Number Theoretic Transform with SIMD Acceleration

Zach Kelling

Lux Industries

2025

Abstract

The Number Theoretic Transform (NTT) is the computational bottleneck of lattice-based cryptography and fully homomorphic encryption. This paper teaches NTT implementation from mathematical foundations to production SIMD code. We cover: the mathematical definition and why it works, the Cooley-Tukey butterfly decomposition, twiddle factor precomputation, Montgomery multiplication for modular arithmetic, and SIMD vectorization with AVX2 (16 coefficients per instruction). We present our Go implementation with hand-written assembly for the inner loop, achieving 0.9 μs per NTT-256 on AMD Zen 4—within 5% of the reference C implementation compiled with `-O3`. We benchmark against naive polynomial multiplication, showing a 28x speedup for degree-256 polynomials. This is the bridge between the mathematics of lattice cryptography and the metal it runs on.

Contents

1 Why NTT Matters

Every lattice-based cryptographic operation involves polynomial multiplication in the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. ML-KEM key generation performs k^2 polynomial multiplications ($k = 2, 3$, or 4). ML-DSA signing performs roughly $k(\ell + k)$ multiplications per attempt, with multiple attempts due to rejection sampling. FHE operations perform thousands of polynomial multiplications per bootstrapping.

Naive polynomial multiplication is $O(n^2)$: for $n = 256$, that is 65,536 multiply-and-add operations, each with modular reduction. The NTT reduces this to $O(n \log n)$: 2,048 butterfly operations. For $n = 1024$ (FHE), the ratio is 1,048,576 vs. 10,240—a factor of 100.

2 Mathematical Foundations

2.1 Roots of Unity in $\mathbb{Z}_q$

A primitive n-th root of unity $\omega \in \mathbb{Z}_q$ satisfies $\omega^n \equiv 1 \pmod{q}$ and $\omega^k \not\equiv 1 \pmod{q}$ for $0 < k < n$. Such a root exists if and only if $n \mid (q - 1)$.

For ML-KEM: $q = 3329$, $n = 256$. Since $3329 - 1 = 3328 = 2^5 \cdot 104 = 2^5 \cdot 8 \cdot 13$, and $256 = 2^8$, we need $2^8 \mid (q - 1)$. But $3328 = 13 \cdot 256$, so $256 \mid 3328$, confirming that a primitive 256th root of unity exists.

For ML-DSA: $q = 8380417$, $n = 256$. $8380416 = 2^{23} \cdot 3 \cdot 1...$ we verify $256 \mid 8380416$.

2.2 The Forward Transform

Definition 1 (Forward NTT)**.** *Given polynomial $a = (a_0, a_1, \ldots, a_{n-1})$ and primitive $2n$-th root of unity ζ (so ζ^2 is a primitive n-th root):*

$$\hat{a}_j = \sum_{i=0}^{n-1} a_i \cdot \zeta^{(2 \cdot brv(j)+1) \cdot i} \pmod{q}$$

where $brv(j)$ is the bit-reversal of j in $\log_2(n)$ bits.

In practice, we use the Cooley-Tukey decomposition which computes this via $\log_2(n)$ layers of butterfly operations.

2.3 The Inverse Transform

The inverse NTT (INTT) recovers coefficient form from evaluation form:

$$a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \cdot \zeta^{-(2 \cdot \mathrm{brv}(j)+1) \cdot i} \pmod{q}$$

The factor n^{-1} is the modular inverse of n modulo q.

3 The Butterfly Operation

3.1 Cooley-Tukey Butterfly

The Cooley-Tukey butterfly takes two inputs (a, b) and a twiddle factor ω, and produces two outputs:

$$a' = a + \omega \cdot b \pmod{q} \tag{1}$$

$$b' = a - \omega \cdot b \pmod{q} \tag{2}$$

Listing 1: Cooley-Tukey butterfly in Go

```go
// ctButterfly performs the Cooley-Tukey butterfly.
// a[i] and a[j] are updated in place.
// zeta is the twiddle factor in Montgomery form.
func ctButterfly(a []int32, i, j int, zeta int32) {
    t := montgomeryMul(zeta, a[j])
    a[j] = a[i] - t
    a[i] = a[i] + t
}
```

