Optimizing ML-KEM and ML-DSA with Vector Instructions and Mathematical Techniques

OpenSSL Conference 2026, Prague

Timo Keller

October 14, 2026

IBM Deutschland Research & Development GmbH

Outline

1. What do we optimize?
2. How do we optimize it?
3. Results

PQC with ML-KEM and ML-DSA

The need for post-quantum cryptography (PQC)

- Shor: quantum polynomial-time integer factorization and discrete logarithm (via HSP for f.g. abelian groups using QFT)
- breaks RSA, DH, and ECDH (used for key exchange)!
- The IBM quantum computer might have enough qubits in a few years!
- There are two FIPS-standardized lattice-based PQC algorithms: ML-KEM and ML-DSA
- They need to run as fast as possible and in constant time.

ML-KEM and ML-DSA: target of our optimizations

$$R_q := \mathbf{F}_q[X]/(X^n + 1)$$

ML-KEM (FIPS 203, key encapsulation)
Public key: $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}, \quad \mathbf{A} \in R_q^{k \times k}$
Encaps/Decaps:

- compute $\mathbf{A}\mathbf{y}$ and $\mathbf{t}^T\mathbf{y}$
- entries are sums of multiplications in R_q

ML-DSA (FIPS 204, signature)
Public key: $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2, \quad \mathbf{A} \in R_q^{k \times \ell}$
Sign/Verify:

- compute $\mathbf{A}\mathbf{y}$, $\mathbf{A}\mathbf{z} - c\mathbf{t}$
- entries are sums of multiplications in R_q

Common bottleneck: matrix-vector multiplications over R_q

Naive R_q-multiplication: $n^2 = 65\,536$ modular multiplications.
With NTT: $O(n \log_2 n)$ modular multiplications.
All optimizations in this talk target this bottleneck.

Overview of our optimizations

Two types of optimizations

1. Vectorization: architectural, specific to 128-bit vector registers (8 16 bit and 4 32 bit integers in parallel)
2. Mathematical: algebraic identities, mainly aimed at reducing the number of multiplications

Important: use signed integers.

Prior art

[1] Harvey 2012: remove some reductions in the butterfly units of (i)NTT by using partial reductions; (i)NTT implementation on x86

[2] Longa–Naehrig 2016: introduce k-reduction; implementation with AVX2 vectors of uint32's

[3] Seiler 2018: signed Montgomery and Barrett reduction, partial reduction for $q = 2^m - 2^n + 1$ (Dilithium/ML-DSA), lazy reductions; vector implementation with AVX2

[4] Bradbury–Hess 2021: Karatsuba multiplication for $\mathsf{F}_p[\sqrt{-1}]$; implementation of Dilithium on Z using vectors of uint32's

[5] Becker et al. 2022: explore the relation between Montgomery reduction/multiplication and Barrett reduction; implementation for ARM using vectors of int32's

Fast multiplication in R_q

- Ring

$$R_q := \mathsf{F}_q[X]/(X^n + 1)$$

  ML-KEM: $q = 13 \cdot 2^8 + 1 < 2^{15}$, ML-DSA: $q = 2^{23} - 2^{13} + 1 < 2^{31}$, both $n = 2^8$.

- ML-KEM with $R_q \cong \prod_{i=1}^{n/2} \mathsf{F}_q[X]/(X^2 - \zeta_i) \cong \mathsf{F}_{q^2}^{n/2}$:

$$\begin{array}{ccc} R_q \times R_q & \xrightarrow{NTT \times NTT} & \mathsf{F}_{q^2}^{n/2} \times \mathsf{F}_{q^2}^{n/2} \\ \Big\downarrow{\scriptstyle (a,b) \mapsto a \cdot b} & & \Big\downarrow{\scriptstyle (\hat{a},\hat{b}) \mapsto (\hat{a}_i \cdot \hat{b}_i)_i} \\ R_q & \xleftarrow{iNTT} & \mathsf{F}_{q^2}^{n/2} \end{array}$$

  On the right-hand side, one has the element-wise product with $n/2$ multiplications in F_{q^2}.

