Vectorized implementations of post-quantum crypto

Peter Schwabe

January 12, 2015

DIMACS Workshop on the Mathematics of Post-Quantum Cryptography

"The multicore revolution"

- Until early years 2000 each new processor generation had higher clock speeds
- Nowadays: increase performance by number of cores:
  - My laptop has 2 phyiscal (and 4 virtual) cores
  - Smartphones typically have 2 or 4 cores
  - Servers have 4, 8, 16,... cores
  - Special-purpose hardware (e.g., GPUs) often comes with many more cores
- Consequence: "The free lunch is over" (Herb Sutter, 2005)

"The multicore revolution"

- Until early years 2000 each new processor generation had higher clock speeds
- Nowadays: increase performance by number of cores:
  - My laptop has 2 phyiscal (and 4 virtual) cores
  - Smartphones typically have 2 or 4 cores
  - Servers have 4, 8, 16,... cores
  - Special-purpose hardware (e.g., GPUs) often comes with many more cores
- Consequence: "The free lunch is over" (Herb Sutter, 2005)

"As a result, system designers and software engineers can no longer rely on increasing clock speed to hide software bloat. Instead, they must somehow learn to make effective use of increasing parallelism."

—Maurice Herlihy: The Multicore Revolution, 2007

Why multicore doesn't matter...
... for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Why multicore doesn't matter. . .
. . . for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Post-quantum crypto is fast

- > 3900 "lattisigns512" signatures per second
- > 45000 "lattisigns512" verifications per second
- > 38000 rainbow5640 signatures per second
- > 57000 rainbow5640 verifications per second

Why multicore doesn't matter...

... for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Post-quantum crypto is fast

- > 3900 "lattisigns512" signatures per second
- > 45000 "lattisigns512" verifications per second
- > 38000 rainbow5640 signatures per second
- > 57000 rainbow5640 verifications per second

- **If you perform only one crypto operation, you don't care**

Why multicore doesn't matter. . .

. . . for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Post-quantum crypto is fast

- > 3900 "lattisigns512" signatures per second
- > 45000 "lattisigns512" verifications per second
- > 38000 rainbow5640 signatures per second
- > 57000 rainbow5640 verifications per second

- **If you perform only one crypto operation, you don't care**
- **Many crypto operations are trivially parallel on multiple cores**

Pipelined and multiscalar processors

- Almost all CPUs chop instructions into smaller tasks, e.g., for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register

Pipelined and multiscalar processors

- Almost all CPUs chop instructions into smaller tasks, e.g., for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register
- Pipelined execution: overlap processing of *independent* instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)

Pipelined and multiscalar processors

- Almost all CPUs chop instructions into smaller tasks, e.g., for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register
- Pipelined execution: overlap processing of *independent* instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- Superscalar execution: duplicate units and process multiple instructions in the same stage

Pipelined and multiscalar processors

- Almost all CPUs chop instructions into smaller tasks, e.g., for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register
- Pipelined execution: overlap processing of *independent* instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- Superscalar execution: duplicate units and process multiple instructions in the same stage
- Crucial to make use of these concepts: *instruction-level parallelism*
- To some extent, compilers will help here

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...
- Need to interleave data items (e.g., 32-bit integers) in memory
- Compilers will not help with vectorization

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...
- Need to interleave data items (e.g., 32-bit integers) in memory
- Compilers will not really help with vectorization

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
  - $4\times$ 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
  - $4\times$ 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**
- 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!

Example 1: Lattice-based crypto

- Latincrypt 2014: Fast LWE signatures, joint work with Dagdelen, Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, and Sánchez.
- Most expensive operation: matrix-vector multiplication mod $2^{29} - 3$

Example 1: Lattice-based crypto

- Latincrypt 2014: Fast LWE signatures, joint work with Dagdelen, Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, and Sánchez.
- Most expensive operation: matrix-vector multiplication mod $2^{29} - 3$
- Use Intel AVX2 instructions: two $4\times$ vectorized double-precision multiply-accumulate every cycle
- Represent elements of $\mathbb{F}_{2^{29}-3}$ as doubles

Example 1: Lattice-based crypto

- Latincrypt 2014: Fast LWE signatures, joint work with Dagdelen, Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, and Sánchez.
- Most expensive operation: matrix-vector multiplication mod $2^{29} - 3$
- Use Intel AVX2 instructions: two $4\times$ vectorized double-precision multiply-accumulate every cycle
- Represent elements of $\mathbb{F}_{2^{29}-3}$ as doubles
- Matrix dimensions: 532×840
- Expected performance: $532 \cdot 840/8 = 55860$ cycles