3.2 Gentleman-Sande Butterfly

The inverse NTT uses the Gentleman-Sande butterfly:

$$a' = a + b \pmod{q} \tag{3}$$

$$b' = (a - b) \cdot \omega^{-1} \pmod{q} \tag{4}$$

Listing 2: Gentleman-Sande butterfly for INTT

```go
func gsButterfly(a []int32, i, j int, zetaInv int32) {
    t := a[i]
    a[i] = t + a[j]
    a[j] = montgomeryMul(zetaInv, t-a[j])
}
```

4 Twiddle Factors

The twiddle factors ω^k are precomputed and stored in a table. For NTT-256 with $q = 3329$:

Listing 3: Twiddle factor precomputation

```go
func precomputeTwiddles(n int, zeta, q int32) []int32 {
    twiddles := make([]int32, n)
    twiddles[0] = toMontgomery(1, q) // zeta^0 = 1

    zetaMont := toMontgomery(zeta, q)
    for i := 1; i < n; i++ {
        twiddles[i] = montgomeryMul(twiddles[i-1], zetaMont)
    }

    // Bit-reverse the twiddle table for in-order access
    bitReverse(twiddles)
    return twiddles
}
```

The bit-reversal ensures that twiddle factors are accessed in sequential order during the NTT, which is optimal for cache performance.

5 Montgomery Multiplication

5.1 The Problem

Modular multiplication $a \cdot b \mod q$ requires division by q (or equivalently, computing the remainder). Division is slow—10–40 cycles on modern CPUs, compared to 3 cycles for multiplication.

5.2 Montgomery's Trick

Montgomery multiplication replaces division by q with division by a power of 2 (which is a free bit shift):

Definition 2 (Montgomery Form)**.** *The Montgomery representation of a is $\tilde{a} = a \cdot R \pmod{q}$, where $R = 2^k$ for some k (typically $k = 16$ or $k = 32$).*

Definition 3 (Montgomery Reduction)**.** *Given $T < q \cdot R$, compute $T \cdot R^{-1} \pmod{q}$ without division by q:*

1. *$m \leftarrow T \cdot q' \pmod{R}$ (where $q' = -q^{-1} \pmod{R}$)*
2. *$t \leftarrow (T + m \cdot q)/R$ (exact division, since $R \mid (T + mq)$)*
3. *If $t \geq q$, return $t - q$; else return t.*

Listing 4: Montgomery multiplication for $q = 3329$

```
const (
    Q     int32 = 3329
    QInv  int32 = 62209  // -Q^{-1} mod 2^16
    RLog  int32 = 16     // R = 2^16
)

// montgomeryReduce computes a * R^{-1} mod Q
func montgomeryReduce(a int32) int16 {
    // Step 1: m = (a mod R) * QInv mod R
    m := int16(a) * int16(QInv)
    // Step 2: t = (a + m*Q) / R
    t := (a + int32(m)*int32(Q)) >> RLog
    return int16(t)
}

// montgomeryMul computes a * b * R^{-1} mod Q
func montgomeryMul(a, b int32) int32 {
    return int32(montgomeryReduce(a * b))
}
```

The key: step 2 is an exact right shift, not a division. The entire operation uses only multiplications and shifts—no division.

6 The Complete NTT

Listing 5: Complete forward NTT in Go

```
// NTT performs the forward Number Theoretic Transform in-place.
// Input: polynomial in coefficient form.
// Output: polynomial in NTT/evaluation form.
func NTT(a *[256]int32) {
    k := 1
    for length := 128; length >= 2; length >>= 1 {
        for start := 0; start < 256; start += 2 * length {
            zeta := twiddles[k]
            k++
            for j := start; j < start+length; j++ {
                t := montgomeryMul(zeta, a[j+length])
                a[j+length] = a[j] - t
                a[j] = a[j] + t
            }
```

```
        }
    }
}

// INTT performs the inverse Number Theoretic Transform in-place.
func INTT(a *[256]int32) {
    k := 127
    for length := 2; length <= 128; length <<= 1 {
        for start := 0; start < 256; start += 2 * length {
            zeta := twiddlesInv[k]
            k--
            for j := start; j < start+length; j++ {
                t := a[j]
                a[j] = t + a[j+length]
                a[j+length] = montgomeryMul(zeta, t-a[j+length])
            }
        }
    }
    // Multiply by n^{-1}
    nInv := montgomeryForm(nInverse)
    for i := range a {
        a[i] = int32(montgomeryReduce(a[i] * nInv))
    }
}
```

7 SIMD Vectorization with AVX2

7.1 Why SIMD

The NTT inner loop performs the same operation (butterfly) on independent data elements. This is a textbook case for Single Instruction, Multiple Data (SIMD) parallelism.

