



HIGH PERFORMANCE NTT DESIGN FOR ML-KEM+ML-DSA

CRYPTO CAFE TALK

Bardia Taghavi

Florida Atlantic University

January 2026

- 1 Introduction: The PQC Transition
 - Why Post-Quantum Cryptography?
 - NIST PQC Standardization
 - Standardized PQC Cryptosystems
- 2 NTT: The Key to Efficiency
 - Multiplication in the Ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$
 - What is NTT?
 - Constant Geometry (CG) NTT
- 3 ML-KEM and ML-DSA: Different Moduli, Same Polynomial Structure
- 4 Hardware Implementation Journey
 - LightNTT: Starting with ML-DSA
 - LightNTT: Hardware Architecture
 - KNightCore: Extending to ML-KEM
 - KNightCore: Hardware Architecture
 - HyperPoly: The Unified Solution
- 5 Challenges and Conclusions
 - Key Challenges
 - Future Work
 - Conclusions

INTRODUCTION: THE PQC TRANSITION

WHY POST-QUANTUM CRYPTOGRAPHY?

- ✧ Current cryptography (RSA, ECC, DH) relies on problems that are **hard on classical computers**:
 - ▷ Integer Factoring Problem (RSA)
 - ▷ Discrete Logarithm Problem (DH)
 - ▷ Elliptic Curve Discrete Logarithm Problem (ECC)

But all of them will be broken by quantum computers!

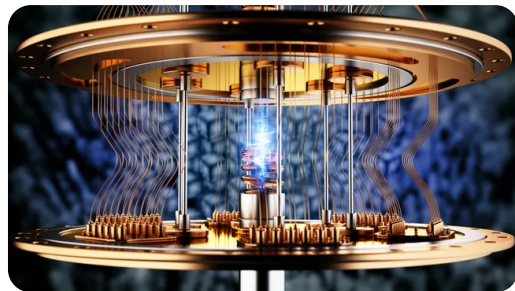
WHY POST-QUANTUM CRYPTOGRAPHY?

- ✧ Current cryptography (RSA, ECC, DH) relies on problems that are **hard on classical computers**:
 - ▷ Integer Factoring Problem (RSA)
 - ▷ Discrete Logarithm Problem (DH)
 - ▷ Elliptic Curve Discrete Logarithm Problem (ECC)

But all of them will be broken by quantum computers!

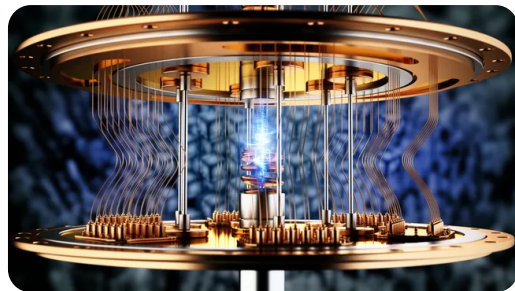
WHY POST-QUANTUM CRYPTOGRAPHY?

- ✧ **Shor's Algorithm** (1994): Solves these problems in polynomial time on quantum computers
- ✧ **The Threat:** When quantum computers become practical, current cryptography breaks
- ✧ **The Solution:** Post-Quantum Cryptography (PQC) - based on problems believed to be hard even for quantum computers



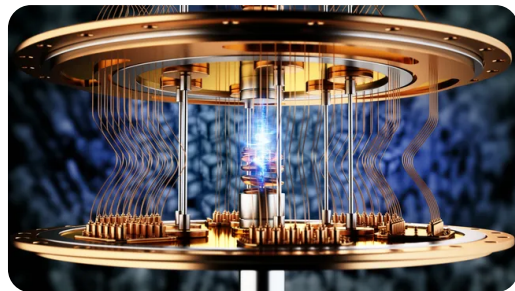
WHY POST-QUANTUM CRYPTOGRAPHY?

- ✧ **Shor's Algorithm** (1994): Solves these problems in polynomial time on quantum computers
- ✧ **The Threat:** When quantum computers become practical, current cryptography breaks
- ✧ **The Solution:** Post-Quantum Cryptography (PQC) - based on problems believed to be hard even for quantum computers



WHY POST-QUANTUM CRYPTOGRAPHY?

