
  return $(X_2, Z_2)$
```

Each "ladder step" as described in Algorithm 2 requires 5 multiplications, 4 squarings, 8 additions and one multiplication with the constant $a24 = (A+2)/4$ in the underlying finite field.

3.3 The `curve25519` function

Bernstein proposed in [3] the `curve25519` function for elliptic-curve Diffie-Hellman key exchange. This function uses arithmetic on the elliptic curve defined by the equation $E : y^2 = x^3 + Ax^2 + x$ over the field $\mathbb{F}_p$, where $p = 2^{255} - 19$ and $A = 486662$; observe that this elliptic curve allows for the x-coordinate-based scalar multiplication described above.

The elliptic curve and underlying finite field are carefully chosen to meet high security requirements and to allow for fast implementation, for a detailed discussion of the security properties of `curve25519` see [3].

The `curve25519` function takes as input two 32-byte strings, one representing the x-coordinate of a point P and the other representing a 256-bit scalar k. It

Algorithm 2 One “ladder step” of the Montgomery ladder

```
const $a24 = (A + 2)/4$ ($A$ from the curve equation)
function LADDERSTEP($X_{Q-P}, X_P, Z_P, X_Q, Z_Q$)
    $t_1 \leftarrow X_P + Z_P$
    $t_6 \leftarrow t_1^2$
    $t_2 \leftarrow X_P - Z_P$
    $t_7 \leftarrow t_2^2$
    $t_5 \leftarrow t_6 - t_7$
    $t_3 \leftarrow X_Q + Z_Q$
    $t_4 \leftarrow X_Q - Z_Q$
    $t_8 \leftarrow t_4 \cdot t_1$
    $t_9 \leftarrow t_3 \cdot t_2$
    $X_{P+Q} \leftarrow (t_8 + t_9)^2$
    $Z_{P+Q} \leftarrow X_{Q-P} \cdot (t_8 - t_9)^2$
    $X_{[2]P} \leftarrow t_6 \cdot t_7$
    $Z_{[2]P} \leftarrow t_5 \cdot (t_7 + a24 \cdot t_5)$
    return ($X_{[2]P}, Z_{[2]P}, X_{P+Q}, Z_{P+Q}$)
end function
```

gives as output a 32-byte string representing the x-coordinate x_Q of $Q = [k]P$. For each of these values `curve25519` is assuming little-endian representation.

For our implementation we decided to follow [3] and compute x_Q by first using Algorithm 1 to compute (X_Q, Z_Q) and then computing $x_Q = Z_Q^{-1} \cdot X_Q$.

4 The MPM library and ECC

4.1 Implementation hierarchy

Implementations of elliptic-curve cryptography are often described as a hierarchy of operations. In [10, Section 5.2.1] Hankerson, Menezes and Vanstone outline a hierarchy of operations in ECC as protocols, point multiplication, elliptic-curve addition and doubling, finite-field arithmetic. Fan, Sakiyama & Verbauwhede expand this in [7] to describe a 5-layer pyramid of

1. Integrity, confidentially, authentication,
2. Elliptic-curve scalar multiplication $[k]P$,
3. Point addition and doubling,
4. Modular operations in $\mathbb{F}_p$,
5. Instructions of a w-bit core.

4.2 $\mathbb{F}_p$ arithmetic using the MPM library

In Section 3 we described how the upper 3 layers of this hierarchy are handled. Hence, the obvious next step is to look at efficient modular operations in $\mathbb{F}_p$ and how these operations can be mapped to the SIMD instructions on 128-bit registers of the SPU.

For this task of mapping operations on large integers to the SPU instruction set, IBM provides a vector-optimized multi-precision math (MPM) library [14] as part of the software development kit (SDK) offered for the Cell Broadband Engine.

This MPM library is provided in source code and its algorithms are generic for arbitrary sized numbers. They operate on 16-bit halfwords as smallest units, elements of our 255-bit field are therefore actually handled as 256-bit values.

As our computation is mostly bottlenecked by costs for multiplications and squarings in the finite field we decided to optimize these functions for 256-bit input values.

