Radbound University

Implementing post-quantum crypto

Peter Schwabe

peter@cryptojedi.org

https://cryptojedi.org

February 1, 2018

Making (crypto) software fast

On “small” processors

- Step 1: Efficiently map algorithm to (arithmetic) instructions

Making (crypto) software fast

On "small" processors

- Step 1: Efficiently map algorithm to (arithmetic) instructions
- Step 2: Reduce memory access

Making (crypto) software fast

On "small" processors

- Step 1: Efficiently map algorithm to (arithmetic) instructions
- Step 2: Reduce memory access

On "interesting" processors

- The above plus **exploit parallelism**

Making (crypto) software fast

On "small" processors

- Step 1: Efficiently map algorithm to (arithmetic) instructions
- Step 2: Reduce memory access

On "interesting" processors

- The above plus **exploit parallelism**
- Exploit parallelism $\neq$ multicore implementations

Making (crypto) software fast

On “small” processors

- Step 1: Efficiently map algorithm to (arithmetic) instructions
- Step 2: Reduce memory access

On “interesting” processors

- The above plus **exploit parallelism**
- Exploit parallelism $\neq$ multicore implementations
- Pipelining: interleave execution of independent instructions
- Requires *instruction-level parallelism*

On "small" processors

- Step 1: Efficiently map algorithm to (arithmetic) instructions
- Step 2: Reduce memory access

On "interesting" processors

- The above plus **exploit parallelism**
- Exploit parallelism $\neq$ multicore implementations
- Pipelining: interleave execution of independent instructions
- Requires *instruction-level parallelism*
- Superscalar execution: multiple units $\Rightarrow$ multiple ops per cycle
- Choose instructions that keep units busy

Making (crypto) software fast

On "small" processors

- Step 1: Efficiently map algorithm to (arithmetic) instructions
- Step 2: Reduce memory access

On "interesting" processors

- The above plus **exploit parallelism**
- Exploit parallelism $\neq$ multicore implementations
- Pipelining: interleave execution of independent instructions
- Requires *instruction-level parallelism*
- Superscalar execution: multiple units $\Rightarrow$ multiple ops per cycle
- Choose instructions that keep units busy
- **Vectorize!**

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...
- 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...
- Compilers will not really help with vectorization

Why is this so great?

- Consider the Intel Skylake processor

Why is this so great?

- Consider the Intel Skylake processor
    - 32-bit load throughput: 2 per cycle
    - 32-bit add throughput: 4 per cycle
    - 32-bit store throughput: 1 per cycle

Why is this so great?

- Consider the Intel Skylake processor
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 2 per cycle
  - $8\times$ 32-bit add throughput: 3 per cycle
  - 256-bit store throughput: 1 per cycle

Why is this so great?

- Consider the Intel Skylake processor
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 2 per cycle
  - $8\times$ 32-bit add throughput: 3 per cycle
  - 256-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do $8\times$ the work**

Why is this so great?

- Consider the Intel Skylake processor
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 2 per cycle
  - $8\times$ 32-bit add throughput: 3 per cycle
  - 256-bit store throughput: 1 per cycle
- **Vector instructions are almost as fast as scalar instructions but do $8\times$ the work**
- Situation on other architectures/microarchitectures is similar
- Reason: cheap way to increase arithmetic throughput (less decoding, address computation, etc.)

Take-home message

"Big multipliers are pre-quantum,
vectorization is post-quantum"

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for "small" q
- These are trivially vectorizable
- So trivial that even compilers may do it!

Standard-lattice-based schemes

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for "small" q
- These are trivially vectorizable
- So trivial that even compilers may do it!
- Standard-lattice-based signatures (e.g., Bai-Galbraith):
  - Multiple attempts for signing (rejection sampling)
  - Each attempt: compute $\mathbf{Av}$ for fixed $\mathbf{A}$

Standard-lattice-based schemes

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for "small" q
- These are trivially vectorizable
- So trivial that even compilers may do it!
- Standard-lattice-based signatures (e.g., Bai-Galbraith):
  - Multiple attempts for signing (rejection sampling)
  - Each attempt: compute $\mathbf{Av}$ for fixed $\mathbf{A}$
- More efficient:
  - Compute multiple products $\mathbf{Av_i}$
  - Typically ignore some results