- ✧ **Shor's Algorithm** (1994): Solves these problems in polynomial time on quantum computers
- ✧ **The Threat:** When quantum computers become practical, current cryptography breaks
- ✧ **The Solution:** Post-Quantum Cryptography (PQC) - based on problems believed to be hard even for quantum computers



NIST PQC STANDARDIZATION

✧ NIST PQC Standardization Process (2016-2024):

- ▷ 69 initial submissions
- ▷ Multiple rounds of evaluation
- ▷ Final standards of main schemes announced in 2024

✧ Key Winners:

- ▷ **ML-KEM** (FIPS 203): Module-Lattice-Based Key-Encapsulation Mechanism
- ▷ **ML-DSA** (FIPS 204): Module-Lattice-Based Digital Signature Algorithm

✧ Both are based on hard mathematical problems on module-lattices

NIST PQC STANDARDIZATION

✧ NIST PQC Standardization Process (2016-2024):

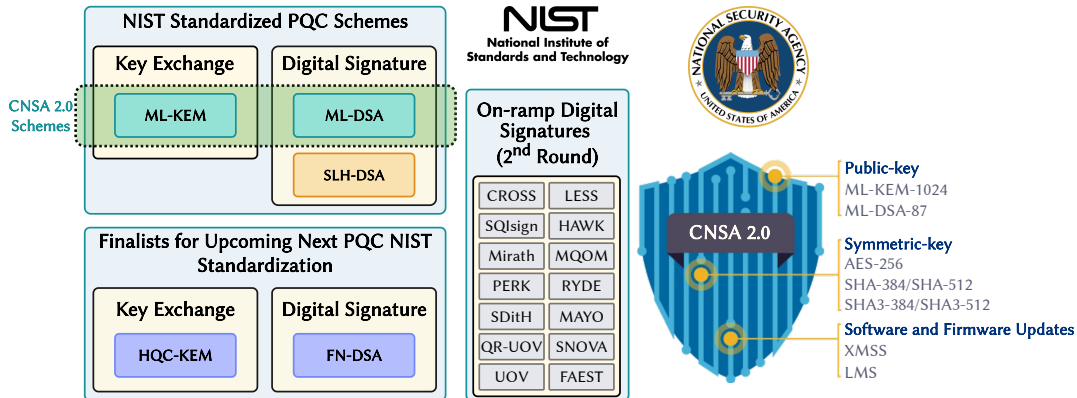
- ▷ 69 initial submissions
- ▷ Multiple rounds of evaluation
- ▷ Final standards of main schemes announced in 2024

✧ **Key Winners:**

- ▷ **ML-KEM** (FIPS 203): Module-Lattice-Based Key-Encapsulation Mechanism
- ▷ **ML-DSA** (FIPS 204): Module-Lattice-Based Digital Signature Algorithm

✧ Both are based on hard mathematical problems on module-lattices

NIST STANDARDIZED PQC CRYPTOSYSTEMS AND CNSA 2.0

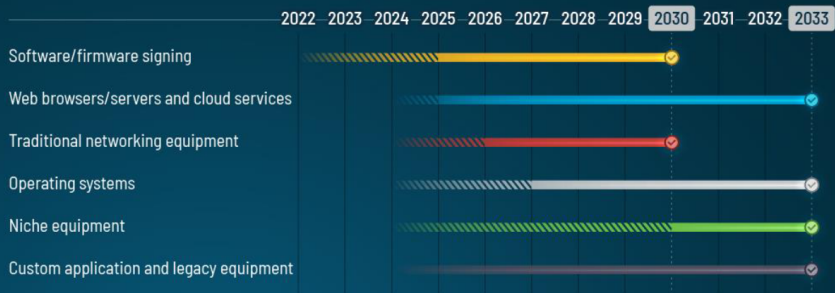


- FIPS-203:** Module-Lattice-Based Key-Encapsulation Mechanism Standard (**ML-KEM**, based on Crystals-Kyber)
- FIPS-204:** Module-Lattice-Based Key-Encapsulation Mechanism Standard (**ML-DSA**, based on Crystals-Dilithium)
- FIPS-205:** Stateless Hash-Based Digital Signature Standard (**SLH-DSA**, based on SPHINCS+)
- FIPS-206:** FFT over NTRU-Lattice-Based Digital Signature Algorithm (**FN-DSA**, based on FALCON)
- FIPS-207:** Hamming Quasi-Cyclic Key-Encapsulation Mechanism Standard (**HQC-KEM**, based on HQC)
- NIST SP 800-208:** Recommendation for Stateful Hash-Based Signature Schemes (**XMSS/LMS**)

CNSA 2.0 TIMELINE

✧ You can read more about CNSA 2.0 in the [CNSA 2.0 FAQ](#) document.

