Who is afraid of vectors?

Optimizing cryptography using SSE, AVX, NEON and Co.

Peter Schwabe

Radboud University Nijmegen, The Netherlands

August 26, 2013

Microsoft Research

"Thus we arbitrarily select a reference organization : the IBM 704-70927090. This organization is then regarded as the prototype of the class of machines which we label:
1) Single Instruction Stream–Single Data Stream (SISD).

Three additional organizational classes are evident.
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD)"

– Michael J. Flynn. Very high-speed computing systems. 1966.

SISD
Example: 32-bit integer addition

```
int64 a
int64 b
a = mem32[addr1 + 0]
b = mem32[addr2 + 0]
(uint32) a += b
mem32[addr3 + 0] = a
```

SIMD with vector instructions

Example: 4 32-bit integer additions

```
reg128 a
reg128 b
a = mem128[addr1 + 0]
b = mem128[addr2 + 0]
4x a += b
mem128[addr3 + 0] = a
```

Why would you care?

- Consider the Intel Nehalem processor

Why would you care?

- Consider the Intel Nehalem processor
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle

Why would you care?

- Consider the Intel Nehalem processor
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - 4× 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle

Why would you care?

- Consider the Intel Nehalem processor
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - 4$\times$ 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do 4$\times$ the work**

Why would you care?

- Consider the Intel Nehalem processor
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - 4× 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do 4× the work**
- Situation on other architectures/microarchitectures is similar

Why would you care? (Part II)

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups

Why would you care? (Part II)

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups
- Secret-data-dependent branches and secret branch conditions are the major sources of timing-attack vulnerabilities

Why would you care? (Part II)

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups
- Secret-data-dependent branches and secret branch conditions are the major sources of timing-attack vulnerabilities
- Strong synergies between speeding up code with vector instructions and protecting code!

Multiple Data Streams

- Where does the data-level parallelism come from?
- Easy case: High-level batching
  - Encrypt n messages instead of one message
  - Compute n signatures instead of one
  - Do cryptanalysis

Multiple Data Streams

- Where does the data-level parallelism come from?
- Easy case: High-level batching
  - Encrypt n messages instead of one message
  - Compute n signatures instead of one
  - Do cryptanalysis
  - Requires rewriting (interleaving) basic data structures, e.g.:

```
typedef struct{           typedef struct{
  uint64_t x[4];            uint64_t x[16];
} bigint256;              } bigint256x4;

bigint256 a,b,c,d;        // (a[0],b[0],c[0],d[0],a[1],...,d[3])
                          bigint256x4 abcd;
```

Multiple Data Streams

- Where does the data-level parallelism come from?
- Easy case: High-level batching
  - Encrypt n messages instead of one message
  - Compute n signatures instead of one
  - Do cryptanalysis
  - Requires rewriting (interleaving) basic data structures, e.g.:

```
typedef struct{          typedef struct{
  uint64_t x[4];           uint64_t x[16];
} bigint256;             } bigint256x4;

bigint256 a,b,c,d;       // (a[0],b[0],c[0],d[0],a[1],...,d[3])
                         bigint256x4 abcd;
```

- Harder: Exploit parallelism *inside one* computation
- This is the topic of this talk

Salsa20 in NEON

- Joint work with Dan Bernstein (CHES 2012)
- NEON:
  - Vector instruction set of ARMv7 processors
  - 16 128-bit vector registers (e.g.)
  - On Cortex-A8: At most one arithmetic instruction, one load/store/shuffle instruction per cycle
- Salsa20:
  - Stream cipher designed by Bernstein in 2005
  - In the eSTREAM software portfolio
  - Generates stream in 64-byte blocks, works on 32-bit integers
  - Per block: 20 rounds; each round doing 16 add-rotate-xor sequences, such as

    ```
    s4 = x0 + x12
    x4 ^= (s4 >>> 25)
    ```

  - These sequences are 4-way parallel!

A first approach

- Per round do $4\times$ something like:

```
4x a0 = diag1 + diag0
4x b0 = a0 << 7
4x a0 unsigned >>= 25
   diag3 ^= b0
   diag3 ^= a0
```

- $+$ some (free) shuffles

A first approach

- Per round do $4\times$ something like:

```
4x a0 = diag1 + diag0
4x b0 = a0 << 7
4x a0 unsigned >>= 25
   diag3 ^= b0
   diag3 ^= a0
```

- $+$ some (free) shuffles
- Intuitive cycle lower bound:
  $(5 \cdot 4 \cdot 20)/64 = 6.25$ cycles/byte

A first approach

- Per round do $4\times$ something like:

```
4x a0 = diag1 + diag0
4x b0 = a0 << 7
4x a0 unsigned >>= 25
   diag3 ^= b0
   diag3 ^= a0
```