At a high level the original MPM multiplication function is structured by an array declaration section, then array initialization via loop dictated by the input sizes, a main body consisting of calculating partial products inside nested loops (again determined by the input sizes), and finally a gather section where the partial products feed into the result.

We optimized this pattern for fixed 256-bit input and 512-bit output values by using the large register array to remove the implicit array calls and then fully unrolling the loops.

The MPM library supplies a specialized function for squaring where significant optimizations should be made over a general multiply by reusing partial products. However our timings indicate that such savings are not achieved until the size of the multi-precision inputs exceeds 512-bits. We therefore take the timings of a multiplication for a squaring.

4.3 What speed can we achieve using MPM?

The Montgomery ladder in the `curve25519` computation consists of 255 ladder steps, hence, computation takes 1276 multiplications, 1020 squarings, 255 multiplications with a constant, 2040 additions and one inversion in the finite field $\mathbb{F}_{2^{255}-19}$. Table 1 gives the number of CPU cycles required for each of these operations (except inversion). Multiplication and squaring are implemented as a call to `mpm_mul` and a subsequent call to `mpm_mod`, addition is implemented as a call to `mpm_add` and a conditional call (`mpm_cmpge`) to `mpm_sub`. For multiplication we include timings of the original MPM functions and of our optimized versions. The original MPM library offers a number of options for each operation. We select the inlined option with equal input sizes for fair comparison.

Operation	Number of cycles
Addition/Subtraction	86
Multiplication (original MPM)	4334
Multiplication (optimized)	4124

Table 1. MPM performance for arithmetic operations in a 256-bit finite field

From these numbers we can compute a lower bound of 9660640 cycles (1276M + 1020S + 2040A, where M, S and A stand for the costs of multiplication, squaring and addition respectively) required for the `curve25519` computation when using MPM. Observe that this lower bound still ignores costs for the inversion and for multiplication with the constant.

The high cost for modular reduction in these algorithms results from the fact, that the MPM library cannot make use of the special form of the modulus $2^{255} - 19$; an improved, specialized reduction routine would probably yield a smaller lower bound. We therefore investigate what lower bound we get when entirely ignoring costs for modular reduction. Table 2 gives numbers of cycles for multiplication and addition of 256-bit integers without modular reduction. This yields a lower bound of 934080 cycles. Any real implementation would, of course, take significantly more time as it would need to account for operations not considered in this estimation.

Operation	Number of cycles
Addition/Subtraction	52
Multiplication (original MPM)	594
Multiplication (optimized)	360

Table 2. MPM performance for arithmetic operations on 256-bit integers

5 Implementation of `curve25519`

As described in Section 3 the computation of the `curve25519` function consists of two parts, the Montgomery ladder computing (X_Q, Z_Q) and the inversion of Z_Q.

We decided to implement the inversion as an exponentiation with $p - 2 = 2^{255} - 21$ using the the same sequence of 254 squarings and 11 multiplications as [3]. This might not be the most efficient algorithm for inversion, but it is the easiest way to implement an inversion algorithm which takes constant time independent of the input.

Completely inlining all multiplications and squarings would result in an excessive increase of the overall code size. We therefore implement multiplication and squaring functions and use calls to these functions.

However for the first part—the Montgomery ladder—we do not use calls to these functions but take one ladder step as smallest unit implemented in a function. This allows for a higher degree of data-level parallelism, especially in the modular reductions, and thus yields a significantly increased performance.

For the speed-critical parts of our implementation we use the `qhasm` programming language [2], which offers us all flexibility for code optimization on the assembly level, while still supporting a more convenient programming model than plain assembly. We extended this language to also support the SPU of the Cell Broadband Engine as target architecture.

In the description of our implementation we will use the term "register variable". Note that for `qhasm` (unlike C) the term register variable refers to variables that are forced to be kept in registers.

5.1 Fast arithmetic

In the following we will first describe how we represent elements of the finite field $\mathbb{F}_{2^{255}-19}$ and then detail the three algorithms that influence execution speed of curve25519 most, namely finite field multiplications, finite field squaring and a Montgomery ladder step.