CNSA 2.0 Timeline

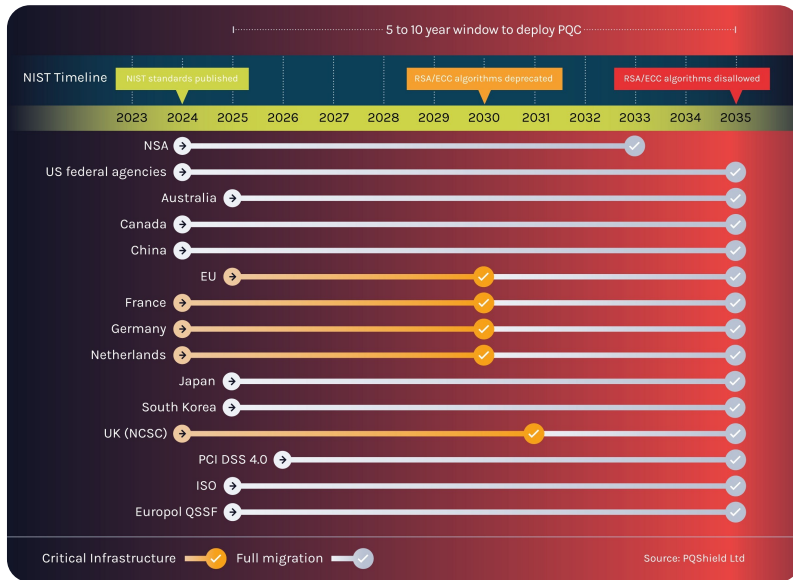


▨ CNSA 2.0 added as an option and tested

▬ CNSA 2.0 as the default and preferred

✓ Exclusively use CNSA 2.0 by this year

PQC Migration Around the World



ML-KEM AND ML-DSA OVERVIEW

ML-KEM

- ✧ Key exchange mechanism
- ✧ Based on MLWE
- ✧ Ring: $\mathcal{R}_q = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$
- ✧ Security levels: AES-128, AES-192, AES-256

ML-DSA

- ✧ Digital signature scheme
- ✧ Based on MLWE + SelfTargetMSIS
- ✧ Ring: $\mathcal{R}_q = \mathbb{Z}_{8380417}[x]/(x^{256} + 1)$
- ✧ Security levels: SHA-256, AES-192, AES-256

Common Challenge

Both require efficient **polynomial multiplication** in $\mathcal{R}_q$ - this is the computational bottleneck!

ML-KEM AND ML-DSA OVERVIEW

ML-KEM

- ✧ Key exchange mechanism
- ✧ Based on MLWE
- ✧ Ring: $\mathcal{R}_q = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$
- ✧ Security levels: AES-128, AES-192, AES-256

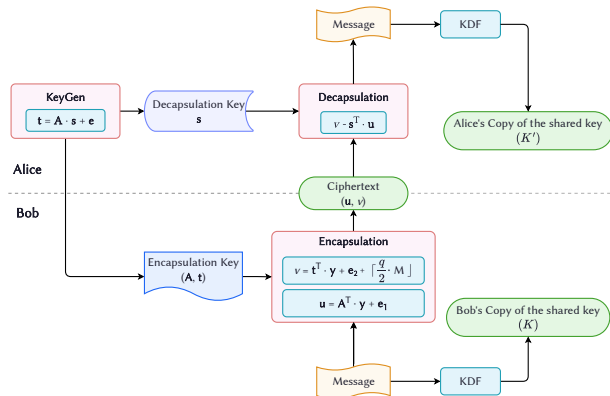
ML-DSA

- ✧ Digital signature scheme
- ✧ Based on MLWE + SelfTargetMSIS
- ✧ Ring: $\mathcal{R}_q = \mathbb{Z}_{8380417}[x]/(x^{256} + 1)$
- ✧ Security levels: SHA-256, AES-192, AES-256

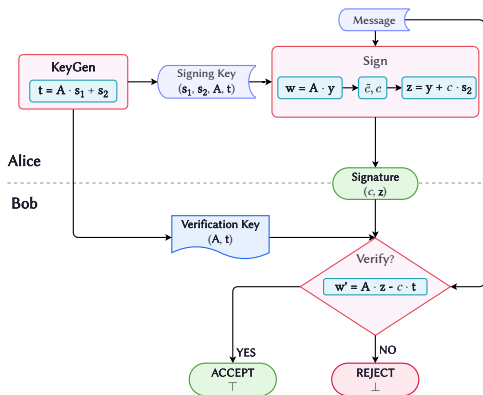
Common Challenge

Both require efficient **polynomial multiplication** in $\mathcal{R}_q$ - this is the computational bottleneck!

ML-KEM AND ML-DSA OVERVIEW



ML-KEM



ML-DSA

THE COMPUTATIONAL BOTTLENECK

✧ Polynomial multiplication in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$:

- ▷ Schoolbook method: $\mathcal{O}(n^2)$ operations
- ▷ For $n = 256$: $256^2 = 65,536$ multiplications per polynomial multiplication
- ▷ ML-KEM/ML-DSA operations require **hundreds** of polynomial multiplications

✧ **Solution:** Number Theoretic Transform (NTT)

- ▷ Reduces complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$
- ▷ Enables efficient hardware implementations

THE COMPUTATIONAL BOTTLENECK

- ✧ Polynomial multiplication in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$:
 - ▷ Schoolbook method: $\mathcal{O}(n^2)$ operations
 - ▷ For $n = 256$: $256^2 = 65,536$ multiplications per polynomial multiplication
 - ▷ ML-KEM/ML-DSA operations require **hundreds** of polynomial multiplications
- ✧ **Solution:** Number Theoretic Transform (NTT)
 - ▷ Reduces complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$
 - ▷ Enables efficient hardware implementations