- $+$ some (free) shuffles
- Intuitive cycle lower bound:
  $(5 \cdot 4 \cdot 20)/64 = 6.25$ cycles/byte
- Problem: The above sequence has a 9-cycle latency, thus:
  $(9 \cdot 4 \cdot 20)/64 = 11.25$ cycles/byte

Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism

Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism
- Good for pipelined and superscalar execution

Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism
- Good for pipelined and superscalar execution
- The vector implementation needs 4-way data parallelism, there is (almost) no instruction-level parallelism left
- Bad for pipelined and superscalar execution

Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism
- Good for pipelined and superscalar execution
- The vector implementation needs 4-way data parallelism, there is (almost) no instruction-level parallelism left
- Bad for pipelined and superscalar execution
- Idea: Blocks are independent, use this to re-introduce instruction-level parallelism

Trading parallelism

- Salsa20 rounds have 4-way data-level parallelism
- In a scalar implementations this turns into 4-way instruction-level parallelism
- Good for pipelined and superscalar execution
- The vector implementation needs 4-way data parallelism, there is (almost) no instruction-level parallelism left
- Bad for pipelined and superscalar execution
- Idea: Blocks are independent, use this to re-introduce instruction-level parallelism
- Lower bound when interleaving 2 blocks: 6.875 cycles/byte
- Lower bound when interleaving 3 blocks: 6.25 cycles/byte

Going even further

- NEON is basically a coprocessor to the ARM core
- ARM decodes instructions, forwards NEON instructions to the NEON unit

Going even further

- NEON is basically a coprocessor to the ARM core
- ARM decodes instructions, forwards NEON instructions to the NEON unit
- Idea: Also keep the ARM core busy with Salsa20 computations
- New bottleneck: ARM core decodes at most 2 instructions per cycle

Going even further

- NEON is basically a coprocessor to the ARM core
- ARM decodes instructions, forwards NEON instructions to the NEON unit
- Idea: Also keep the ARM core busy with Salsa20 computations
- New bottleneck: ARM core decodes at most 2 instructions per cycle
- Add-rotate-xor is only 2 ARM instructions
- Best tradeoff: One block on ARM, two blocks on NEON

A flavor of the code

```
4x a0 = diag1 + diag0
    4x next_a0 = next_diag1 + next_diag0
            s4 = x0 + x12
            s9 = x5 + x1
4x b0 = a0 << 7
    4x next_b0 = next_a0 << 7
4x a0 unsigned>>= 25
    4x next_a0 unsigned>>= 25
            x4 ^= (s4 >>> 25)
            x9 ^= (s9 >>> 25)
            s8 = x4 + x0
            s13 = x9 + x5
   diag3 ^= b0
       next_diag3 ^= next_b0
   diag3 ^= a0
       next_diag3 ^= next_a0
            x8 ^= (s8 >>> 23)
            x13 ^= (s13 >>> 23)
```

Result

5.47 cycles/byte for Salsa20 encryption on ARM Cortex-A8 with NEON

http://cryptojedi.org/crypto/#neoncrypto

ECDH on the Cell Broadband Engine

- Joint work with Neil Costigan (Africacrypt 2009)
- Cell Broadband Engine (CBE):
  - Processor in the PS 3 and in IBM Cell Blades
  - Has one Power G5 core and 8 (6) "Synergistic Processor Units" (SPUs)
  - SPU: all instructions are vector instructions, 128 128-bit registers
  - At most one arithmetic instruction, one load/store/shuffle instruction per cycle
  - Largest multiplier: $16 \times 16 \rightarrow 32$ bits (4-way parallel)
- Curve25519
  - Elliptic-curve DH key exchange proposed by Bernstein in 2006
  - Uses Montgomery curve over $\mathbb{F}_{2^{255}-19}$
  - Main computation: 255 Montgomery ladder steps, each with 5M$+4$S$+8$A$+1$d

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

- Intuitive: Use 16 16-bit integers $a_0, \dots, a_{15}$ in 2 registers to represent

$$A = \sum_{i=0}^{15} a_i 2^{16\cdot i}$$

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

- Intuitive: Use 16 16-bit integers $a_0, \ldots, a_{15}$ in 2 registers to represent
$$A = \sum_{i=0}^{15} a_i 2^{16 \cdot i}$$
- Schoolbook multiplication gives 256 16×16-bit multiplications, 224 32-bit additions

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

- Intuitive: Use 16 16-bit integers $a_0, \dots, a_{15}$ in 2 registers to represent
$$A = \sum_{i=0}^{15} a_i 2^{16 \cdot i}$$
- Schoolbook multiplication gives 256 16×16-bit multiplications, 224 32-bit additions
- But wait, what happens to carries?
    - Answer on the Cell SPU: extra instruction (and `muladd` cannot add in carries)
    - Answer for most other vector instruction sets: they're gone