5.2 Representing elements of $\mathbb{F}_{2^{255}-19}$

We represent an element a of $\mathbb{F}_{2^{255}-19}$ as a tuple $(a_0, \dots, a_{19})$ where

$$a = \sum_{i=0}^{19} a_i 2^{\lceil 12.75i \rceil}. \tag{1}$$

We call a coefficient a_i *reduced* if $a_i \in [0, 2^{13} - 1]$. Analogously we call the representation of an element $a \in \mathbb{F}_{2^{255}-19}$ reduced if all its coefficients $a_0, \dots, a_{19}$ are reduced.

As described in section 2 the Cell Broadband Engine can only perform 16-bit integer multiplication, where one instruction performs 4 such multiplications in parallel. In order to achieve high performance of finite field arithmetic it is crucial to properly arrange the values $a_0, \dots a_{19}$ in registers and to adapt algorithms for field arithmetic to make use of this SIMD capability.

Multiplication and Squaring in $\mathbb{F}_{2^{255}-19}$ As input to field multiplication we get two finite field elements $(a_0, \dots, a_{19})$ and $(b_0, \dots, b_{19})$. We assume that these field elements are in reduced representation. This input is arranged in 10 register variables `a03`, `a47`, `a811`, `a1215`, `a1619`, `b03`, `b47`, `b811`, `b1215` and `b1619` as follows: Register variable `a03` contains in its word elements the coefficients a_0, a_1, a_2, a_3, register variable `a47` contains in its word elements the coefficients a_4, a_5, a_6, a_7, and so on.

The idea of multiplication is to compute coefficients $r_0, \ldots, r_{38}$ of $r = ab$ where:

$$\begin{aligned}
r_0 =& a_0b_0 \\
r_1 =& a_1b_0 + a_0b_1 \\
r_2 =& a_2b_0 + a_1b_1 + a_0b_2 \\
r_3 =& a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\
r_4 =& a_4b_0 + 2a_3b_1 + 2a_2b_2 + 2a_1b_3 + a_0b_4 \\
r_5 =& a_5b_0 + a_4b_1 + 2a_3b_2 + 2a_2b_3 + a_1b_4 + a_0b_5 \\
r_6 =& a_6b_0 + a_5b_1 + a_4b_2 + 2a_3b_3 + a_2b_4 + a_1b_5 + a_0b_6 \\
r_7 =& a_7b_0 + a_6b_1 + a_5b_2 + a_4b_3 + a_3b_4 + a_2b_5 + a_1b_6 + a_0b_7 \\
r_8 =& a_8b_0 + 2a_7b_1 + 2a_6b_2 + 2a_5b_3 + a_4b_4 + 2a_3b_5 + 2a_2b_6 + 2a_1b_7 + a_0b_8 \\
&\vdots
\end{aligned}$$

This computation requires 400 multiplications and 361 additions. Making use of the SIMD instructions, at best 4 of these multiplications can be done in parallel, adding the result of a multiplication is at best for free using the `mpya` instruction, so we need at least 100 instructions to compute the coefficients $r_0, \ldots, r_{38}$. Furthermore we need to multiply some intermediate products by 2, an effect resulting from the non-integer radix 12.75 used for the representation of finite field elements. As we assume the inputs to have reduced coefficients, all result coefficients r_i fit into 32-bit word elements.

We will now describe how the coefficients $r0, \ldots, r38$ can be computed using 145 pipeline-0 instructions (arithmetic instructions). This computation requires some rearrangement of coefficients in registers using the `shufb` instruction but with careful instruction scheduling and alignment these pipeline-1 instructions do not increase the number of cycles needed for multiplication. From the description of the arithmetic instructions it should be clear which rearrangement of inputs is necessary.

First use 15 `shl` instructions to have register variables
`b03s1` containing $b_0, b_1, b_2, 2b_3$,
`b03s2` containing $b_0, b_1, 2b_2, 2b_3$,
`b03s3` containing $b_0, 2b_1, 2b_2, 2b_3$,
`b47s1` containing $b_4, b_5, b_6, 2b_7$ and so on.