Standard-lattice-based schemes

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for "small" q
- These are trivially vectorizable
- So trivial that even compilers may do it!
- Standard-lattice-based signatures (e.g., Bai-Galbraith):
  - Multiple attempts for signing (rejection sampling)
  - Each attempt: compute $\mathbf{Av}$ for fixed $\mathbf{A}$
- More efficient:
  - Compute multiple products $\mathbf{Av_i}$
  - Typically ignore some results
- Reason: reuse coefficients of $\mathbf{A}$ in cache

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

$$
\begin{aligned}
r_0 &= f_0 g_0 \\
r_1 &= f_0 g_1 + f_1 g_0 \\
r_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 \\
r_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 \\
r_4 &= f_1 g_3 + f_2 g_2 + f_3 g_1 \\
r_5 &= f_2 g_3 + f_3 g_2 \\
r_6 &= f_3 g_3
\end{aligned}
$$

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

$$
\begin{aligned}
r_0 &= f_0g_0 \\
r_1 &= f_0g_1 + f_1g_0 \\
r_2 &= f_0g_2 + f_1g_1 + f_2g_0 \\
r_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 \\
r_4 &= f_1g_3 + f_2g_2 + f_3g_1 \\
r_5 &= f_2g_3 + f_3g_2 \\
r_6 &= f_3g_3
\end{aligned}
$$

- Can easily load (f_0, f_1, f_2, f_3) and (g_0, g_1, g_2, g_3)
- Multiply, obtain $(f_0g_0, f_1g_1, f_2g_2, f_3g_3)$

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

$$
\begin{aligned}
r_0 &= f_0 g_0 \\
r_1 &= f_0 g_1 + f_1 g_0 \\
r_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 \\
r_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 \\
r_4 &= f_1 g_3 + f_2 g_2 + f_3 g_1 \\
r_5 &= f_2 g_3 + f_3 g_2 \\
r_6 &= f_3 g_3
\end{aligned}
$$

- Can easily load (f_0, f_1, f_2, f_3) and (g_0, g_1, g_2, g_3)
- Multiply, obtain $(f_0 g_0, f_1 g_1, f_2 g_2, f_3 g_3)$
- And now what?

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

$$
\begin{aligned}
r_0 &= f_0 g_0 \\
r_1 &= f_0 g_1 + f_1 g_0 \\
r_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 \\
r_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 \\
r_4 &= f_1 g_3 + f_2 g_2 + f_3 g_1 \\
r_5 &= f_2 g_3 + f_3 g_2 \\
r_6 &= f_3 g_3
\end{aligned}
$$

- Can easily load (f_0, f_1, f_2, f_3) and (g_0, g_1, g_2, g_3)
- Multiply, obtain $(f_0 g_0, f_1 g_1, f_2 g_2, f_3 g_3)$
- And now what?
- Looks like we need to *shuffle* a lot!

Karatsuba and Toom

- Our polynomials have many more coefficients (say, 256–1024)
- Idea: use Karatsuba's trick:
  - consider $n = 2k$-coefficient polynomials f and g
  - Split multiplication $f \cdot g$ into 3 half-size multiplications

$$
\begin{aligned}
& (a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= \; & a_\ell b_\ell + X^k (a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= \; & a_\ell b_\ell + X^k ((a_\ell + a_h)(b_\ell + b_h) - a_\ell b \ell - a_h b_h) + X^n a_h b_h
\end{aligned}
$$

Karatsuba and Toom

- Our polynomials have many more coefficients (say, 256–1024)
- Idea: use Karatsuba's trick:
  - consider $n = 2k$-coefficient polynomials f and g
  - Split multiplication $f \cdot g$ into 3 half-size multiplications

$$
\begin{aligned}
& (a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= \; & a_\ell b_\ell + X^k (a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= \; & a_\ell b_\ell + X^k ((a_\ell + a_h)(b_\ell + b_h) - a_\ell b\ell - a_h b_h) + X^n a_h b_h
\end{aligned}
$$

- Apply recursively to obtain 9 quarter-size multiplications, 27 eighth-size multiplications etc.