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

- Intuitive: Use 16 16-bit integers $a_0, \dots, a_{15}$ in 2 registers to represent

$$A = \sum_{i=0}^{15} a_i 2^{16 \cdot i}$$

- Schoolbook multiplication gives 256 16×16-bit multiplications, 224 32-bit additions
- But wait, what happens to carries?
  - Answer on the Cell SPU: extra instruction (and `muladd` cannot add in carries)
  - Answer for most other vector instruction sets: they're gone
- Carry-safe representation: Use $(a_0, \dots, a_{19})$ with

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

Multiplication in carry-safe representation

- Start with *reduced* elements $(a_0, \ldots, a_{19})$ and $(b_0, \ldots, b_{19})$, i.e.,

$$a_i, b_i \in [0, 2^{13} - 1], \quad i = 0, \ldots, 19$$

- Use 100 `mul` and `muladd` instructions to produce result $(r_0, \ldots, r_{38})$,

$$r_i \in [0, 2^{32} - 1], \quad i = 0, \ldots, 38$$

Multiplication in carry-safe representation

- Start with *reduced* elements $(a_0, \ldots, a_{19})$ and $(b_0, \ldots, b_{19})$, i.e.,

$$a_i, b_i \in [0, 2^{13} - 1], \quad i = 0, \ldots, 19$$

- Use 100 `mul` and `muladd` instructions to produce result $(r_0, \ldots, r_{38})$,

$$r_i \in [0, 2^{32} - 1], \quad i = 0, \ldots, 38$$

- $+$ lots of shuffles (free)
- $+$ overhead from non-integer radix
- $+$ overhead to combine intermediate results

Multiplication in carry-safe representation

- Start with *reduced* elements $(a_0, \ldots, a_{19})$ and $(b_0, \ldots, b_{19})$, i.e.,

$$a_i, b_i \in [0, 2^{13} - 1], \quad i = 0, \ldots, 19$$

- Use 100 `mul` and `muladd` instructions to produce result $(r_0, \ldots, r_{38})$,

$$r_i \in [0, 2^{32} - 1], \quad i = 0, \ldots, 38$$

- $+$ lots of shuffles (free)
- $+$ overhead from non-integer radix
- $+$ overhead to combine intermediate results
- Total: 145 arithmetic instructions, 145 cycles

Handling the carries

- Need to get reduced r from $(r_0, \dots, r_{38})$
- Standard carry chain:
  - Carry from r_{20} to r_{21}, from r_{21} to r_{22} etc., finally from r_{38} to r_{39}
  - Add $19 \cdot r_{20}$ to r_0, $19 \cdot r_{21}$ to r_1 etc.
  - Carry from r_0 to r_1, from r_1 to r_2 etc.

Handling the carries

- Need to get reduced r from $(r_0, \ldots, r_{38})$
- Standard carry chain:
  - Carry from r_{20} to r_{21}, from r_{21} to r_{22} etc., finally from r_{38} to r_{39}
  - Add $19 \cdot r_{20}$ to r_0, $19 \cdot r_{21}$ to r_1 etc.
  - Carry from r_0 to r_1, from r_1 to r_2 etc.
- Two problems with this:
  - No data-level parallelism (cannot really make use of vector instructions)

Handling the carries

- Need to get reduced r from $(r_0, \ldots, r_{38})$
- Standard carry chain:
  - Carry from r_{20} to r_{21}, from r_{21} to r_{22} etc., finally from r_{38} to r_{39}
  - Add $19 \cdot r_{20}$ to r_0, $19 \cdot r_{21}$ to r_1 etc.
  - Carry from r_0 to r_1, from r_1 to r_2 etc.
- Two problems with this:
  - No data-level parallelism (cannot really make use of vector instructions)
  - (Almost) no instruction-level parallelism (arithmetic happens only about every 4th cycle)

Interleaved reduction

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

Interleaved reduction

- Four independent 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 $r_{21} \to r_{22}$, $r_{25} \to r_{26}$, $r_{29} \to r_{30}$, $r_{33} \to r_{34}$
- $\dots$

Interleaved reduction

- Four independent 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 $r_{24} \rightarrow r_{25}$, $r_{28} \rightarrow r_{29}$, $r_{32} \rightarrow r_{33}$, $r_{36} \rightarrow r_{37}$
- ...