Now we can proceed producing intermediate result variables
`r03` containing $a_0b_0, a_0b_1, a_0b_2, a_0b_3$ (one `mpy` instruction),
`r14` containing $a_1b_0, a_1b_1, a_1b_2, 2a_1b_3$ (one `mpy` instruction),
`r25` containing $a_2b_0, a_2b_1, 2a_2b_2, 2a_2b_3$ (one `mpy` instruction),
`r36` containing $a_3b_0, 2a_3b_1, 2a_3b_2, 2a_3b_3$ (one `mpy` instruction),
`r47` containing $a_4b_0 + a_0b_4, a_4b_1 + a_0b_5, a_4b_2 + a_0b_6, a_4b_3 + a_0b_7$ (one `mpy` and one `mpya` instruction),
`r58` containing $a_5b_0 + a_1b_4, a_5b_1 + a_1b_5, a_5b_2 + a_1b_6, 2a_5b_3 + 2a_1b_7$ (one `mpy` and one `mpya` instruction) and so on. In total these computations need 36 `mpy` and 64 `mpya` instructions.

As a final step these intermediate results have to be joined to produce the coefficients $r_0, \ldots r_{38}$ in the register variables `r03`, `r47`,... `r3639`. We can do this using 30 additions if we first combine intermediate results using the `shufb` instruction. For example we join in one register variable the highest word of `r14` and the three lowest words of `r58` before adding this register variable to `r47`.

The basic idea for squaring is the same as for multiplication. We can make squaring slightly more efficient by exploiting the fact that some intermediate results are equal.

For a squaring of a value a given in reduced representation $(a_0, \ldots, a_{19})$, formulas for the result coefficients $r_0, \ldots, r_{38}$ are the following:

$$\begin{aligned}
r_0 =& a_0a_0\\
r_1 =& 2a_1a_0\\
r_2 =& 2a_2a_0 + a_1a_1\\
r_3 =& 2a_3a_0 + 2a_2a_1\\
r_4 =& 2a_4a_0 + 4a_3a_1 + 2a_2a_2\\
r_5 =& 2a_5a_0 + 2a_4a_1 + 4a_3a_2\\
r_6 =& 2a_6a_0 + 2a_5a_1 + 2a_4a_2 + 2a_3a_3\\
r_7 =& 2a_7a_0 + 2a_6a_1 + 2a_5a_2 + 2a_4a_3\\
r_8 =& 2a_8a_0 + 4a_7a_1 + 4a_6a_2 + 4a_5a_3 + a_4a_4\\
&\vdots
\end{aligned}$$

The main part of the computation only requires 60 multiplications (24 `mpya` and 36 `mpy` instructions). However, some partial results have to be multiplied by 4; this requires more preprocessing of the inputs, we end up using 35 instead of 15 `shl` instructions before entering the main block of multiplications. Squaring is therefore only 20 cycles faster than multiplication.

During both multiplication and squaring, we can overcome latencies by interleaving independent instructions.

5.3 Reduction

The task of the reduction step is to compute from the coefficients $r_0, \ldots r_{38}$ a reduced representation $(r_0, \ldots, r_{19})$. Implementing this computation efficiently is challenging in two ways: In a typical reduction chain every instruction is dependent on the result of the preceding instruction. This makes it very hard to vectorize operations in SIMD instructions and to hide latencies.

We will now describe a way to handle reduction hiding most instruction latencies but without data level parallelism through SIMD instructions.

The basic idea of reduction is to first reduce the coefficients r_{20} to r_{38} (producing a coefficient r_{39}), then add $19r_{20}$ to r_0, $19r_{21}$ to r_1 and so on until adding $19r_{39}$ to r_{19} and then reduce the coefficients r_0 to r_{19}.

Multiplications by 19 result from the fact, that the coefficient a_{20} stands for $a_{20} \cdot 2^{255}$ (see equation (1)). By the definition of the finite field $\mathbb{F}_{2^{255}-19}$, $2^{255}a_{20}$ is the same as $19a_{20}$. Equivalent statements hold for the coefficients $a_{21}, \ldots, a_{39}$.

The most speed critical parts of this reduction are the two carry chains from r_{20} to r_{39} and from r_0 to r_{19}. In order to overcome latencies in these chains we break each of them into four parallel carry chains, Algorithm 3 describes this structure of our modular reduction algorithm.