Karatsuba and Toom

- Our polynomials have many more coefficients (say, 256–1024)
- Idea: use Karatsuba's trick:
  - consider $n = 2k$-coefficient polynomials f and g
  - Split multiplication $f \cdot g$ into 3 half-size multiplications

$$
\begin{aligned}
& (a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= \; & a_\ell b_\ell + X^k (a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= \; & a_\ell b_\ell + X^k ((a_\ell + a_h)(b_\ell + b_h) - a_\ell b\ell - a_h b_h) + X^n a_h b_h
\end{aligned}
$$

- Apply recursively to obtain 9 quarter-size multiplications, 27 eighth-size multiplications etc.
- Generalization: Toom-Cook. Obtain, e.g., 5 third-size multiplications
- Split into sufficiently many "small" multiplications, vectorize across those

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1X + a_2X^2$

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1X + a_2X^2$
- Coefficients in memory:

```
a0, a1, a2, b0, b1, b2, c0,..., h1, h2
```

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1 X + a_2 X^2$
- Coefficients in memory:

```
a0, a1, a2, b0, b1, b2, c0,..., h1, h2
```

- Problem:
  - Vector loads will yield

$$v_0 = (a_0, a_1, a_2, b_0) \qquad \ldots \qquad v_6 = (g_2, h_0, h_1, h_2)$$

  - However, we need

$$v_0 = (a_0, c_0, e_0, h_0) \qquad \ldots \qquad v_6 = (b_2, d_2, f_2, g_2)$$

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1X + a_2X^2$
- Coefficients in memory:

```
a0, a1, a2, b0, b1, b2, c0,..., h1, h2
```

- Problem:
  - Vector loads will yield

$$v_0 = (a_0, a_1, a_2, b_0) \qquad \dots \qquad v_6 = (g_2, h_0, h_1, h_2)$$

  - However, we need

$$v_0 = (a_0, c_0, e_0, h_0) \qquad \dots \qquad v_6 = (b_2, d_2, f_2, g_2)$$

- Solution: transpose data matrix (or interleave words):

```
a0, c0, e0, h0, a1, c1, e1,..., f2, g2
```

Two applications of Karatsuba/Toom

Streamlined NTRU Prime 4591^{761}

- Multiply in the ring $\mathcal{R} = \mathbb{Z}_{4591}[X]/(X^{761} - X - 1)$
- Pad input polynomial to 768 coefficients
- 5 levels of Karatsuba: 243 multiplications of 24-coefficient polynomials
- Massively lazy reduction using double-precision floats
- 28 682 Haswell cycles for multiplication in $\mathcal{R}$

Streamlined NTRU Prime 4591^{761}

- Multiply in the ring $\mathcal{R} = \mathbb{Z}_{4591}[X]/(X^{761} - X - 1)$
- Pad input polynomial to 768 coefficients
- 5 levels of Karatsuba: 243 multiplications of 24-coefficient polynomials
- Massively lazy reduction using double-precision floats
- 28 682 Haswell cycles for multiplication in $\mathcal{R}$

NTRU-HRSS-KEM

- Multiply in the ring $\mathcal{R} = \mathbb{Z}_{8192}[X]/(X^{701} - 1)$
- Use Toom-Cook to split into 7 quarter-size, then 2 levels of Karatsuba
- Obtain 63 multiplications of 44-coefficient polynomials
- 11 722 Haswell cycles for multiplication in $\mathcal{R}$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q - 1)$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q - 1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q-1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)
- Big advantage: fast *negacyclic number-theoretic transform*
- Given $g \in \mathcal{R}$, n-th primitive root of unity ω and $\psi = \sqrt{\omega}$, compute

$$\mathrm{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q-1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)
- Big advantage: fast *negacyclic number-theoretic transform*
- Given $g \in \mathcal{R}$, n-th primitive root of unity ω and $\psi = \sqrt{\omega}$, compute

$$\mathrm{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

- Compute $f \cdot g$ as $\mathrm{NTT}^{-1}(\mathrm{NTT}(f) \circ \mathrm{NTT}(g))$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q - 1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)
- Big advantage: fast *negacyclic number-theoretic transform*
- Given $g \in \mathcal{R}$, n-th primitive root of unity ω and $\psi = \sqrt{\omega}$, compute

$$\mathrm{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

- Compute $f \cdot g$ as $\mathrm{NTT}^{-1}(\mathrm{NTT}(f) \circ \mathrm{NTT}(g))$
- NTT^{-1} is essentially the same computation as NTT

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
    - Write polynomial f as $f_0(X^2) + X f_1(X^2)$

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

  - f_0 has $n/2$ coefficients
  - Evaluate f_0 at all $(n/2)$-th roots of unity by recursive application
  - Same for f_1

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

  - f_0 has $n/2$ coefficients
  - Evaluate f_0 at all $(n/2)$-th roots of unity by recursive application
  - Same for f_1
- Apply recursively through $\log n$ levels