Interleaved reduction

- Four independent 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 $4 \cdot 20/32 = 2.5$

Exploit higher-level parallelism

- Many field operations in one Montgomery ladder step are independent
    - Group $2\times$ 4 multiplications together (squarings as multiplications)
    - Group additions/subtractions in blocks of 4
    - Always process 4 operations at a time
    - Leaves just one single multiplication at the end

Exploit higher-level parallelism

- Many field operations in one Montgomery ladder step are independent
  - Group $2\times$ 4 multiplications together (squarings as multiplications)
  - Group additions/subtractions in blocks of 4
  - Always process 4 operations at a time
  - Leaves just one single multiplication at the end
- Reduces number of arithmetic instructions for 4 multiplications from 580 to 420

Exploit higher-level parallelism

- Many field operations in one Montgomery ladder step are independent
  - Group $2\times$ 4 multiplications together (squarings as multiplications)
  - Group additions/subtractions in blocks of 4
  - Always process 4 operations at a time
  - Leaves just one single multiplication at the end
- Reduces number of arithmetic instructions for 4 multiplications from 580 to 420
- Uses SIMD for reduction: speed up by a factor of 4

Results

$\mathbf{696240}$ **cycles** for Curve25519 on one SPU of the CBE

http://cryptojedi.org/crypto/#celldh

Lattice-based signatures in AVX

- Joint work with Güneysu, Oder, and Pöppelmann (PQCrypto 2013)
- AVX:
  - Vector-instruction set for recent Intel and AMD processors
  - 16 256-bit registers
  - Only single-precision and double-precision float arithmetic
  - One 4-way-parallel double-precision multiplication and addition every cycle (on Sandy Bridge and Ivy Bridge)
- Lattice-based signatures
  - Consider scheme introduced by Lyubashevsky at Eurocrypt 2012
  - Aim at 100-bit security
  - Arithmetic in $R = \mathbb{F}_p[X]/(X^{512} + 1)$, with $p = 8383489$
  - p has 23 bits and $p \equiv 1 \pmod{1024}$

Lattice-based signatures in AVX

- Joint work with Güneysu, Oder, and Pöppelmann (PQCrypto 2013)
- AVX:
  - Vector-instruction set for recent Intel and AMD processors
  - 16 256-bit registers
  - Only single-precision and double-precision float arithmetic
  - One 4-way-parallel double-precision multiplication and addition every cycle (on Sandy Bridge and Ivy Bridge)
- Lattice-based signatures
  - Consider scheme introduced by Lyubashevsky at Eurocrypt 2012
  - Aim at ~~100-bit~~ 80-bit security
  - Arithmetic in $R = \mathbb{F}_p[X]/(X^{512}+1)$, with $p = 8383489$
  - p has 23 bits and $p \equiv 1 \pmod{1024}$

Lattice-based signatures in AVX

- Joint work with Güneysu, Oder, and Pöppelmann (PQCrypto 2013)
- AVX:
  - Vector-instruction set for recent Intel and AMD processors
  - 16 256-bit registers
  - Only single-precision and double-precision float arithmetic
  - One 4-way-parallel double-precision multiplication and addition every cycle (on Sandy Bridge and Ivy Bridge)
- Lattice-based signatures
  - Consider scheme introduced by Lyubashevsky at Eurocrypt 2012
  - Aim at ~~100-bit~~ 80-bit (?) security
  - Arithmetic in $R = \mathbb{F}_p[X]/(X^{512} + 1)$, with $p = 8383489$
  - p has 23 bits and $p \equiv 1 \pmod{1024}$

Representation of elements of R

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \dots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

Representation of elements of R

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \ldots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients

Representation of elements of R

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \ldots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
  - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}

Representation of elements of R

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \ldots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
  - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}
  - Compute $c \leftarrow a \cdot \overline{p^{-1}}$
  - Round c (high-throughput `vroundpd` instruction)
  - Compute $c \leftarrow c \cdot p$
  - Subtract c from a

Representation of elements of R

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \dots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
  - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}
  - Compute $c \leftarrow a \cdot \overline{p^{-1}}$
  - Round c (high-throughput `vroundpd` instruction)
  - Compute $c \leftarrow c \cdot p$
  - Subtract c from a
  - Rounding mode determines whether this maps to $[-\frac{p-1}{2}, \frac{p-1}{2}]$ or to $[0, p-1]$

Representation of elements of R

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \ldots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
  - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}
  - Compute $c \leftarrow a \cdot \overline{p^{-1}}$
  - Round c (high-throughput `vroundpd` instruction)
  - Compute $c \leftarrow c \cdot p$
  - Subtract c from a
  - Rounding mode determines whether this maps to $[-\frac{p-1}{2}, \frac{p-1}{2}]$ or to $[0, p-1]$
- Use lazy reduction: product of two 22-bit numbers has 44 bits, quite some space in the 53-bit mantissa

Multiplication in R

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \ldots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \ldots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

Multiplication in R

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \dots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \dots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

- Consider multiplication $d = a \cdot b$ in R
- Compute

$$\bar{a} = (a_0, \psi a_1, \dots, \psi^{511} a_{511}) \text{ and}$$
$$\bar{b} = (b_0, \psi b_1, \dots, \psi^{511} b_{511})$$

Multiplication in R

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \dots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \dots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