Algorithm 3 Structure of the modular reduction

```
Carry from $r_{20}$ to $r_{21}$, from $r_{24}$ to $r_{25}$, from $r_{28}$ to $r_{29}$ and from $r_{32}$ to $r_{33}$
Carry from $r_{21}$ to $r_{22}$, from $r_{25}$ to $r_{26}$, from $r_{29}$ to $r_{30}$ and from $r_{33}$ to $r_{34}$
Carry from $r_{22}$ to $r_{23}$, from $r_{26}$ to $r_{27}$, from $r_{30}$ to $r_{31}$ and from $r_{34}$ to $r_{35}$
Carry from $r_{23}$ to $r_{24}$, from $r_{27}$ to $r_{28}$, from $r_{31}$ to $r_{32}$ and from $r_{35}$ to $r_{36}$

Carry from $r_{24}$ to $r_{25}$, from $r_{28}$ to $r_{29}$, from $r_{32}$ to $r_{33}$ and from $r_{36}$ to $r_{37}$
Carry from $r_{25}$ to $r_{26}$, from $r_{29}$ to $r_{30}$, from $r_{33}$ to $r_{34}$ and from $r_{37}$ to $r_{38}$
Carry from $r_{26}$ to $r_{27}$, from $r_{30}$ to $r_{31}$, from $r_{34}$ to $r_{35}$ and from $r_{38}$ to $r_{39}$
Carry from $r_{27}$ to $r_{28}$, from $r_{31}$ to $r_{32}$ and from $r_{35}$ to $r_{36}$

Add $19r_{20}$ to $r_0$, add $19r_{21}$ to $r_1$, add $19r_{22}$ to $r_2$ and add $19r_{23}$ to $r_3$
Add $19r_{24}$ to $r_4$, add $19r_{25}$ to $r_5$, add $19r_{26}$ to $r_6$ and add $19r_{27}$ to $r_7$
Add $19r_{28}$ to $r_8$, add $19r_{29}$ to $r_9$, add $19r_{30}$ to $r_{10}$ and add $19r_{31}$ to $r_{11}$
Add $19r_{32}$ to $r_{12}$, add $19r_{33}$ to $r_{13}$, add $19r_{34}$ to $r_{14}$ and add $19r_{35}$ to $r_{15}$
Add $19r_{36}$ to $r_{16}$, add $19r_{37}$ to $r_{17}$, add $19r_{38}$ to $r_{18}$ and add $19r_{39}$ to $r_{19}$

Carry from $r_{16}$ to $r_{17}$, from $r_{17}$ to $r_{18}$, from $r_{18}$ to $r_{19}$ and from $r_{19}$ to $r_{20}$
Add $19r_{20}$ to $r_0$

Carry from $r_0$ to $r_1$, from $r_4$ to $r_5$, from $r_8$ to $r_9$ and from $r_{12}$ to $r_{13}$
Carry from $r_1$ to $r_2$, from $r_5$ to $r_6$, from $r_9$ to $r_{10}$ and from $r_{13}$ to $r_{14}$
Carry from $r_2$ to $r_3$, from $r_6$ to $r_7$, from $r_{10}$ to $r_{11}$ and from $r_{14}$ to $r_{15}$
Carry from $r_3$ to $r_4$, from $r_7$ to $r_8$, from $r_{11}$ to $r_{12}$ and from $r_{15}$ to $r_{16}$

Carry from $r_4$ to $r_5$, from $r_8$ to $r_9$, from $r_{12}$ to $r_{13}$ and from $r_{16}$ to $r_{17}$
Carry from $r_5$ to $r_6$, from $r_9$ to $r_{10}$, from $r_{13}$ to $r_{14}$ and from $r_{17}$ to $r_{18}$
Carry from $r_6$ to $r_7$, from $r_{10}$ to $r_{11}$, from $r_{14}$ to $r_{15}$ and from $r_{18}$ to $r_{19}$
Carry from $r_7$ to $r_8$, from $r_{11}$ to $r_{12}$ and from $r_{15}$ to $r_{16}$
```

Each of the carry operations in Algorithm 3 can be done using one `shufb`, one `rotmi` and one `a` instruction. Furthermore we need 8 masking instructions (bitwise `and`) for each of the two carry chains.

