Fast elliptic-curve cryptography on the Cell Broadband Engine

Neil Costigan (DCU), **Peter Schwabe** (TU/e)

Eindhoven University of Technology

June 24, 2009

Africacrypt 2009, Gammarth

The Cell Broadband Engine

Some general information

- Processor jointly developed by Sony, Toshiba and IBM
- Runs in Playstation 3, QS20 and QS21 blades, supercomputers (Roadrunner), extension cards
- 1 Power G5 core and 8 (6) Synergistic Processor Units (SPU)
- Clock frequency of 3.2 GHz

The Cell Broadband Engine

Some general information

- Processor jointly developed by Sony, Toshiba and IBM
- Runs in Playstation 3, QS20 and QS21 blades, supercomputers (Roadrunner), extension cards
- 1 Power G5 core and 8 (6) Synergistic Processor Units (SPU)
- Clock frequency of 3.2 GHz

Cryptography on the CBE

- Cluster of 200 Playstations (1200 SPUs) at EPFL has been used to find MD5 collisions (best paper award, CRYPTO '09)
- Existing fast implementations of secret-key primitives, e.g. AES-ECB encrypt: 12.43 cycles/byte on one SPU

How about public-key crypto?

- RSA-1024 enc. or dec.: 4,074,000 cycles [Shimizu et al. 2005]
- DSA-1024 key generation: 1,331,000 cycles [Shimizu et al. 2005]
- DSA-1024 sig. generation: 2,250,000 cycles [Shimizu et al. 2005]
- DSA-1024 sig. verification: 4,375,000 cycles [Shimizu et al. 2005]
- RSA-2048 sig. generation: 50,035,200 cycles [Costigan, Scott 2007]

How about public-key crypto?

- RSA-1024 enc. or dec.: 4,074,000 cycles [Shimizu et al. 2005]
- DSA-1024 key generation: 1,331,000 cycles [Shimizu et al. 2005]
- DSA-1024 sig. generation: 2,250,000 cycles [Shimizu et al. 2005]
- DSA-1024 sig. verification: 4,375,000 cycles [Shimizu et al. 2005]
- RSA-2048 sig. generation: 50,035,200 cycles [Costigan, Scott 2007]

As a comparison: Some numbers for some Core 2 (Q9550)

- `curve25519`: 384,192 cycles for 255-bit ECDH joint key [Gaudry, Thomé]
- `gls2127`: 318,019 cycles for 256-bit ECDH joint key [Galbraith, Lin, Scott]

Why is the Cell doing so bad?

Obvious: Comparing apples with oranges

- Modular arithmetic vs. elliptic-curve cryptography
- Signing/encrypting vs. key exchange

Why is the Cell doing so bad?

Obvious: Comparing apples with oranges

- Modular arithmetic vs. elliptic-curve cryptography
- Signing/encrypting vs. key exchange

Some less obvious reasons

- Public-key crypto usually relies on large-integer arithmetic
- Performance usally bottlenecked by multiplications and squarings
- Core 2 supports multiplication of 64-bit integers
- Cell only supports multiplication of 16-bit integers
- Apparently the Cell is just a bad platform for PKC?

Why is the Cell doing so bad?

Obvious: Comparing apples with oranges

- Modular arithmetic vs. elliptic-curve cryptography
- Signing/encrypting vs. key exchange

Some less obvious reasons

- Public-key crypto usually relies on large-integer arithmetic
- Performance usally bottlenecked by multiplications and squarings
- Core 2 supports multiplication of 64-bit integers
- Cell only supports multiplication of 16-bit integers
- Apparently the Cell is just a bad platform for PKC?

Let's try: Implement fast elliptic-curve cryptography for the CBE.

The `curve25519` ECDH software

- `curve25519` was introduced by Bernstein in 2006
- Setting speed records on various platforms
- Uses Montgomery curve $E : y^2 = x^3 + 486662x^2 + x$ over the field $\mathbb{F}_{2^{255}-19}$
- Two parts: 255-step Montgomery ladder for scalar multiplication and a field inversion
- One ladder step requires: 5 multiplications, 4 squarings, 8 additions and 1 multiplication with a constant
- In total: 1276 multiplications, 1020 squarings, 255 multiplications with a constant, 2040 additions and 1 inversion

Making it fast – a first approach

Standard approach: ECC as pyramid of

- Elliptic-curve scalar multiplication,
- Point addition and doubling,
- Modular operations in $\mathbb{F}_p$,
- Instructions on a w-bit core.