- Consider multiplication $d = a \cdot b$ in R
- Compute

$$\begin{aligned}\bar{a} &= (a_0, \psi a_1, \dots, \psi^{511} a_{511}) \text{ and} \\ \bar{b} &= (b_0, \psi b_1, \dots, \psi^{511} b_{511})\end{aligned}$$

- Obtain $\bar{d} = (d_0, \psi d_1, \dots, \psi^{511} d_{511})$ as

$$\bar{d} = \mathsf{NTT}_\omega^{-1}(\mathsf{NTT}_\omega(\bar{a}) \circ \mathsf{NTT}_\omega(\bar{b})),$$

where $\circ$ denotes component-wise multiplication

Multiplication in R

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \ldots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \ldots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

- Consider multiplication $d = a \cdot b$ in R
- Compute

$$\bar{a} = (a_0, \psi a_1, \ldots, \psi^{511} a_{511}) \text{ and}$$
$$\bar{b} = (b_0, \psi b_1, \ldots, \psi^{511} b_{511})$$

- Obtain $\bar{d} = (d_0, \psi d_1, \ldots, \psi^{511} d_{511})$ as

$$\bar{d} = \mathsf{NTT}_\omega^{-1}(\mathsf{NTT}_\omega(\bar{a}) \circ \mathsf{NTT}_\omega(\bar{b})),$$

  where $\circ$ denotes component-wise multiplication
- Component-wise multiplication is trivially vectorizable

NTT in AVX (Part I)

- Perform 9 levels with 256 “butterfly transformations” each
- Butterfly on level k:
  - Pick up a_i and a_{i+2^k}
  - Multiply a_{i+2^k} by a power of ω to obtain t
  - Compute $a_{i+2^k} \leftarrow a_i - t$
  - Compute $a_i \leftarrow a_i + t$
- Easy vectorization on levels $k = 2, \ldots, 8$:
  - Pick up $v_0 = a_i, a_{i+1}, a_{i+2}, a_{i+3}$ and $v_1 = a_{i+2^k}, a_{i+2^k+1}, a_{i+2^k+2}, a_{i+2^k+3}$
  - Perform all operations on v_0 and v_1
- Levels 0 and 1: More tricky: Use permutation instructions and “horizontal additions”

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access
- On one level of butterfly, pairs of values interact
- Through two levels, 4-tuples interact
- Through three levels, 8-tuples interact, etc.

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access
- On one level of butterfly, pairs of values interact
- Through two levels, 4-tuples interact
- Through three levels, 8-tuples interact, etc.
- Merge 3 levels: Load $8 \cdot 4 = 32$ values, perform arithmetic, store the results

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access
- On one level of butterfly, pairs of values interact
- Through two levels, 4-tuples interact
- Through three levels, 8-tuples interact, etc.
- Merge 3 levels: Load $8 \cdot 4 = 32$ values, perform arithmetic, store the results
- Final performance for NTT: 4484 cycles on Ivy Bridge

Result

634988 cycles on average to sign a 59-byte message on Ivy Bridge
45036 cycles to verify a signature on Ivy Bridge

http://cryptojedi.org/crypto/#lattisigns

Going binary

- So far: considered vectors of integers and floats
- How about arithmetic in binary fields?

Going binary

- So far: considered vectors of integers and floats
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.

Going binary

- So far: considered vectors of integers and floats
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.
- This is called *bitslicing*, introduced by Biham in 1997 for DES

Going binary

- So far: considered vectors of integers and floats
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.
- This is called *bitslicing*, introduced by Biham in 1997 for DES
- Other views on bitslicing:
  - Simulation of hardware implementations in software

Going binary

- So far: considered vectors of integers and floats
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.
- This is called *bitslicing*, introduced by Biham in 1997 for DES
- Other views on bitslicing:
  - Simulation of hardware implementations in software
  - Computations on a transposition of data

Bitslicing issues

- XOR, AND, OR, etc are usually fast (e.g., 3 128-bit operations per cycle on Intel Core 2)
- Can be very fast for operations that are not natively supported (like arithmetic in binary fields)

Bitslicing issues

- XOR, AND, OR, etc are usually fast (e.g., 3 128-bit operations per cycle on Intel Core 2)
- Can be very fast for operations that are not natively supported (like arithmetic in binary fields)
- Active data set increases massively (e.g., $128\times$)
- For "normal" vector operations, register space is increased accordingly (e.g, 16 256-bit vector registers vs. 16 64-bit integer registers)
- For bitslicing: Need to fit more data into the same registers
- Typical consequence: more loads and stores (that easily become the performance bottleneck)

CFS signatures in AVX

- Joint work with Dan Bernstein and Tony Chou (CHES 2013)
- AVX: One bit-logical operation on 256-bit vectors every cycle
- CFS: Code-based signature system by Courtois, Finiasz and Sendrier from 2001
- We use 80-bit security parameters from Indocrypt 2012 paper by Landais and Sendrier