In total, a call to the multiplication function (including reduction) takes 444 cycles, a call to the squaring function takes 424 cycles. This includes 144 cycles for multiplication (124 cycles for squaring), 244 cycles for reduction and some more cycles to load input and store output. Furthermore the cost of a function call is included in these numbers.

Montgomery ladder step For the implementation of a Montgomery ladder step we exploit the fact that we can optimize a fixed sequence of arithmetic instructions in $\mathbb{F}_{2^{255}-19}$ instead of single instructions. This makes it much easier to make efficient use of the SIMD instruction set, in particular, for modular reduction.

The idea is to arrange the operations in $\mathbb{F}_{2^{255}-19}$ into blocks of 4 equal or similar instructions, similar meaning that multiplications and squarings can be grouped together and additions and subtractions can be grouped together as well. Then these operations can be carried out using the 4-way parallel SIMD instructions in the obvious way; for example for 4 multiplications $r = a \cdot b$, $s = c \cdot d$, $t = e \cdot f$ and $u = g \cdot h$ we first produce register variables `aceg0` containing in its word elements a_0, c_0, e_0, g_0 and `bdgh0` containing b_0, d_0, e_0, g_0 and so on. Then the first coefficient of r, s, t and u can be computed by applying the `mpy` instruction on `aceg0` and `bdfh0`. All other result coefficients of r, s, t and u can be computed in a similar way using `mpy` and `mpya` instructions.

This way of using the SIMD capabilities not only makes multiplication slightly faster (420 arithmetic instructions instead of 576 for 4 multiplications), it also allows for much faster reduction: The reduction algorithm described above can now be applied to 4 results in parallel, reducing the cost of a reduction by a factor of 4.

In Algorithm 4 we describe how we divide a Montgomery ladder step into blocks of 4 similar operations. In this algorithm the computation of Z_{P+Q} in the last step requires one multiplication and reduction which we carry out as described in the previous section. The computation of a ladder step again requires rearrangement of data in registers using the `shufb` instruction. Again we can hide these pipeline-1 instructions almost entirely by interleaving with arithmetic pipeline-1 instructions.

One remark regarding subtractions occurring in this computation: As reduction expects all coefficients to be larger than zero, we cannot just compute the difference of each coefficient. Instead, for the subtraction $a - b$ we first add $2p$ to a and then subtract b. For blocks containing additions and subtractions in Algorithm 4 we compute the additions together with additions of $2p$ and perform the subtraction in a separate step.

In total one call to the ladder-step function takes 2433 cycles.

6 Results and Comparison

6.1 Benchmarking Methodology

In order to make our benchmarking results comparable and verifiable we use the SUPERCOP toolkit, a benchmarking framework developed within eBACS, the benchmarking project of ECRYPT II [4]. The software presented in this paper passes the extensive tests of this toolkit showing compatibility to other `curve25519` implementations, in particular the reference implementation included in the toolkit.

Algorithm 4 Structure of a Montgomery ladder step (see Algorithm 2) optimized for 4-way parallel computation

```
$t_1 \leftarrow X_P + Z_P$
$t_2 \leftarrow X_P - Z_P$
$t_3 \leftarrow X_Q + Z_Q$
$t_4 \leftarrow X_Q - Z_Q$
Reduce $t_1, t_2, t_2, t_3$

$t_6 \leftarrow t_1^2$
$t_7 \leftarrow t_2^2$
$t_8 \leftarrow t_4 \cdot t_1$
$t_9 \leftarrow t_3 \cdot t_2$
Reduce $t_6, t_7, t_8, t_9$

$t_{10} = a24 \cdot t_6$
$t_{11} = (a24 - 1) \cdot t_7$

$t_5 \leftarrow t_6 - t_7$
$t_4 \leftarrow t_{10} - t_{11}$
$t_1 \leftarrow t_8 - t_9$
$t_0 \leftarrow t_8 + t_9$
Reduce $t_5, t_4, t_1, t_0$

$Z_{[2]P} \leftarrow t_5 \cdot t_4$
$X_{P+Q} \leftarrow t_0^2$
$X_{[2]P} \leftarrow t_6 \cdot t_7$
$t_2 \leftarrow t_1^2$
Reduce $Z_{[2]P}, X_{P+Q}, X_{[2]P}, t_2$

$Z_{P+Q} \leftarrow X_{Q-P} \cdot t_2$
Reduce $Z_{P+Q}$
```

For scalar multiplication software, SUPERCOP measures two different cycle counts: The `crypto_scalarmult` benchmark measures cycles for a scalar multiplication of an arbitrary point; the `crypto_scalarmult_base` benchmark measures cycles needed for a scalar multiplication of a fixed base point.