NTT: THE KEY TO EFFICIENCY

MULTIPLICATION IN THE RING $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ a and b are polynomials in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ A special case of the convolution, when the result is $a \cdot b \pmod{x^n + 1}$:

$$c_j = \sum_{i=0}^j a_i \cdot b_{j-i} - \sum_{i=j+1}^{n-1} a_i \cdot b_{n+j-i}, \quad j = \{0, 1, 2, \dots, n-1\}$$

✧ So, for example for $n = 4$ and $a = [a_0, a_1, a_2, a_3]$ and $b = [b_0, b_1, b_2, b_3]$:

$$c_0 = a_0 \cdot b_0 - a_1 \cdot b_3 - a_2 \cdot b_2 - a_3 \cdot b_1$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 - a_2 \cdot b_3 - a_3 \cdot b_2$$

$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 - a_3 \cdot b_3$$

$$c_3 = a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0$$

✧ The only thing left is to reduce the coefficients modulo q .

✧ But it's still $\mathcal{O}(n^2)$.

MULTIPLICATION IN THE RING $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ a and b are polynomials in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ A special case of the convolution, when the result is $a \cdot b \pmod{x^n + 1}$:

$$c_j = \sum_{i=0}^j a_i \cdot b_{j-i} - \sum_{i=j+1}^{n-1} a_i \cdot b_{n+j-i}, \quad j = \{0, 1, 2, \dots, n-1\}$$

✧ So, for example for $n = 4$ and $a = [a_0, a_1, a_2, a_3]$ and $b = [b_0, b_1, b_2, b_3]$:

$$c_0 = a_0 \cdot b_0 - a_1 \cdot b_3 - a_2 \cdot b_2 - a_3 \cdot b_1$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 - a_2 \cdot b_3 - a_3 \cdot b_2$$

$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 - a_3 \cdot b_3$$

$$c_3 = a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0$$

✧ The only thing left is to reduce the coefficients modulo q .

✧ But it's still $\mathcal{O}(n^2)$.

MULTIPLICATION IN THE RING $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ a and b are polynomials in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ A special case of the convolution, when the result is $a \cdot b \pmod{x^n + 1}$:

$$c_j = \sum_{i=0}^j a_i \cdot b_{j-i} - \sum_{i=j+1}^{n-1} a_i \cdot b_{n+j-i}, \quad j = \{0, 1, 2, \dots, n-1\}$$

✧ So, for example for $n = 4$ and $a = [a_0, a_1, a_2, a_3]$ and $b = [b_0, b_1, b_2, b_3]$:

$$c_0 = a_0 \cdot b_0 - a_1 \cdot b_3 - a_2 \cdot b_2 - a_3 \cdot b_1$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 - a_2 \cdot b_3 - a_3 \cdot b_2$$

$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 - a_3 \cdot b_3$$

$$c_3 = a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0$$

✧ The only thing left is to reduce the coefficients modulo q .

✧ But it's still $\mathcal{O}(n^2)$.

MULTIPLICATION IN THE RING $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ a and b are polynomials in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

✧ A special case of the convolution, when the result is $a \cdot b \pmod{x^n + 1}$:

$$c_j = \sum_{i=0}^j a_i \cdot b_{j-i} - \sum_{i=j+1}^{n-1} a_i \cdot b_{n+j-i}, \quad j = \{0, 1, 2, \dots, n-1\}$$

✧ So, for example for $n = 4$ and $a = [a_0, a_1, a_2, a_3]$ and $b = [b_0, b_1, b_2, b_3]$:

$$c_0 = a_0 \cdot b_0 - a_1 \cdot b_3 - a_2 \cdot b_2 - a_3 \cdot b_1$$

$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0 - a_2 \cdot b_3 - a_3 \cdot b_2$$

$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0 - a_3 \cdot b_3$$

$$c_3 = a_0 \cdot b_3 + a_1 \cdot b_2 + a_2 \cdot b_1 + a_3 \cdot b_0$$

✧ The only thing left is to reduce the coefficients modulo q .

✧ But it's still $\mathcal{O}(n^2)$.

WHAT IS NTT?

- ✧ **Number Theoretic Transform (NTT)**: Generalization of FFT for finite fields/rings
- ✧ Transforms polynomials to a domain where multiplication is **pointwise**
- ✧ **Negacyclic NTT** for rings $\mathbb{Z}_q[x]/(x^n + 1)$:
 - ▷ Uses primitive $2n$ -th root of unity ζ modulo q
 - ▷ $\zeta^{2n} \equiv 1 \pmod{q}$ and $\zeta^n \equiv -1 \pmod{q}$

Polynomial Multiplication with NTT

1. **Transform**: $\hat{a} = \text{NTT}(a)$, $\hat{b} = \text{NTT}(b)$
2. **Pointwise multiply**: $\hat{c} = \hat{a} \circ \hat{b}$ (only n multiplications!)
3. **Inverse transform**: $c = \text{iNTT}(\hat{c})$
 $\Rightarrow c = a \cdot b \pmod{x^n + 1}$

WHAT IS NTT?