Making it fast – a first approach

Standard approach: ECC as pyramid of

- Elliptic-curve scalar multiplication (Montgomery ladder),
- Point addition and doubling,
- Modular operations in $\mathbb{F}_p$,
- Instructions on a w-bit core.

Making it fast – a first approach

Standard approach: ECC as pyramid of

- Elliptic-curve scalar multiplication (Montgomery ladder),
- Point addition and doubling (Ladder steps using XZ-coordinates),
- Modular operations in $\mathbb{F}_p$,
- Instructions on a w-bit core.

Making it fast – a first approach

Standard approach: ECC as pyramid of

- Elliptic-curve scalar multiplication (Montgomery ladder),
- Point addition and doubling (Ladder steps using XZ-coordinates),
- Modular operations in $\mathbb{F}_p$,
- Instructions on a w-bit core (SPU instruction set).

Making it fast – a first approach

Standard approach: ECC as pyramid of

- Elliptic-curve scalar multiplication (Montgomery ladder),
- Point addition and doubling (Ladder steps using XZ-coordinates),
- Modular operations in $\mathbb{F}_p$ ...use some library?,
- Instructions on a w-bit core (SPU instruction set).

Making it fast – a first approach

Standard approach: ECC as pyramid of

- Elliptic-curve scalar multiplication (Montgomery ladder),
- Point addition and doubling (Ladder steps using XZ-coordinates),
- Modular operations in $\mathbb{F}_p$ ...use some library?,
- Instructions on a w-bit core (SPU instruction set).

IBM's MPM library

- Multiprecison Math Library for the Cell
- "Big-integer" support optimized for the SPU
- Supports Montgomery modular multiplication

What speed can we get with MPM?

Benchmarks of modular arithmetic

Operation	Number of cycles
Addition/Subtraction	86
Montgomery Mul. (original MPM)	1197
Montgomery Mul. (optimized for 256-bit numbers)	892

$\Longrightarrow$ at least 2227040 cycles (1276M + 1020S + 2040A)
Ignoring cost for inversion and multiplication with constant

What speed can we get with MPM?

Benchmarks of modular arithmetic

Operation	Number of cycles
Addition/Subtraction	86
Montgomery Mul. (original MPM)	1197
Montgomery Mul. (optimized for 256-bit numbers)	892

$\Longrightarrow$ at least 2227040 cycles (1276M + 1020S + 2040A)
Ignoring cost for inversion and multiplication with constant

Benchmarks of big-integer arithmetic

Operation	Number of cycles
Addition/Subtraction	52
Multiplication (original MPM)	594
Multiplication (optimized for 256-bit numbers)	360

$\Longrightarrow$ at least 934080 cycles (1276M + 1020S + 2040A)

A closer look at the SPU

- 128 registers of width 128 bit
- All (arithmetic) instructions are SIMD
  - $16\times$ 8 bit
  - $8\times$ 16 bit
  - $4\times$ 32 bit
  - Exception: Multiplication is $4\times$ 16 bit, 32-bit results
  - Can do multiplication and addition (`muladd`) in one instruction
- At most one arithmetic instruction per cycle
- Additional load/store/shuffle instruction per cycle
- Fully in-order execution
- Relevant instruction latencies between 2 and 7 (mostly 4)

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

The standard approach

- 255-bit numbers, 128-bit registers $\Rightarrow$ use 2 registers

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

The standard approach

- 255-bit numbers, 128-bit registers $\Rightarrow$ use 2 registers
- Schoolbook multiplication then requires at least
  - 256 multiplications (64 instructions)
  - 224 (carry-extended) additions (58 instructions)
  - 224 carry generates (58 instructions)
  - Quite a bit of shifting/shuffling

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

The standard approach

- 255-bit numbers, 128-bit registers $\Rightarrow$ use 2 registers
- Schoolbook multiplication then requires at least
  - 256 multiplications (64 instructions)
  - 224 (carry-extended) additions (58 instructions)
  - 224 carry generates (58 instructions)
  - Quite a bit of shifting/shuffling
- Most of the time we are not multiplying
- Huge effort to handle carry bits
- Huge effort to move partial results around
- Situation is similar for other multiplication algorithms