Modular arithmetic

Barrett reduction and Montgomery multiplication

Type	Input conditions	Output guarantee	Used for	# mults
Barrett	$a \in [-8q,\ 8q]$	$r \equiv a \pmod q$, $r \in [0, q)$[1]	Reduction	2
Mont_2	$a \in [0, q)$, $\lvert b\rvert \leq 8q$	$r \equiv abR_M^{-1} \pmod q$, $\lvert r\rvert < q$	Mult. in Montgomery form	4
Mont_1	$a \in [0, q)$, b, b_{tw} precomp	$r \equiv ab \pmod q$, $r \in (-q, q)$	Reduced mult. (b fixed)	3

$R_M := 2^{16}$, $R_M^{-1} \equiv q^{-1} \pmod{2^{16}}$ (Montgomery factor), $b_{\text{tw}} := b\, q_{\text{inv}} \bmod 2^{16}$ with $q_{\text{inv}} := -3327$.

Montgomery form $aR_M \pmod q$, Montgomery product $(aR_M)(bR_M)R_M^{-1} = abR_M \pmod q$.

[1] reduce once: reduce $x \in (-q, q)$ to $[0, q)$ via $\min((\texttt{uint16})\,x,\ (\texttt{uint16})\,x + q)$ using IBM Z's `vec_min`: constant-time! (Existing approaches use masking and bit operations.)

A one-line optimization

```
#define BARRETT_SHIFT 32 // before: 24
```

Barrett reduction (ML-KEM) (with $m := \left\lfloor \frac{2^s}{q} \right\rfloor$ precomputed):

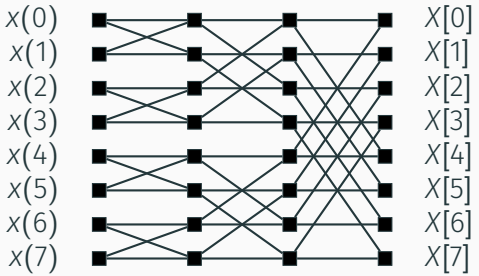
$$\widehat{q} := \left\lfloor \frac{x\,m}{2^s} \right\rfloor, \qquad r := \text{reduce_once}(x - \widehat{q}q).$$

Correctness only requires Barrett shift $s \geq 24 = 2\lceil \log_2 q \rceil$ (and no overflow).

- Barrett shift $s = 32$ makes $(xm) \gg 32$ the high half of a 32-by-32-bit product.
- Hence we need only 2 instead of 3 int32 multiplications.

The number theoretic transform (NTT), ML-KEM

The Number Theoretic Transform (NTT)



$\log_2 n$ levels each with $n/2$ independent butterfly units.

with butterfly units

$$X_i = x_i + \zeta \cdot x_j$$
$$X_j = x_i - \zeta \cdot x_j$$

for the NTT and

$$X_i = x_i + x_j$$
$$X_j = \zeta \cdot (x_i - x_j)$$

for the iNTT (for certain indices i, j and roots of unity ζ)

Some general optimization strategies

- vectorization: The butterflies in one level are independent, and can therefore be run in parallel.
- lazy reductions: Reduce mod q only when necessary to prevent overflow.
- partial reductions: Only reduce mod kq, not mod q.

Forward NTT (ML-KEM)

Layer	$\lvert x_j \rvert$	$\lvert \zeta \cdot x_j \rvert$	$\lvert x_i \pm \zeta \cdot x_j \rvert$
1	$< q$	$< q$	$< 2q$
2	$< 2q$	$< q$	$< 3q$
3	$< 3q$	$< q$	$< 4q$
4	$< 4q$	$< q$	$< 5q$
5	$< 5q$	$< q$	$< 6q$
6	$< 6q$	$< q$	$< 7q$
7	$< 7q$	$< q$	$< 8q$

- Since $8q = 26632 < 32768 = 2^{15}$, lazy reduction suffices throughout.
- Full reduction only in the last layer.
- Last two layers use `vec_perm` (in the other layers elements combined in butterflies are ≥ 8 elements apart).

Inverse NTT (ML-KEM)

The butterfly computes

$$X_i = x_i + x_j, \qquad X_j = \zeta(x_i - x_j) = \mathbf{Mont}_1(x_i - x_j, \zeta) \in (-q, q).$$

Layer	$\lvert x_i + x_j \rvert$	Action
1	$< 2q$	—
2	$< 4q$	—
3	$< 8q$	reduce to $[0, q)$
4	$< 2q$	—
5	$< 4q$	—
6	$< 8q$	reduce to $[0, q)$
7	$< 2q$	final multiplication by $(n/2)^{-1}$

The twisted Karatsuba multiplication

Twisted Karatsuba multiplication

We use the following technique to speed up the multiplication in F_{q^2}:

- naive multiplication in $\mathsf{F}_{q^2} \cong \mathsf{F}_q[\sqrt{\zeta}]$:
  $(a_1\sqrt{\zeta} + a_0)(b_1\sqrt{\zeta} + b_0) = (a_1b_0 + a_0b_1)\sqrt{\zeta} + (a_0b_0 + \zeta a_1b_1)$
  (5 mod q multiplications, in total 19 int16 multiplications).
- **new: "twisted Karatsuba multiplication"** (in Montgomery form):
  Karatsuba multiplication is for $\mathbb{Z}$. We use and modify the idea for F_{q^2}.
  Instead compute $P := (a_1 + a_0)(b_1 + b_0), P_0 := a_0b_0, P_1 := a_1b_1$.
  Then $(a_1b_0 + a_0b_1) = P - P_0 - P_1$ and $(a_0b_0 + \zeta a_1b_1) = P_0 + \zeta P_1$.
  (4 mod q multiplications, in total **15 int16 multiplications** (can reduce to 14))
- Hence we need 26 % **fewer multiplications**.

Deferring demontgomerization

In ML-KEM, we have to compute scalar products and matrix-vector products of rank $k \leq 4$.

1. Compute products (in $\mathbf{F}_{q^2}^{n/2}$) in Montgomery form.
2. Since the rank $k \leq 4$, we can use lazy reductions:

$$\left| \sum_{i=1}^{k} (\text{output of twisted Karatsuba}) \right| \leq k \cdot 6007 \leq 24028 < 2^{15}$$

3. Demontgomerize (and iNTT) when needed at the end.

Differences for ML-DSA

Our optimization techniques for ML-DSA

Differences from ML-KEM:

- $q = 2^{23} - 2^{13} + 1$
- We must use vectors of int32's.
- Only final reduction in (i)NTT: values remain in range $[-9q, 9q]$ (NTT) valid for reduction; $[-2^8 q, 2^8 q]$ for Montgomery reduction in iNTT.
- $512 \mid (q-1)$, hence $X^n + 1$ splits into linear factors over F_q, so we have scalar multiplications in F_q.

Implementation

Implementation

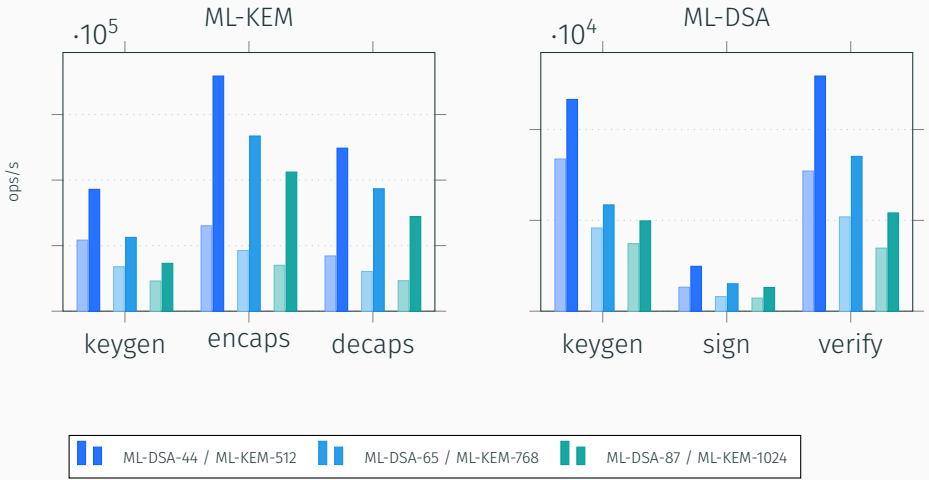
- Implemented in C with vector extensions and `vec_mulh` (e.g. POWER).
- Hashes have hardware support through CPACF on Z.
- Exploiting the special shape of q for reductions did not yield a speedup.
- Formal verification of ML-KEM optimization in Lean.
- Optimizations pass valgrind constant-time test.
- gcc outputs data-independent code.
- AI usage:
  - generation of comments
  - extraction of permutation indices for `vec_perm` in (i)NTT
  - changes to build system

Thanks

I thank Mounir Idrassi for a very thorough code review!

Measurements

Performance (gcc 16): ML-KEM & ML-DSA



gcc 16 `-O3 -march=native -fomit-frame-pointer` on a z17; light = autovec, dark = our optimized code

Thank you!

Questions?