- ✧ **Number Theoretic Transform (NTT)**: Generalization of FFT for finite fields/rings
- ✧ Transforms polynomials to a domain where multiplication is **pointwise**
- ✧ **Negacyclic NTT** for rings $\mathbb{Z}_q[x]/(x^n + 1)$:
 - ▷ Uses primitive $2n$ -th root of unity ζ modulo q
 - ▷ $\zeta^{2n} \equiv 1 \pmod{q}$ and $\zeta^n \equiv -1 \pmod{q}$

Polynomial Multiplication with NTT

1. **Transform**: $\hat{a} = \text{NTT}(a)$, $\hat{b} = \text{NTT}(b)$
2. **Pointwise multiply**: $\hat{c} = \hat{a} \circ \hat{b}$ (only n multiplications!)
3. **Inverse transform**: $c = \text{iNTT}(\hat{c})$
 $\Rightarrow c = a \cdot b \pmod{x^n + 1}$

WHAT IS NTT?

- ✧ **Number Theoretic Transform (NTT)**: Generalization of FFT for finite fields/rings
- ✧ Transforms polynomials to a domain where multiplication is **pointwise**
- ✧ **Negacyclic NTT** for rings $\mathbb{Z}_q[x]/(x^n + 1)$:
 - ▷ Uses primitive $2n$ -th root of unity ζ modulo q
 - ▷ $\zeta^{2n} \equiv 1 \pmod{q}$ and $\zeta^n \equiv -1 \pmod{q}$

Polynomial Multiplication with NTT

1. **Transform**: $\hat{a} = \text{NTT}(a)$, $\hat{b} = \text{NTT}(b)$
2. **Pointwise multiply**: $\hat{c} = \hat{a} \circ \hat{b}$ (only n multiplications!)
3. **Inverse transform**: $c = \text{iNTT}(\hat{c})$
 $\Rightarrow c = a \cdot b \pmod{x^n + 1}$

NTT DEFINITION

✧ $\hat{a} = \text{NTT}(a) = (\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1})$:

NTT Equation

$$\hat{a}_j = \sum_{i=0}^{n-1} \zeta^{2ij+i} \cdot a_i \pmod{q}.$$

✧ $a = \text{iNTT}(\hat{a}) = (a_0, a_1, \dots, a_{n-1})$:

iNTT Equation

$$a_i = n^{-1} \sum_{j=0}^{n-1} \zeta^{-2ij-i} \cdot \hat{a}_j \pmod{q}.$$

✧ Time complexity: still $\mathcal{O}(n^2)$

NTT DEFINITION

✧ $\hat{a} = \text{NTT}(a) = (\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1})$:

NTT Equation

$$\hat{a}_j = \sum_{i=0}^{n-1} \zeta^{2ij+i} \cdot a_i \pmod{q}.$$

✧ $a = \text{iNTT}(\hat{a}) = (a_0, a_1, \dots, a_{n-1})$:

iNTT Equation

$$a_i = n^{-1} \sum_{j=0}^{n-1} \zeta^{-2ij-i} \cdot \hat{a}_j \pmod{q}.$$

✧ Time complexity: still $\mathcal{O}(n^2)$

NTT DEFINITION

✧ $\hat{a} = \text{NTT}(a) = (\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1})$:

NTT Equation

$$\hat{a}_j = \sum_{i=0}^{n-1} \zeta^{2ij+i} \cdot a_i \pmod{q}.$$

✧ $a = \text{iNTT}(\hat{a}) = (a_0, a_1, \dots, a_{n-1})$:

iNTT Equation

$$a_i = n^{-1} \sum_{j=0}^{n-1} \zeta^{-2ij-i} \cdot \hat{a}_j \pmod{q}.$$

✧ Time complexity: still $\mathcal{O}(n^2)$

COMPUTING NTT FASTER

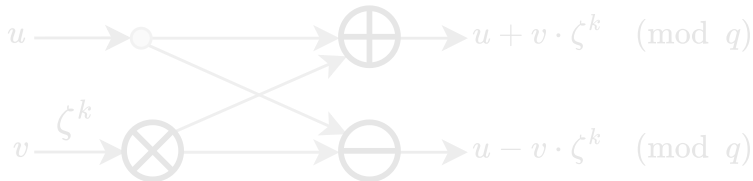
✧ Based on the Fast Fourier Transform (FFT) algorithm, we can compute NTT faster:

▷ **Cooley-Tukey (CT)** algorithm for NTT.