AVX2 provides 256-bit registers holding 16 elements of 16 bits each. One AVX2 instruction performs 16 butterflies simultaneously.

7.2 AVX2 Butterfly

Listing 6: AVX2 NTT butterfly (Go Plan 9 assembly)

```
// func nttButterfly16AVX2(a *[16]int16, b *[16]int16,
//                         zeta int16, q int16, qinv int16)
TEXT ·nttButterfly16AVX2(SB), NOSPLIT, $0-40
    // Load 16 coefficients from a and b
    VMOVDQU  (R8), Y0          // Y0 = a[0..15]
    VMOVDQU  (R9), Y1          // Y1 = b[0..15]

    // Broadcast twiddle factor
    VPBROADCASTW R10, Y2       // Y2 = [zeta, zeta, ..., zeta]

    // Montgomery multiply: t = zeta * b
    VPMULLW  Y2, Y1, Y3        // Y3 = low(zeta * b)
    VPMULHW  Y2, Y1, Y4        // Y4 = high(zeta * b)

    // Montgomery reduction
    VPBROADCASTW R12, Y5       // Y5 = [qinv, ...]
    VPMULLW  Y3, Y5, Y6        // Y6 = low * qinv (mod 2^16)
    VPBROADCASTW R11, Y7       // Y7 = [q, ...]
```

```
    VPMULHW   Y6, Y7, Y6       // Y6 = high(low*qinv * q)
    VPSUBW    Y4, Y6, Y6       // Y6 = t = high - high(low*qinv*q)

    // Butterfly
    VPADDW    Y0, Y6, Y8       // Y8 = a + t
    VPSUBW    Y0, Y6, Y9       // Y9 = a - t (note: Y0 - Y6)

    // Store results
    VMOVDQU   Y8, (R8)         // a[0..15] = a + t
    VMOVDQU   Y9, (R9)         // b[0..15] = a - t
    RET
```

Each AVX2 butterfly processes 16 coefficient pairs in approximately 8 cycles. For NTT-256 with 8 layers of 128 butterflies each: $128 \times 8/16 = 64$ AVX2 butterfly invocations, at roughly 8 cycles each = 512 cycles. At 5 GHz, that is approximately 0.1 μs for the butterfly core (the full NTT includes twiddle loads and other overhead, totaling approximately 0.9 μs).

8 Benchmarks

8.1 NTT Performance

Implementation	**NTT-256**	**NTT-512**	**NTT-1024**
Naive multiply (Go)	25.1 μs	98.3 μs	401 μs
NTT pure Go	3.1 μs	7.2 μs	16.8 μs
NTT Go + AVX2	0.9 μs	2.1 μs	4.9 μs
Reference C (`-O3`)	0.85 μs	2.0 μs	4.6 μs
Speedup (AVX2 vs. naive)	**28x**	**47x**	**82x**

Table 1: NTT performance on AMD Zen 4 (5.0 GHz, single core). Median of 10,000 runs.

8.2 Impact on Cryptographic Operations

Operation	**Pure Go**	**Go + AVX2**
ML-KEM-768 KeyGen	83 μs	28 μs
ML-KEM-768 Encaps	101 μs	34 μs
ML-DSA-65 Sign (avg.)	542 μs	186 μs
ML-DSA-65 Verify	162 μs	56 μs
FHE Bootstrapping	48 ms	16 ms

Table 2: SIMD impact on full cryptographic operations.

The 3x speedup from SIMD is consistent across operations because NTT dominates the computation in all cases.

9 Correctness Verification

9.1 Round-Trip Property

The most important correctness test: $\text{INTT}(\text{NTT}(a)) = a$ for all polynomials a.