We currently implement `crypto_scalarmult_base` as `crypto_scalarmult`; faster implementations would be useful in applications that frequently call
`crypto_scalarmult_base`

Two further benchmarks regard our `curve25519` software in the context of Diffie-Hellman key exchange: The `crypto_dh_keypair` benchmark measures the number of cycles to generate a key pair consisting of a secret and a public key. The `crypto_dh` benchmark measures cycles to compute a joint key, given a secret and a public key.

6.2 Results

We benchmarked our software on a computer named hex01, a QS21 blade at the Chair for Operating Systems at RWTH Aachen University. It contains two 3200MHz Cell Broadband Engine processors (revision 5.1); measurements used one SPU of one CBE:

SUPERCOP benchmark on hex01	number of cycles
crypto_scalarmult	777000
crypto_scalarmult_base	777000
crypto_dh_keypair	804000
crypto_dh	780720

We furthermore benchmarked the software on a computer named node033, a QS22 blade at the Research Center Jülich. It contains two PowerXCell 8i processors (Cell Broadband Engine (revision 48.0)); measurements used one SPU of one CBE:

SUPERCOP benchmark on node033	number of cycles
crypto_scalarmult	777000
crypto_scalarmult_base	777000
crypto_dh_keypair	804000
crypto_dh	780840

We also benchmarked our software on a computer named cosmovoid, a Sony Playstation 3 at the Chair for Operating Systems at RWTH Aachen University. It has a 3192MHz Cell Broadband Engine processor (revision 5.1); measurements used one SPU of this CBE.

SUPERCOP benchmark on cosmovoid	number of cycles
crypto_scalarmult	777040
crypto_scalarmult_base	777040
crypto_dh_keypair	803960
crypto_dh	780720

6.3 Comparison

To compare our results on a cost-performance basis with ECDH software for other platforms, we consider the cheapest hardware configuration containing a Cell Broadband Engine, namely the Sony Playstation 3, and compare the results to an Intel-Core-2-based configuration running the ECDH software presented in [8]. This is the currently fastest implementation of ECDH for the Core 2 processor providing a similar security as curve25519. Note that this software is not protected against timing attacks.

We benchmarked this software on a machine called archer. at the National Taiwan University. This machine has a 2500MHz Intel Core 2 Q9300 processor with 4 cores; measurements used one core.

SUPERCOP reports 365363 cycles for the `crypto_dh` benchmark (this software is not benchmarked as scalar-multiplication software). Key-pair generation specializes the scalar multiplication algorithm for the known basepoint; the `crypto_dh_keypair` benchmark reports 151215 cycles.

For the Core-2-based configuration we determined which processor (as part of a workstation) yields the best cost-performance ratio. Currently this is the Q9300, a quad-core processor operating at 2.5 GHz. To estimate a price for a complete workstation including this processor we determined the lowest prices for processor, case, motherboard, memory, hard disk and power supply from the online retailer zipzoomfly.com yielding $402.66.

To determine the best price for the Sony Playstation 3 we used Google Product Search. The currently (Jan 6, 2009) cheapest offer is $279 for the Playstation 3 with a 40 GB hard disk.

The Sony Playstation 3 makes 6 SPUs available for general purpose computations. Using our implementation running at 780720 cycles (`crypto_dh` on `cosmovoid`) on 6 SPUs operating at 3192MHz yields 24528 `curve25519` computations per second. Taking the $279 market price for the Playstation as a basis, the cheapest CBE-based hardware can thus perform 87 computations of `curve25519` per second per dollar.