CT Butterfly Pattern

$$u + v \cdot \zeta^k \pmod{q}$$

$$u - v \cdot \zeta^k \pmod{q}$$



COMPUTING NTT FASTER

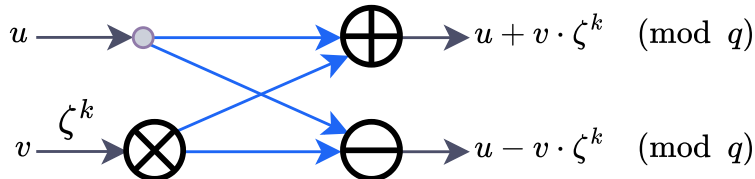
✧ Based on the Fast Fourier Transform (FFT) algorithm, we can compute NTT faster:

▷ **Cooley-Tukey (CT)** algorithm for NTT.

CT Butterfly Pattern

$$u + v \cdot \zeta^k \pmod{q}$$

$$u - v \cdot \zeta^k \pmod{q}$$



COMPUTING iNTT FASTER

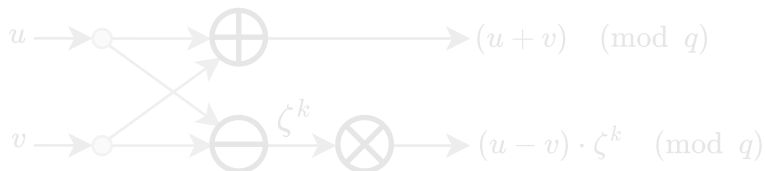
✧ Based on the Fast Fourier Transform (FFT) algorithm, we can compute NTT faster:

▷ **Gentleman-Sande (GS)** algorithm for iNTT.

GS Butterfly Pattern

$$u + v \pmod{q}$$

$$u - v \cdot \zeta^k \pmod{q}$$



COMPUTING iNTT FASTER

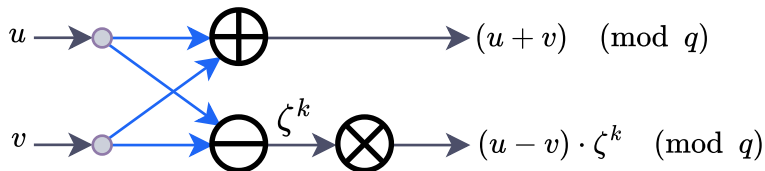
✧ Based on the Fast Fourier Transform (FFT) algorithm, we can compute NTT faster:

▷ **Gentleman-Sande (GS)** algorithm for iNTT.