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

Redundant representation

- Represent an element $a \in \mathbb{F}_{2^{255}-19}$ as $(a_0, \ldots, a_{19})$ where

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

- We call a coefficent a_i reduced, if $a_i \in [0, 2^{13} - 1]$
- We call $a \in \mathbb{F}_{2^{255}-19}$ reduced if all coefficients are reduced

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

Redundant representation

- Represent an element $a \in \mathbb{F}_{2^{255}-19}$ as $(a_0, \ldots, a_{19})$ where

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

- We call a coefficent a_i reduced, if $a_i \in [0, 2^{13} - 1]$
- We call $a \in \mathbb{F}_{2^{255}-19}$ reduced if all coefficients are reduced
- Multiplication only needs 100 `mul`/`muladd` instructions

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

Redundant representation

- Represent an element $a \in \mathbb{F}_{2^{255}-19}$ as $(a_0, \dots, a_{19})$ where

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

- We call a coefficent a_i reduced, if $a_i \in [0, 2^{13} - 1]$
- We call $a \in \mathbb{F}_{2^{255}-19}$ reduced if all coefficients are reduced
- Multiplication only needs 100 `mul`/`muladd` instructions
- ... plus some overhead from non-integer radix
- ... plus some overhead to construct final result $(r_0, \dots, r_{38})$
- In total: 145 arithmetic instructions, 145 cycles

Hiding latencies during reduction

- During multiplication using SIMD and hiding latencies is easy
- Reduction: non-reduced $(r_0, \ldots, r_{38}) \longrightarrow$ reduced $(r_0, \ldots, r_{19})$

Hiding latencies during reduction

- During multiplication using SIMD and hiding latencies is easy
- Reduction: non-reduced $(r_0, \ldots, r_{38}) \longrightarrow$ reduced $(r_0, \ldots, r_{19})$

Standard reduction chain

- Carry from r_{20} to $r_{21}, \ldots$ from r_{38} to r_{39}
- Reduce "polynomial"
- Carry from r_0 to r_1 etc.
- Problem: Each instruction depends on result from previous instruction
- Just do arithmetic about every 4th cycle
- Cannot use SIMD capabilities

Hiding latencies during reduction

Interleaved reduction

- Four independet parallel reduction chains
- Carry $r_{20} \rightarrow r_{21}$, $r_{24} \rightarrow r_{25}$, $r_{28} \rightarrow r_{29}$, $r_{32} \rightarrow r_{33}$

Hiding latencies during reduction

Interleaved reduction

- Four independet parallel reduction chains
- Carry $r_{20} \to r_{21}$, $r_{24} \to r_{25}$, $r_{28} \to r_{29}$, $r_{32} \to r_{33}$
- Carry $r_{21} \to r_{22}$, $r_{25} \to r_{26}$, $r_{29} \to r_{30}$, $r_{33} \to r_{34}$
- ...

Hiding latencies during reduction

Interleaved reduction

- Four independet parallel reduction chains
- Carry $r_{20} \rightarrow r_{21}$, $r_{24} \rightarrow r_{25}$, $r_{28} \rightarrow r_{29}$, $r_{32} \rightarrow r_{33}$
- Carry $r_{21} \rightarrow r_{22}$, $r_{25} \rightarrow r_{26}$, $r_{29} \rightarrow r_{30}$, $r_{33} \rightarrow r_{34}$
- ...
- Carry $r_{24} \rightarrow r_{25}$, $r_{28} \rightarrow r_{29}$, $r_{32} \rightarrow r_{33}$, $r_{36} \rightarrow r_{37}$
- ...