CFS signatures in AVX

- Joint work with Dan Bernstein and Tony Chou (CHES 2013)
- AVX: One bit-logical operation on 256-bit vectors every cycle
- CFS: Code-based signature system by Courtois, Finiasz and Sendrier from 2001
- We use 80-bit security parameters from Indocrypt 2012 paper by Landais and Sendrier
- Basic idea:
  - Uses hidden binary Goppa code over $\mathbb{F}_{2^{20}}$ that can correct $t = 8$ errors
  - Signer hashes message M to a syndrome
  - If this syndrome corresponds to a word of distance at most t to a codeword, use secret decoding algorithm to obtain error positions and use those as a signature

CFS signatures in AVX

- Joint work with Dan Bernstein and Tony Chou (CHES 2013)
- AVX: One bit-logical operation on 256-bit vectors every cycle
- CFS: Code-based signature system by Courtois, Finiasz and Sendrier from 2001
- We use 80-bit security parameters from Indocrypt 2012 paper by Landais and Sendrier
- Basic idea:
  - Uses hidden binary Goppa code over $\mathbb{F}_{2^{20}}$ that can correct $t = 8$ errors
  - Signer hashes message M to a syndrome
  - If this syndrome corresponds to a word of distance at most t to a codeword, use secret decoding algorithm to obtain error positions and use those as a signature
  - Problem: This is likely to fail; so guess $\delta = 2$ additional error positions

CFS signatures in AVX

- Joint work with Dan Bernstein and Tony Chou (CHES 2013)
- AVX: One bit-logical operation on 256-bit vectors every cycle
- CFS: Code-based signature system by Courtois, Finiasz and Sendrier from 2001
- We use 80-bit security parameters from Indocrypt 2012 paper by Landais and Sendrier
- Basic idea:
  - Uses hidden binary Goppa code over $\mathbb{F}_{2^{20}}$ that can correct $t = 8$ errors
  - Signer hashes message M to a syndrome
  - If this syndrome corresponds to a word of distance at most t to a codeword, use secret decoding algorithm to obtain error positions and use those as a signature
  - Problem: This is likely to fail; so guess $\delta = 2$ additional error positions
  - Expected number of guesses: $\approx t! = 40320$ (embarrassingly parallel!)

Representing elements of $\mathbb{F}_{2^{20}}$

- Use polynomial representation with reduction trinomial $X^{20} + X^3 + 1$
- Bitsliced representation:

```
#include <immintrin.h>
typedef __m256d bit;

typedef struct{
  bit v[20];
} bgf20e __attribute__ ((aligned (32)));
```

Representing elements of $\mathbb{F}_{2^{20}}$

- Use polynomial representation with reduction trinomial $X^{20} + X^3 + 1$
- Bitsliced representation:

```
#include <immintrin.h>
typedef __m256d bit;

typedef struct{
  bit v[20];
} bgf20e __attribute__ ((aligned (32)));
```

- Addition corresponds to 40 loads, 20 XORs, 20 stores: 56 cycles

Representing elements of $\mathbb{F}_{2^{20}}$

- Use polynomial representation with reduction trinomial $X^{20} + X^3 + 1$
- Bitsliced representation:

```
#include <immintrin.h>
typedef __m256d bit;

typedef struct{
  bit v[20];
} bgf20e __attribute__ ((aligned (32)));
```

- Addition corresponds to 40 loads, 20 XORs, 20 stores: 56 cycles
- Squaring is just modular reduction: 64 cycles

Multiplication in $\mathbb{F}_{2^{20}}$

- First do binary-polynomial multiplication, then reduction
- We are currently investigating benefits of towering

Multiplication in $\mathbb{F}_{2^{20}}$

- First do binary-polynomial multiplication, then reduction
- We are currently investigating benefits of towering
- Schoolbook: 400 ANDs $+361$ XORs $+$ reduction

Multiplication in $\mathbb{F}_{2^{20}}$

- First do binary-polynomial multiplication, then reduction
- We are currently investigating benefits of towering
- Schoolbook: 400 ANDs $+361$ XORs $+$ reduction
- Much better: Karatsuba
  - Karatsuba:

$$\begin{aligned}
& (a_0 + X^n a_1)(b_0 + X^n b_1) \\
= \; & a_0 b_0 + X^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + X^{2n} a_1 b_1
\end{aligned}$$

Multiplication in $\mathbb{F}_{2^{20}}$

- First do binary-polynomial multiplication, then reduction
- We are currently investigating benefits of towering
- Schoolbook: 400 ANDs $+361$ XORs $+$ reduction
- Much better: refined Karatsuba
  - Karatsuba:

$$\begin{aligned} & (a_0 + X^n a_1)(b_0 + X^n b_1) \\ = \quad & a_0 b_0 + X^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + X^{2n} a_1 b_1 \end{aligned}$$