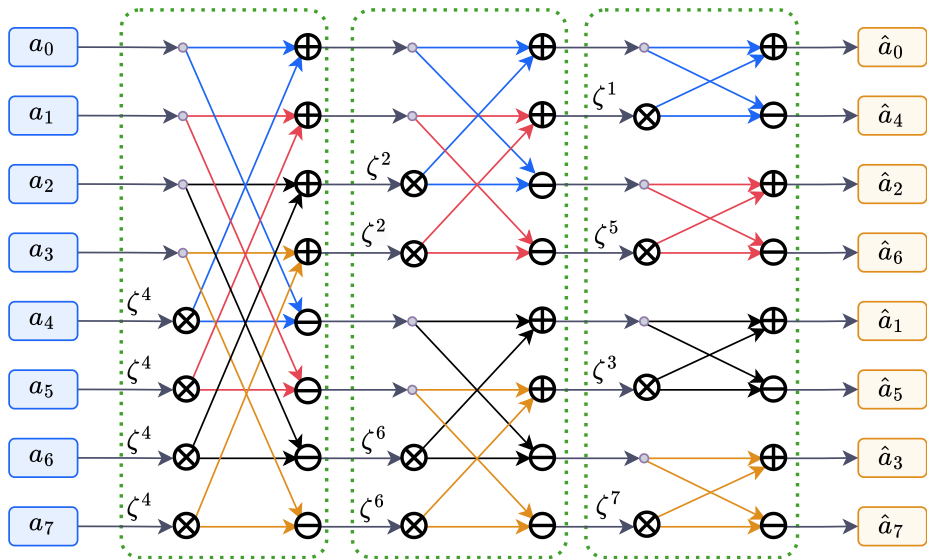
GS Butterfly Pattern

$$u + v \pmod{q}$$

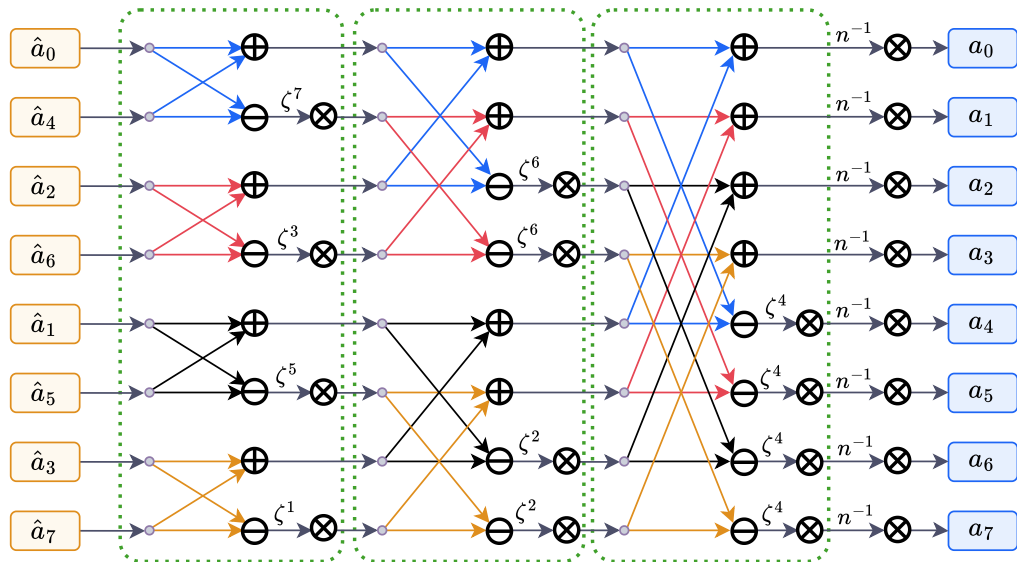
$$u - v \cdot \zeta^k \pmod{q}$$



EXAMPLE: BUTTERFLY DIAGRAM OF 8-POINT NTT



EXAMPLE: BUTTERFLY DIAGRAM OF 8-POINT INTT



✧ **FFT-like Algorithm:** $\mathcal{O}(n \log n)$

- ▷ Exploits periodicity and symmetry of roots of unity
- ▷ Recursive divide-and-conquer approach
- ▷ For $n = 256$: $\sim 2,048$ operations vs 65,536 (32x speedup!)

Total Polynomial Multiplication Cost

- ✧ **NTT:** $\mathcal{O}(n \log n)$
- ✧ **Pointwise Multiplication (PWM):** $\mathcal{O}(n)$
- ✧ **Inverse NTT (iNTT):** $\mathcal{O}(n \log n)$
- ✧ **Total:** $\mathcal{O}(n \log n)$ vs $\mathcal{O}(n^2)$

✧ **FFT-like Algorithm:** $\mathcal{O}(n \log n)$

- ▷ Exploits periodicity and symmetry of roots of unity
- ▷ Recursive divide-and-conquer approach
- ▷ For $n = 256$: $\sim 2,048$ operations vs 65,536 (32x speedup!)

Total Polynomial Multiplication Cost

- ✧ **NTT:** $\mathcal{O}(n \log n)$
- ✧ **Pointwise Multiplication (PWM):** $\mathcal{O}(n)$
- ✧ **Inverse NTT (iNTT):** $\mathcal{O}(n \log n)$
- ✧ **Total:** $\mathcal{O}(n \log n)$ vs $\mathcal{O}(n^2)$

POLYNOMIAL MULTIPLICATION IN NTT DOMAIN

✧ The goal is to compute $c = a \cdot b \pmod{\mathbb{Z}_q[x]/(x^{256} + 1)}$.

✧ Compute the **NTT** of a and b :

$$\hat{a} = (\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{255})$$

$$\hat{b} = (\hat{b}_0, \hat{b}_1, \dots, \hat{b}_{255})$$

✧ Compute the **pointwise multiplication (PWM)** in the NTT domain:

$$\hat{c}_i = \hat{a}_i \cdot \hat{b}_i \pmod{q}$$

and we get:

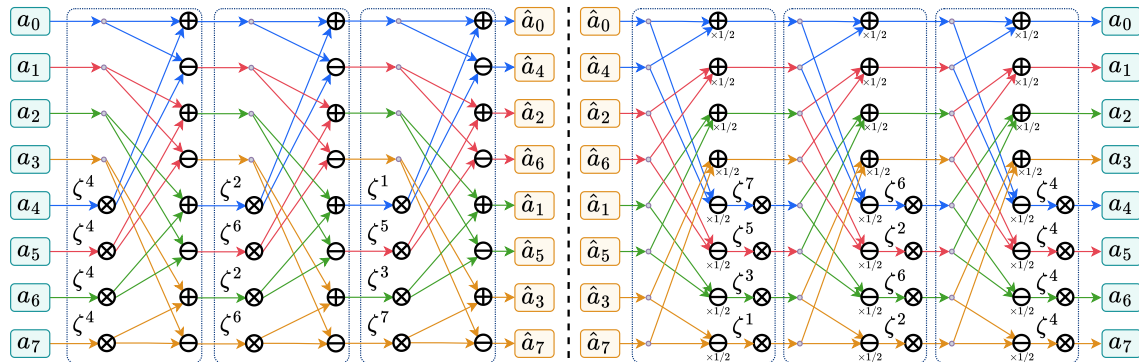
$$\hat{c} = (\hat{c}_0, \hat{c}_1, \dots, \hat{c}_{255})$$

✧ Compute the **inverse NTT** of $\hat{c}$:

$$c = \text{iNTT}(\hat{c})$$

CONSTANT GEOMETRY (CG) NTT

- ✧ Unlike the mentioned NTT algorithms, in the CG NTT, the spacing between the pairs are always the same.