The Q9300-based workstation has 4 cores operating at 2.5GHz, using the above-mentioned implementation which takes 365363 cycles, we can thus perform 27368 joint-key computations per second. Taking $400 market price for a Q9300-based workstation as a basis, the cheapest Core-2-based hardware can thus perform 68 joint-key computations per second per dollar.

Note, that this comparison is not fair in several ways: The cheapest Q9300-based workstation has for example more memory than the Playstation 3 (1GB instead of 256MB) and a larger hard disk (160GB instead of 40GB).

On the other hand we only use the 6 SPUs of the CBE for the `curve25519` computation, the PPU is still available for other tasks, whereas the performance estimation for the Core-2-based system assumes 100% workload on all CPU cores.

In any case it is clear that the Cell Broadband Engine, when used properly, is one of the best available CPUs for public-key cryptography.

References

1. Wesley Alvaro, Jakub Kurzak, and Jack Dongarra. Fast and small short vector SIMD matrix multiplication kernels for the synergistic processing element of the CELL processor. In *Computational Science – ICCS 2008)*, volume 5101 of *Lecture Notes in Computer Science*, pages 935–944. Springer, 2008.
2. Daniel J. Bernstein. qhasm: tools to help write high-speed software. `http://cr.yp.to/qhasm.html` (accessed Jan 1, 2009).
3. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2005.

4. Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT benchmarking of cryptographic systems, Nov 2008. `http://bench.cr.yp.to/` (accessed Jan 1, 2009).
5. Neil Costigan and Michael Scott. Accelerating SSL using the vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3. In *Proceedings of SPEED workshop*, 2007. `http://www.hyperelliptic.org/SPEED/`.
6. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov 1976. `http://citeseer.ist.psu.edu/diffie76new.html`.
7. Junfeng Fan, Kazuo Sakiyama, and Ingrid Verbauwhede. Elliptic curve cryptography on embedded multicore systems. In *Workshop on Embedded Systems Security - WESS 2007*, pages 17–22, Salzburg,Austria, 2007.
8. Stephen D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on general curves, 2008. `http://eprint.iacr.org/2008/194`.
9. P. Gaudry and E. Thomé. The $\mathtt{mp}\mathbb{F}_q$ library and implementing curve-based key exchanges. In *Proceedings of SPEED workshop*, 2007. `http://www.loria.fr/~gaudry/publis/mpfq.pdf`.
10. Darrel Hankerson, Alfred J. Menezes, and Scott A. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, Berlin, 2003.
11. IBM DeveloperWorks. Cell broadband engine architecture and its first implementation, Nov 2005. `http://www-128.ibm.com/developerworks/power/library/pa-cellperf/`.
12. IBM DeveloperWorks. Cell broadband engine programming handbook (version 1.1), April 2007. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/9F820A5FFA3ECE8C8725716A0062585F`.
13. IBM DeveloperWorks. SPU assembly language specification (version 1.6), Sep 2007. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/EFA2B196893B550787257060006FC9FB`.
14. IBM DeveloperWorks. Example library API reference (version 3.1), Sep 2008. `http://www.ibm.com/developerworks/power/cell/documents.html`.
15. IBM DeveloperWorks. SPE cryptographic library user documentation 1.0, Sep 2008. `http://www.ibm.com/developerworks/power/cell/documents.html`.
16. Peter. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
17. Kanna Shimizu, Daniel Brokenshire, and Mohammad Peyravian. Cell Broadband Engine support for privacy, security, and digital rights management applications. White paper, IBM, Oct 2005. `http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3F88DA69A1C0AC40872570AB00570985`.
18. Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today, Dec 2008. `http://www.win.tue.nl/hashclash/rogue-ca/` (accessed Jan 4, 2009).
19. Marc Stevens, Arjen Lenstra, and Benne de Weger. Nostradamus – predicting the winner of the 2008 US presidential elections using a Sony PlayStation 3, Nov 2007. `http://www.win.tue.nl/hashclash/Nostradamus/` (accessed Jan 4, 2009).