Hiding latencies during reduction

Interleaved reduction

- Four independet parallel reduction chains
- Carry $r_{20} \rightarrow r_{21}$, $r_{24} \rightarrow r_{25}$, $r_{28} \rightarrow r_{29}$, $r_{32} \rightarrow r_{33}$
- Carry $r_{21} \rightarrow r_{22}$, $r_{25} \rightarrow r_{26}$, $r_{29} \rightarrow r_{30}$, $r_{33} \rightarrow r_{34}$
- ...
- Carry $r_{24} \rightarrow r_{25}$, $r_{28} \rightarrow r_{29}$, $r_{32} \rightarrow r_{33}$, $r_{36} \rightarrow r_{37}$
- ...
- Looks stupid (increasing reduction steps from 20 to 32)
- But: Do arithmetic every cycle, increase speed by a factor of $4 \cdot 20/32 = 2.5$

Hiding latencies during reduction

Interleaved reduction

- Four independet parallel reduction chains
- Carry $r_{20} \to r_{21}$, $r_{24} \to r_{25}$, $r_{28} \to r_{29}$, $r_{32} \to r_{33}$
- Carry $r_{21} \to r_{22}$, $r_{25} \to r_{26}$, $r_{29} \to r_{30}$, $r_{33} \to r_{34}$
- $\dots$
- Carry $r_{24} \to r_{25}$, $r_{28} \to r_{29}$, $r_{32} \to r_{33}$, $r_{36} \to r_{37}$
- $\dots$
- Looks stupid (increasing reduction steps from 20 to 32)
- But: Do arithmetic every cycle, increase speed by a factor of $4 \cdot 20/32 = 2.5$
- We are still not using SIMD capabilities!

Optimize EC instead of GF arithmetic

- Consider *sequences* of finite-field operations instead of single operations
- Here: Optimize Montgomery ladder step
  - Group $2\times$ 4 multiplications together (squarings as multiplications)
  - Group additions/subtractions in blocks of 4
  - Do “digit slicing” [Grabher, Großschädl, Page, 2008]
  - Leaves just one single multiplication at the end

Optimize EC instead of GF arithmetic

- Consider *sequences* of finite-field operations instead of single operations
- Here: Optimize Montgomery ladder step
  - Group $2\times$ 4 multiplications together (squarings as multiplications)
  - Group additions/subtractions in blocks of 4
  - Do "digit slicing" [Grabher, Großschädl, Page, 2008]
  - Leaves just one single multiplication at the end
- Reduces number of arithmetic instructions for 4 multiplications from 580 to 420

Optimize EC instead of GF arithmetic

- Consider *sequences* of finite-field operations instead of single operations
- Here: Optimize Montgomery ladder step
  - Group $2\times$ 4 multiplications together (squarings as multiplications)
  - Group additions/subtractions in blocks of 4
  - Do “digit slicing” [Grabher, Großschädl, Page, 2008]
  - Leaves just one single multiplication at the end
- Reduces number of arithmetic instructions for 4 multiplications from 580 to 420
- Uses SIMD for reduction: Increasing speed by a factor of 4!

Results

- Benchmarks used SUPERCOP
- hex01 is a QS21 blade at the Chair for Operating Systems, RWTH Aachen
- cosmovoid is a Playstation 3 at the Chair for Operating Systems, RWTH Aachen
- node001 is a QS22 blade at Research Center Jülich

SUPERCOP benchmark	hex01	node001	cosmovoid
crypto_scalarmult	697080	697080	697040
crypto_scalarmult_base	697080	697080	697080
crypto_dh_keypair	720120	720120	720200
crypto_dh	697080	697080	697040

- Including costs for key verification and key compression
- Constant time – protected against timing attacks

"Comparison"

- `gls1271` on a Q9550: 318019 cycles
- `curve25519` on a Q9550: 384192 cycles
- `curve25519` on a CBE: 697080 cycles

“Comparison”

- `gls1271` on a Q9550: 318019 cycles
- `curve25519` on a Q9550: 384192 cycles
- `curve25519` on a CBE: 697080 cycles
- Q9550 has 4 cores at 2.83GHz, CBE has 8 (6) cores at 3.2GHz

"Comparison"

- gls1271 on a Q9550: 318019 cycles
- curve25519 on a Q9550: 384192 cycles
- curve25519 on a CBE: 697080 cycles
- Q9550 has 4 cores at 2.83GHz, CBE has 8 (6) cores at 3.2GHz
- gls1271 on a Q9550: 38220 ECDH/second
- curve25519 on a Q9550: 31637 ECDH/second
- curve25519 on a CBE: 39432 (29574) ECDH/second

Some more information

- Software is public domain
- Software: http://www.cryptojedi.org/crypto/#celldh
- SUPERCOP: http://bench.cr.yp.to/