Vectorizing the NTT

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ "butterflies" each
- Butterfly on level k:
  - Pick up f_i and f_{i+2^k}
  - Multiply f_{i+2^k} by a power of ω to obtain t
  - Compute $f_{i+2^k} \leftarrow a_i - t$
  - Compute $f_i \leftarrow a_i + t$
- All $n/2$ butterflies on one level are independent
- Vectorize across those butterflies

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
  - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
  - Use double-precision floats to represent coefficients

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
  - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
  - Use double-precision floats to represent coefficients
- Alkim, Ducas, Pöppelmann, Schwabe, 2016:
  - 8448 Haswell cycles ($n = 1024$, 14-bit q)
  - Still use doubles

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
  - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
  - Use double-precision floats to represent coefficients
- Alkim, Ducas, Pöppelmann, Schwabe, 2016:
  - 8448 Haswell cycles ($n = 1024$, 14-bit q)
  - Still use doubles
- Longa, Naehrig, 2016:
  - 9100 Haswell cycles ($n = 1024$, 14-bit q)
  - Uses vectorized integer arithmetic

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
  - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
  - Use double-precision floats to represent coefficients
- Alkim, Ducas, Pöppelmann, Schwabe, 2016:
  - 8448 Haswell cycles ($n = 1024$, 14-bit q)
  - Still use doubles
- Longa, Naehrig, 2016:
  - 9100 Haswell cycles ($n = 1024$, 14-bit q)
  - Uses vectorized integer arithmetic
- Seiler, 2018:
  - 2784 Haswell cycles ($n = 1024$, 14-bit q)
  - 460 Haswell cycles ($n = 256$, 13-bit q)
  - Uses vectorized integer arithmetic

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: “symmetric” parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: “symmetric” parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs
- Typical hash construction:
  - Process message in blocks
  - Each block modifies an internal state
  - Cannot vectorize across blocks

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: "symmetric" parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs
- Typical hash construction:
  - Process message in blocks
  - Each block modifies an internal state
  - Cannot vectorize across blocks
- Idea: Vectorize internal processing (permutation or compression function)
- Two problems:
  - Often strong dependencies between instructions
  - Need limited instruction-level parallelism for pipelining

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: “symmetric” parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs
- Typical hash construction:
  - Process message in blocks
  - Each block modifies an internal state
  - Cannot vectorize across blocks
- Idea: Vectorize internal processing (permutation or compression function)
- Two problems:
  - Often strong dependencies between instructions
  - Need limited instruction-level parallelism for pipelining
- Consequence: consider designing with parallel hash/XOF calls!

PQCRYPTO $\neq$ Lattices

- So far we've looked at lattices, how about other PQCRYPTO?
- Code-based crypto (and some $\mathcal{MQ}$-based crypto) need binary-field arithmetic
- Typical: operations in $\mathbb{F}_{2^k}$ for $k \in 1, \ldots, 20$

PQCRYPTO $\neq$ Lattices

- So far we've looked at lattices, how about other PQCRYPTO?
- Code-based crypto (and some $\mathcal{MQ}$-based crypto) need binary-field arithmetic
- Typical: operations in $\mathbb{F}_{2^k}$ for $k \in 1, \ldots, 20$
- Most architectures don't support this efficiently
- Traditional approach: use lookups (log tables)

PQCRYPTO $\neq$ Lattices

- So far we've looked at lattices, how about other PQCRYPTO?
- Code-based crypto (and some $\mathcal{MQ}$-based crypto) need binary-field arithmetic
- Typical: operations in $\mathbb{F}_{2^k}$ for $k \in 1, \ldots, 20$
- Most architectures don't support this efficiently
- Traditional approach: use lookups (log tables)
- Obvious question: can vector operations help?

- So far: vectors of bytes, 32-bit words, floats,...
- Consider now vectors of bits

Bitslicing

- So far: vectors of bytes, 32-bit words, floats,...
- Consider now vectors of bits
- Perform arithmetic on those vectors using `XOR`, `AND`, `OR`
- "Simulate hardware implemenations in software"

Bitslicing

- So far: vectors of bytes, 32-bit words, floats,...
- Consider now vectors of bits
- Perform arithmetic on those vectors using `XOR`, `AND`, `OR`
- "Simulate hardware implemenations in software"
- Technique was introduced by Biham in 1997 for DES
- Bitslicing works for every algorithm
- *Efficient* bitslicing needs a huge amount of data-level parallelism