Example 1: Lattice-based crypto

- Latincrypt 2014: Fast LWE signatures, joint work with Dagdelen, Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, and Sánchez.
- Most expensive operation: matrix-vector multiplication mod $2^{29} - 3$
- Use Intel AVX2 instructions: two $4\times$ vectorized double-precision multiply-accumulate every cycle
- Represent elements of $\mathbb{F}_{2^{29}-3}$ as doubles
- Matrix dimensions: 532×840
- Expected performance: $532 \cdot 840/8 = 55860$ cycles
- Actual performance: 278912 cycles

Example 1: Lattice-based crypto

- Latincrypt 2014: Fast LWE signatures, joint work with Dagdelen, Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, and Sánchez.
- Most expensive operation: matrix-vector multiplication mod $2^{29} - 3$
- Use Intel AVX2 instructions: two $4\times$ vectorized double-precision multiply-accumulate every cycle
- Represent elements of $\mathbb{F}_{2^{29}-3}$ as doubles
- Matrix dimensions: 532×840
- Expected performance: $532 \cdot 840/8 = 55860$ cycles
- Actual performance: 278912 cycles; reason: L2-cache throughput

Example 1: Lattice-based crypto

- Latincrypt 2014: Fast LWE signatures, joint work with Dagdelen, Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, and Sánchez.
- Most expensive operation: matrix-vector multiplication mod $2^{29} - 3$
- Use Intel AVX2 instructions: two $4\times$ vectorized double-precision multiply-accumulate every cycle
- Represent elements of $\mathbb{F}_{2^{29}-3}$ as doubles
- Matrix dimensions: 532×840
- Expected performance: $532 \cdot 840/8 = 55860$ cycles
- Actual performance: 278912 cycles; reason: L2-cache throughput
- This performance is already after compressing matrix entries to 32-bit integers

Example 1: Lattice-based crypto

- Latincrypt 2014: Fast LWE signatures, joint work with Dagdelen, Bansarkhani, Göpfert, Güneysu, Oder, Pöppelmann, and Sánchez.
- Most expensive operation: matrix-vector multiplication mod $2^{29} - 3$
- Use Intel AVX2 instructions: two $4\times$ vectorized double-precision multiply-accumulate every cycle
- Represent elements of $\mathbb{F}_{2^{29}-3}$ as doubles
- Matrix dimensions: 532×840
- Expected performance: $532 \cdot 840/8 = 55860$ cycles
- Actual performance: 278912 cycles; reason: L2-cache throughput
- This performance is already after compressing matrix entries to 32-bit integers
- **Lesson: standard-lattice crypto vectorizes trivially, but bottlenecked by loads of large matrix**

Example 2: Ideal lattices

- PQCrypto 2013: Software for GLP signatures, joint work with Güneysu, Oder, and Pöppelmann
- Most costly operation: multiply in $\mathcal{R} = \mathbb{F}_p[x]/\langle x^n + 1\rangle$, where
  - n is a power of 2
  - p is a prime congruent to 1 modulo $2n$
- Specifically, we used
- $n = 512$ and
- $p = 8383489$

Multiplication in $\mathcal{R}$

- 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

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

Multiplication in $\mathcal{R}$

- 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 $\mathcal{R}$, compute

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

Multiplication in $\mathcal{R}$

- 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 $\mathcal{R}$, 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

Multiplication in $\mathcal{R}$

- 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 $\mathcal{R}$, 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/AVX2

- Loop over 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 $a_i \leftarrow a_i + t$

NTT in AVX/AVX2

- Loop over 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

NTT in AVX/AVX2

- Loop over 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 $k = 2, \dots, 8$:
  - 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/AVX2

- Loop over 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 $k = 2, \dots, 8$:
  - 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"
- Lower cycle bound from arithmetic: 2176 cycles
- Actual performance: 4484 cycles (Ivy Bridge)

NTT in AVX/AVX2

- Loop over 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 $k = 2, \dots, 8$:
  - 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"
- Lower cycle bound from arithmetic: 2176 cycles
- Actual performance: 4484 cycles (Ivy Bridge)
- **Lesson: ideal lattices vectorize well for suitable parameters**

Example 3: CFS signatures

- Code-based signatures by Courtois, Finiasz, and Sendrier, 2001
- Basic idea:
  - Hash the message to a syndrome
  - If the syndrome has distance $\leq t$ from a code word, use secret decoding algorithm to determine error positions
  - Send error positions
  - Address low chance of having distance $\leq t$ by guessing positions
  - Average number of decoding attempts: $\approx t!$

Example 3: CFS signatures

- Code-based signatures by Courtois, Finiasz, and Sendrier, 2001
- Basic idea:
  - Hash the message to a syndrome
  - If the syndrome has distance $\leq t$ from a code word, use secret decoding algorithm to determine error positions
  - Send error positions
  - Address low chance of having distance $\leq t$ by guessing positions
  - Average number of decoding attempts: $\approx t!$
- Indocrypt 2012: Landais and Sendrier propose parameters and optimization techniques, and present a software implementation of CFS