  - Refined Karatsuba:

$$\begin{aligned} & (a_0 + X^n a_1)(b_0 + X^n b_1) \\ = \quad & (1 - X^n)(a_0 b_0 - X^n a_1 b_1) + X^n (a_0 + a_1)(b_0 + b_1) \end{aligned}$$

- Refined Karatsuba uses $M_{2n} = 3M_n + 7n - 3$ instead of $M_{2n} = 3M_n + 8n - 4$ bit operations

Multiplication in $\mathbb{F}_{2^{20}}$

- First do binary-polynomial multiplication, then reduction
- We are currently investigating benefits of towering
- Schoolbook: 400 ANDs $+361$ XORs $+$ reduction
- Much better: refined Karatsuba
  - Karatsuba:

$$\begin{aligned} & (a_0 + X^n a_1)(b_0 + X^n b_1) \\ = \; & a_0 b_0 + X^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + X^{2n} a_1 b_1 \end{aligned}$$

  - Refined Karatsuba:

$$\begin{aligned} & (a_0 + X^n a_1)(b_0 + X^n b_1) \\ = \; & (1 - X^n)(a_0 b_0 - X^n a_1 b_1) + X^n (a_0 + a_1)(b_0 + b_1) \end{aligned}$$

- Refined Karatsuba uses $M_{2n} = 3M_n + 7n - 3$ instead of $M_{2n} = 3M_n + 8n - 4$ bit operations
- With two levels of refined Karatsuba: 225 ANDs $+303$ XORs $+$ reduction
- Performance: 744 cycles per 256 multiplications

Result

$< \mathbf{425,000,000}$ **cycles** on average for signing on Ivy Bridge

`http://cryptojedi.org/crypto/#mcbits` (not yet online)

Result

$< \mathbf{425,000,000}$ **cycles** on average for signing on Ivy Bridge
($10\times$ faster than previous results)

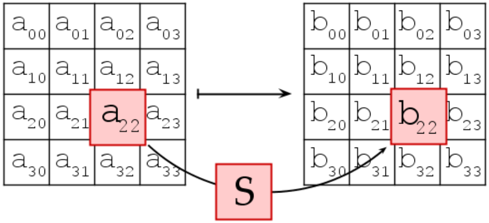
`http://cryptojedi.org/crypto/#mcbits` (not yet online)

Back to symmetric crypto: AES in SSE

- Joint work with Emilia Käsper (CHES 2009)
- AES:
  - Block cipher introduced as Rijndael by Daemen and Rijmen in 1999
  - Transforms a 16-byte state (block) through 10 rounds (for 128-bit key)
  - Each round consists of 4 operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey (last round doesn't have MixColumns)

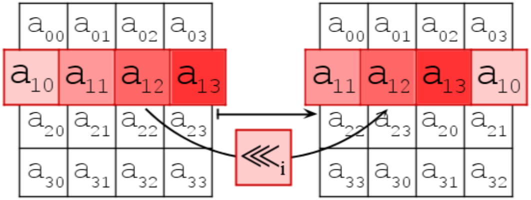
The AES operations, part I

- SubBytes is an S-Box acting on individual bytes
- Substitution based on inversion in $\mathbb{F}_{2^8}$



a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
S
b_{00} b_{01} b_{02} b_{03}
b_{10} b_{11} b_{12} b_{13}
b_{20} b_{21} b_{22} b_{23}
b_{30} b_{31} b_{32} b_{33}

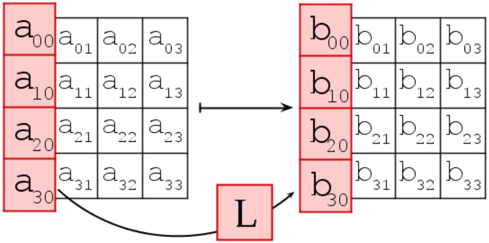
- ShiftRows rotates each row by a different amount



a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
$\lll_i$
a_{00} a_{01} a_{02} a_{03}
a_{11} a_{12} a_{13} a_{10}
a_{22} a_{23} a_{20} a_{21}
a_{33} a_{30} a_{31} a_{32}

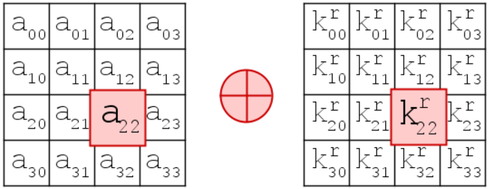
The AES operations, part II

- MixColumns is a linear transformation on columns



a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
L
b_{00} b_{01} b_{02} b_{03}
b_{10} b_{11} b_{12} b_{13}
b_{20} b_{21} b_{22} b_{23}
b_{30} b_{31} b_{32} b_{33}