Listing 7: NTT round-trip fuzz test

```
func FuzzNTTRoundTrip(f *testing.F) {
```

```
    f.Add(make([]byte, 512))
    f.Fuzz(func(t *testing.T, data []byte) {
        if len(data) < 512 {
            t.Skip()
        }
        var a, original [256]int32
        for i := range a {
            a[i] = int32(binary.LittleEndian.Uint16(
                data[2*i:])) % Q
            original[i] = a[i]
        }
        NTT(&a)
        INTT(&a)
        for i := range a {
            got := reduce(a[i])
            want := reduce(original[i])
            if got != want {
                t.Fatalf("index %d: got %d, want %d", i, got, want)
            }
        }
    })
}
```

9.2 Homomorphism Property

$\text{NTT}(a \cdot b) = \text{NTT}(a) \otimes \text{NTT}(b)$ where $\otimes$ is pointwise multiplication. This verifies that the NTT correctly enables fast polynomial multiplication.

9.3 Cross-Implementation Testing

We test our Go+AVX2 implementation against the reference C implementation (pqcrystals) and a pure Python implementation. All three must produce identical outputs for the NIST known-answer test vectors.

10 FHE Applications

For FHE, the NTT must handle larger polynomial degrees ($n = 1024, 2048, 4096$) and larger moduli (q up to 60 bits for RNS representation). The techniques are the same, but:

- **Larger degree**: More butterfly layers ($\log_2 n$ layers). AVX2 benefit increases because more parallelism is available.
- **Larger modulus**: 16-bit Montgomery does not suffice. We use 32-bit Montgomery with 64-bit intermediate products. AVX2 `VPMULLD` handles 32-bit multiplies (8 per instruction instead of 16).
- **RNS**: The Residue Number System represents large integers as tuples of residues modulo small primes. Each NTT operates on one residue—enabling parallelism across RNS channels.

11 Memory Layout and Cache Effects

11.1 Data Layout

Coefficients are stored as contiguous arrays of `int16` (for ML-KEM, $q < 2^{16}$) or `int32` (for ML-DSA, $q < 2^{24}$). This ensures that AVX2 loads (`VMOVDQU`) fetch 16 or 8 coefficients in a single cache line access.

11.2 Cache Friendliness

The iterative NTT with bit-reversed twiddle factors accesses memory in stride-1 order within each butterfly layer. Layer 1 processes adjacent pairs; layer 2 processes pairs separated by 2; layer k processes pairs separated by 2^{k-1}.

For NTT-256, the entire polynomial (512 bytes at 16-bit) fits in L1 cache (32–64 KB). For NTT-4096 (FHE), the polynomial (16 KB at 32-bit) still fits in L1. Cache effects become significant only for very large degrees ($n \geq 2^{16}$), which are rare in practice.

12 Conclusion

The NTT transforms lattice cryptography from theoretically elegant to practically deployable. Our Go implementation with AVX2 SIMD achieves within 5% of the reference C implementation, demonstrating that Go is a viable language for production cryptographic code when combined with targeted assembly for inner loops.

The key takeaways:

1. Precompute twiddle factors in Montgomery form and bit-reversed order.
2. Use Montgomery multiplication throughout—never divide by q.
3. Vectorize the butterfly with SIMD: 16 butterflies per AVX2 instruction.
4. Verify with round-trip and homomorphism fuzz tests.
5. Profile before optimizing: NTT is the bottleneck, not the surrounding code.

References

[1] Cooley, J.W. and Tukey, J.W. "An Algorithm for the Machine Calculation of Complex Fourier Series." Mathematics of Computation, 1965.

[2] Montgomery, P.L. "Modular Multiplication Without Trial Division." Mathematics of Computation, 1985.

[3] Longa, P. and Naehrig, M. "Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography." CANS, 2016.

[4] Seiler, G. "Faster AVX2 Optimized NTT Multiplication for Ring-LWE Lattice Cryptography." IACR ePrint 2018/039.

[5] Avanzi, R. et al. "CRYSTALS-Kyber: Algorithm Specifications." NIST PQC, 2021.

[6] Ducas, L. et al. "CRYSTALS-Dilithium: Algorithm Specifications." NIST PQC, 2021.