4-coefficient binary polynomials

$(a_3x^3 + a_2x^2 + a_1x + a_0)$, with $a_i \in \{0, 1\}$

4-coefficient bitsliced binary polynomials

```
typedef unsigned char poly4; /* 4 coefficients in the low 4 bits */
typedef unsigned long long poly4x64[4];

void poly4_bitslice(poly4x64 r, const poly4 x[64])
{
  int i,j;
  for(i=0;i<4;i++)
  {
    r[i] = 0;
    for(j=0;j<64;j++)
      r[i] |= (unsigned long long)(1 & (x[j] >> i))<<j;
  }
}
```

Bitsliced binary-polynomial multiplication

```
typedef unsigned long long poly4x64[4];
typedef unsigned long long poly7x64[7];

void poly4x64_mul(poly7x64 r, const poly4x64 a, const poly4x64 b)
{
  r[0] = a[0] & b[0];
  r[1] = (a[0] & b[1]) ^ (a[1] & b[0]);
  r[2] = (a[0] & b[2]) ^ (a[1] & b[1]) ^ (a[2] & b[0]);
  r[3] = (a[0] & b[3]) ^ (a[1] & b[2]) ^ (a[2] & b[1]) ^ (a[3] & b[0]);
  r[4] = (a[1] & b[3]) ^ (a[2] & b[2]) ^ (a[3] & b[1]);
  r[5] = (a[2] & b[3]) ^ (a[3] & b[2]);
  r[6] = (a[3] & b[3]);
}
```

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$
- Higher level:
  - Additive FFT for efficient root finding
  - Transposed FFT for syndrome computation
  - Batcher sort for random permutations

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$
- Higher level:
  - Additive FFT for efficient root finding
  - Transposed FFT for syndrome computation
  - Batcher sort for random permutations
- Results:
  - 75 935 744 Ivy Bridge cycles for 256 decodings at $\approx$ 256-bit pre-quantum security
  - **Not** $75\,935\,744/256 = 296\,624$ cycles for one decoding
  - Reason: Need 256 independent decodings for parallelism

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$
- Higher level:
    - Additive FFT for efficient root finding
    - Transposed FFT for syndrome computation
    - Batcher sort for random permutations
- Results:
    - 75 935 744 Ivy Bridge cycles for 256 decodings at $\approx$ 256-bit pre-quantum security
    - **Not** $75\,935\,744/256 = 296\,624$ cycles for one decoding
    - Reason: Need 256 independent decodings for parallelism
- Chou, CHES 2017: use *internal* parallelism
    - Target even higher security (297 bits pre-quantum)
    - Does *not* require independent decryptions
    - Even faster, even when considering throughput

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable
- Distinguish 3 (or 4) different cases, depending on the field
- $\mathbb{F}_{31}$: 16-bit-word vector elements, use integer arithmetic

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable
- Distinguish 3 (or 4) different cases, depending on the field
- $\mathbb{F}_{31}$: 16-bit-word vector elements, use integer arithmetic
- $\mathbb{F}_2/\mathbb{F}_4$: Use bitslicing (see Joost's talk)

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable
- Distinguish 3 (or 4) different cases, depending on the field
- $\mathbb{F}_{31}$: 16-bit-word vector elements, use integer arithmetic
- $\mathbb{F}_2/\mathbb{F}_4$: Use bitslicing (see Joost's talk)
- $\mathbb{F}_{16}/\mathbb{F}_{256}$: Use vector-permute instructions for table lookups
- For $\mathbb{F}_{256}$ use tower-field arithmetic on top of $\mathbb{F}_{16}$

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2016:
  64 eqns in 64 vars over $\mathbb{F}_{31}$: 6616 Haswell cycles

Recent $\mathcal{MQ}$ results

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2016:
  64 eqns in 64 vars over $\mathbb{F}_{31}$: 6616 Haswell cycles
- Chen, Li, Peng, Yang, Cheng, 2017:
  - 256 eqns in 256 vars over $\mathbb{F}_2$: 92800 Haswell cycles
  - 128 eqns in 128 vars over $\mathbb{F}_4$: 32300 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{16}$: 9600 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{31}$: 8700 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{256}$: 16200 Haswell cycles
  - In particular for $\mathbb{F}_2$ speedups for public inputs

Recent $\mathcal{MQ}$ results

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2016:
  64 eqns in 64 vars over $\mathbb{F}_{31}$: 6616 Haswell cycles
- Chen, Li, Peng, Yang, Cheng, 2017:
  - 256 eqns in 256 vars over $\mathbb{F}_2$: 92800 Haswell cycles
  - 128 eqns in 128 vars over $\mathbb{F}_4$: 32300 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{16}$: 9600 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{31}$: 8700 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{256}$: 16200 Haswell cycles
  - In particular for $\mathbb{F}_2$ speedups for public inputs
- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2017:
  128 eqns in 128 vars over $\mathbb{F}_4$: 17 558 Haswell cycles (batched)

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?
- Most speed-critical operation is Winternitz public-key computation
- Compute 67 independent hash chains of length 15 each
- All hashes have the same (short) input length
- This is trivially vectorizable!