Example 3: CFS signatures

- Code-based signatures by Courtois, Finiasz, and Sendrier, 2001
- Basic idea:
  - Hash the message to a syndrome
  - If the syndrome has distance $\leq t$ from a code word, use secret decoding algorithm to determine error positions
  - Send error positions
  - Address low chance of having distance $\leq t$ by guessing positions
  - Average number of decoding attempts: $\approx t!$
- Indocrypt 2012: Landais and Sendrier propose parameters and optimization techniques, and present a software implementation of CFS
- CHES 2013: Bernstein, Chou, Schwabe: $10\times$ speedup

Example 3: CFS signatures

- Code-based signatures by Courtois, Finiasz, and Sendrier, 2001
- Basic idea:
  - Hash the message to a syndrome
  - If the syndrome has distance $\leq t$ from a code word, use secret decoding algorithm to determine error positions
  - Send error positions
  - Address low chance of having distance $\leq t$ by guessing positions
  - Average number of decoding attempts: $\approx t!$
- Indocrypt 2012: Landais and Sendrier propose parameters and optimization techniques, and present a software implementation of CFS
- CHES 2013: Bernstein, Chou, Schwabe: $10\times$ speedup
- Main technique for the speedup: vectorization

Vectorizing binary arithmetic

- With $t = 8$ we need ≈ 40320 decoding attempts
- Arithmetic is on small-degree polynomials over $\mathbb{F}_{2^{20}}$

Vectorizing binary arithmetic

- With $t = 8$ we need ≈ 40320 decoding attempts
- Arithmetic is on small-degree polynomials over $\mathbb{F}_{2^{20}}$
- So far: considered vectors of integers and floats
- How about arithmetic in binary fields?

Vectorizing binary arithmetic

- With $t = 8$ we need ≈ 40320 decoding attempts
- Arithmetic is on small-degree polynomials over $\mathbb{F}_{2^{20}}$
- 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.

Vectorizing binary arithmetic

- With $t = 8$ we need ≈ 40320 decoding attempts
- Arithmetic is on small-degree polynomials over $\mathbb{F}_{2^{20}}$
- 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

Vectorizing binary arithmetic

- With $t = 8$ we need ≈ 40320 decoding attempts
- Arithmetic is on small-degree polynomials over $\mathbb{F}_{2^{20}}$
- 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

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

- First do binary-polynomial multiplication, then reduction
- Possibly better: tower-field constructions

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

- First do binary-polynomial multiplication, then reduction
- Possibly better: tower-field constructions
- Schoolbook: 400 ANDs $+361$ XORs $+$ reduction

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

- First do binary-polynomial multiplication, then reduction
- Possibly better: tower-field constructions
- 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
- Possibly better: tower-field constructions
- 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

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

- First do binary-polynomial multiplication, then reduction
- Possibly better: tower-field constructions
- 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
- 2 levels of refined Karatsuba: 225 ANDs $+303$ XORs $+$ reduction
- Performance: 744 cycles per 256 multiplications

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

- First do binary-polynomial multiplication, then reduction
- Possibly better: tower-field constructions
- 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
- 2 levels of refined Karatsuba: 225 ANDs $+303$ XORs $+$ reduction
- Performance: 744 cycles per 256 multiplications
- **Lesson: code-based crypto vectorizes (bitslices) well, but need to find parallelism**

Summary

- Parallelism $\neq$ parallelism

Summary

- Parallelism $\neq$ parallelism
- *Don't think* about a program as one sequence of instructions operating on one set of data
- Think about a program as one long instruction stream operating in parallel in multiple independent sets of data

Summary

- Parallelism $\neq$ parallelism
- *Don't think* about a program as one sequence of instructions operating on one set of data
- Think about a program as one long instruction stream operating in parallel in multiple independent sets of data
- Data flow from one data set to another ("vector permutation") incurs overhead

Summary

- Parallelism $\neq$ parallelism
- *Don't think* about a program as one sequence of instructions operating on one set of data
- Think about a program as one long instruction stream operating in parallel in multiple independent sets of data
- Data flow from one data set to another ("vector permutation") incurs overhead
- Synergy between vectorization and timing-attack protection:
  - Think branchfree
  - Don't think lookup tables

Papers

- Özgür Dagdelen, Rachid El Bansarkhani, Florian Göpfert, Tim Güneysu, Tobias Oder, Thomas Pöppelmann, Ana Helena Sánchez, and Peter Schwabe: **High-speed signatures from standard lattices.**
  http://cryptojedi.org/papers/#lwesign (online soon)
- Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe: **Software speed records for lattice-based signatures.**
  http://cryptojedi.org/papers/#lattisigns
- Daniel J. Bernstein, Tung Chou, and Peter Schwabe: **McBits: fast constant-time code-based cryptography.**
  http://cryptojedi.org/papers/#mcbits