- AddRoundKey XORs the 128-bit round key to the state



a_{00} a_{01} a_{02} a_{03}
a_{10} a_{11} a_{12} a_{13}
a_{20} a_{21} a_{22} a_{23}
a_{30} a_{31} a_{32} a_{33}
$\oplus$
k^r_{00} k^r_{01} k^r_{02} k^r_{03}
k^r_{10} k^r_{11} k^r_{12} k^r_{13}
k^r_{20} k^r_{21} k^r_{22} k^r_{23}
k^r_{30} k^r_{31} k^r_{32} k^r_{33}

Bitsliching AES

... the direct way

- Consider AES in counter mode; encryption of consecutive blocks is independent
- With 128-bit vector registers: process 128 blocks (2 KB) in parallel

Bitslicing AES
... the direct way

- Consider AES in counter mode; encryption of consecutive blocks is independent
- With 128-bit vector registers: process 128 blocks (2 KB) in parallel
- Approach taken by, e.g., Matsui and Nakajima (CHES 2007)
- Good performance of 9.2 cycles/byte for long messages (and bitsliced input)
- Bad performance for short IP packets

Bitsliced AES for small packets

- Idea: Main part of AES is SubBytes, already 16-way parallel
- Consider only 8 consecutive blocks

Bitsliced AES for small packets

- Idea: Main part of AES is SubBytes, already 16-way parallel
- Consider only 8 consecutive blocks
- Pack bits into 128-bit vector registers:

row 0										row 1	row 2	row 3
column 0				. . . . . .	column 3				. . .	. . .	. . .
block 0	block 1	. . .	block 7	. . . . . .	block 0	block 1	. . .	block 7	. . .	. . .	. . .

- Bits inside one byte belong to different blocks, so all instructions can work on bytes

Bitsliced AES for small packets

- Idea: Main part of AES is SubBytes, already 16-way parallel
- Consider only 8 consecutive blocks
- Pack bits into 128-bit vector registers:

row 0									row 1	row 2	row 3
column 0				......	column 3				...	...	...
block 0	block 1	...	block 7	......	block 0	block 1	...	block 7	...	...	...

- Bits inside one byte belong to different blocks, so all instructions can work on bytes
- For ShiftRows and MixColumns use fast `pshufb` byte-shuffle instruction (SSSE3, Intel only)

Bitsliced AES S-Box

- Start with a good hardware implementation of SubBytes (inversion in $\mathbb{F}_{2^8}$): Canright, 2005; Boyar, Peralta, 2009: 117 gates
- That should turn into 117 bit-logical instructions, right?

Bitsliced AES S-Box

- Start with a good hardware implementation of SubBytes (inversion in $\mathbb{F}_{2^8}$): Canright, 2005; Boyar, Peralta, 2009: 117 gates
- That should turn into 117 bit-logical instructions, right? No:
  - We only have 16 registers

Bitsliced AES S-Box

- Start with a good hardware implementation of SubBytes (inversion in $\mathbb{F}_{2^8}$): Canright, 2005; Boyar, Peralta, 2009: 117 gates
- That should turn into 117 bit-logical instructions, right? No:
  - We only have 16 registers
  - Only 2-operand instructions, e.g., $a \leftarrow a \oplus b$

Bitsliced AES S-Box

- Start with a good hardware implementation of SubBytes (inversion in $\mathbb{F}_{2^8}$): Canright, 2005; Boyar, Peralta, 2009: 117 gates
- That should turn into 117 bit-logical instructions, right? No:
  - We only have 16 registers
  - Only 2-operand instructions, e.g., $a \leftarrow a \oplus b$
- "Gate" counts, hardware vs. software:

	XOR	**AND/OR**	**MOV**	**total**
Hardware	82	35	–	117
Software	93	35	35	163

Results

9.32 cycles/byte for AES-CTR on Intel Core 2 Q6600
7.58 cycles/byte for AES-CTR on Intel Core 2 Q9550

http://cryptojedi.org/crypto/#aesbs

References

Daniel J. Bernstein and Peter Schwabe: **NEON crypto.** http://cryptojedi.org/#neoncrypto

Neil Costigan and Peter Schwabe **Fast elliptic-curve cryptography on the Cell Broadband Engine.** http://cryptojedi.org/#celldh

Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe: **Software speed records for lattice-based signatures.** http://cryptojedi.org/#lattisigns

Daniel J. Bernstein, Tung Chou, and Peter Schwabe: **McBits: fast constant-time code-based cryptography.** http://cryptojedi.org/#mcbits

Emilia Käsper and Peter Schwabe: **Faster and Timing-Attack Resistant AES-GCM.** http://cryptojedi.org/#aesbs