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?
- Most speed-critical operation is Winternitz public-key computation
- Compute 67 independent hash chains of length 15 each
- All hashes have the same (short) input length
- This is trivially vectorizable!
- Examples:
  - Oliveira, López, Cabral, 2017: Optimize LMS and XMSS
  - $\approx$ 10ms for XMSS signing ($h = 20$) on Skylake

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?
- Most speed-critical operation is Winternitz public-key computation
- Compute 67 independent hash chains of length 15 each
- All hashes have the same (short) input length
- This is trivially vectorizable!
- Examples:
  - Oliveira, López, Cabral, 2017: Optimize LMS and XMSS
  - $\approx$ 10ms for XMSS signing ($h = 20$) on Skylake
  - Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe, Wilcox-O'Hearn, 2015: Optimize SPHINCS
  - Vectorize also Merkle-tree hashes inside HORST computation
  - $\approx$ 52 Mio cycles for signing on Haswell

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

2. Branches

$$v \leftarrow (c[0]?a : b, c[1]?c : d, c[2]?e : f, c[3]?g : h)$$

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

2. Branches

$$v \leftarrow (c[0]?a : b, c[1]?c : d, c[2]?e : f, c[3]?g : h)$$

Rethink algorithms

- Consequence: rethink algorithms without those constructs
- Different approach to thinking algorithms: a lot of fun!

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

2. Branches

$$v \leftarrow (c[0]?a : b, c[1]?c : d, c[2]?e : f, c[3]?g : h)$$

Rethink algorithms

- Consequence: rethink algorithms without those constructs
- Different approach to thinking algorithms: a lot of fun!
- More importantly: eliminates most notorious timing side channels!
- Efficient vectorized implementations are often also "constant-time"

References

- Alkim, Bindel, Buchmann, Dagdelen, Schwabe: *TESLA: Tightly-Secure Efficient Signatures from Standard Lattices.* https://cryptojedi.org/papers/#tesla (superseded by https://eprint.iacr.org/2015/755)
- Bernstein, Chuengsatiansup, Lange, van Vredendaal: *NTRU Prime: reducing attack surface at low cost.* http://cr.yp.to/papers.html#ntruprime
- Hülsing, Rijneveld, Schanck, Schwabe: *High-speed key encapsulation from NTRU.* https://cryptojedi.org/papers/#ntrukem

References

- Güneysu, Oder, Pöppelmann, Schwabe: *Software speed records for lattice-based signatures.* https://cryptojedi.org/papers/#lattisigns
- Alkim, Ducas, Pöppelmann, Schwabe: *Post-quantum key exchange – a new hope.* https://cryptojedi.org/papers/#newhope
- Longa, Naehrig: *Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography.* https://eprint.iacr.org/2016/504
- Seiler: *Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography* https://eprint.iacr.org/2018/039

References

- Bernstein, Chou, Schwabe: *McBits: fast constant-time code-based cryptography.* https://cryptojedi.org/papers/#mcbits
- Chou: *McBits revisited.* https://eprint.iacr.org/2017/793

References

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe: *From 5-pass MQ-based identification to MQ-based signatures.* https://cryptojedi.org/papers/#mqdss
- Chen, Li, Peng, Yang, Cheng: *Implementing 128-bit Secure MPKC Signatures.* https://eprint.iacr.org/2017/636
- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe: *SOFIA: MQ-based signatures in the QROM.* https://cryptojedi.org/papers/#sofia

References

- Oliveira, López, Cabral: *High Performance of Hash-based Signature Schemes* http://thesai.org/Publications/ViewPaper?Volume=8&Issue=3&Code=IJACSA&SerialNo=58
- Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe, Wilcox-O'Hearn: *SPHINCS: practical stateless hash-based signatures.* https://cryptojedi.org/papers/#sphincs