ML-KEM AND ML-DSA: DIFFERENT MODULI, SAME POLYNOMIAL STRUCTURE

ML-KEM AND ML-DSA: DIFFERENCES AND SIMILARITIES

✧ **ML-KEM** Ring: $\mathcal{R}_{3329} = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$

✧ **ML-DSA** Ring: $\mathcal{R}_{8380417} = \mathbb{Z}_{8380417}[x]/(x^{256} + 1)$

Similarities

- ✧ Both are module-lattice-based cryptography schemes
- ✧ Both use the same polynomial structure: $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^{256} + 1)$
- ✧ Both use hash functions (SHAKE and SHA3), TRNG, and polynomial-based operations:
 - ▷ NTT
 - ▷ iNTT
 - ▷ PWM
 - ▷ ADD
 - ▷ SUB

ML-KEM AND ML-DSA: DIFFERENCES AND SIMILARITIES

✧ **ML-KEM** Ring: $\mathcal{R}_{3329} = \mathbb{Z}_{3329}[x]/(x^{256} + 1)$

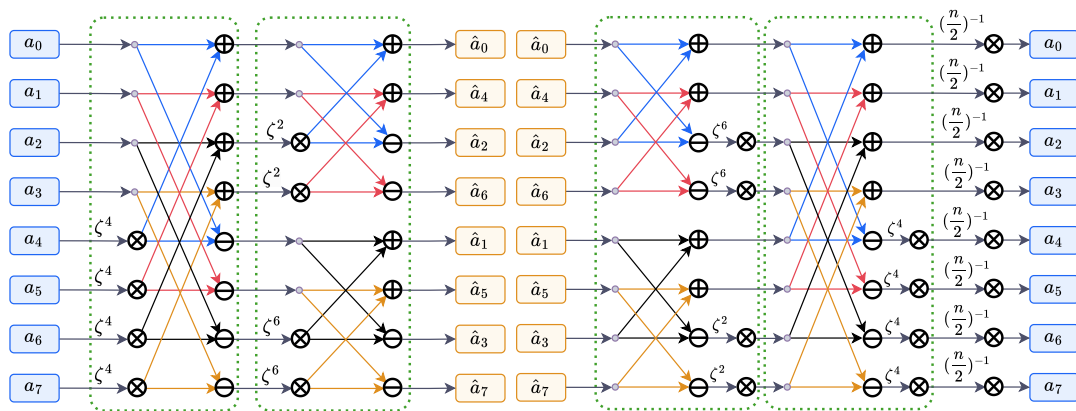
✧ **ML-DSA** Ring: $\mathcal{R}_{8380417} = \mathbb{Z}_{8380417}[x]/(x^{256} + 1)$

Differences

- ✧ Each scheme has its own dedicated hardware blocks.
- ✧ Different moduli: $q = 3329$ vs $q = 8380417$:
 - ▷ The bus width will be different: 12 bits for ML-KEM and 23 bits for ML-DSA
 - ▷ $2n$ -th root of unity exists for ML-DSA, but for ML-KEM only n -th root of unity exists:
 - NTT in ML-KEM converts a polynomial of degree 255 to 128 polynomials of degree 1 (instead of 256 polynomials of degree 0 for ML-DSA) → **incomplete NTT**
 - Stages for ML-KEM are one less than for ML-DSA (7 instead of 8)
 - PWM is different: 5 multiplications are required to compute 2 elements of the product polynomial in the NTT domain

NTT FOR ML-KEM

- ✧ Last stage of the NTT is skipped for ML-KEM.
- ✧ And the first stage of the iNTT is skipped for ML-KEM.



✧ The goal is to compute $c = a \cdot b \pmod{\mathbb{Z}_{3329}[x]/(x^{256} + 1)}$.

✧ Compute the **incomplete NTT** of a and b (in 7 stages):

$$\hat{a} = (\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{255})$$

$$\hat{b} = (\hat{b}_0, \hat{b}_1, \dots, \hat{b}_{255})$$

✧ Compute the **pointwise multiplication (PWM)** in the NTT domain:

$$\hat{c}_{2i} = \hat{a}_{2i} \cdot \hat{b}_{2i} + \hat{a}_{2i+1} \cdot \hat{b}_{2i+1} \cdot 17^{2\text{BitRev}_7(i)+1} \pmod{q}$$

$$\hat{c}_{2i+1} = \hat{a}_{2i} \cdot \hat{b}_{2i+1} + \hat{a}_{2i+1} \cdot \hat{b}_{2i} \pmod{q}$$

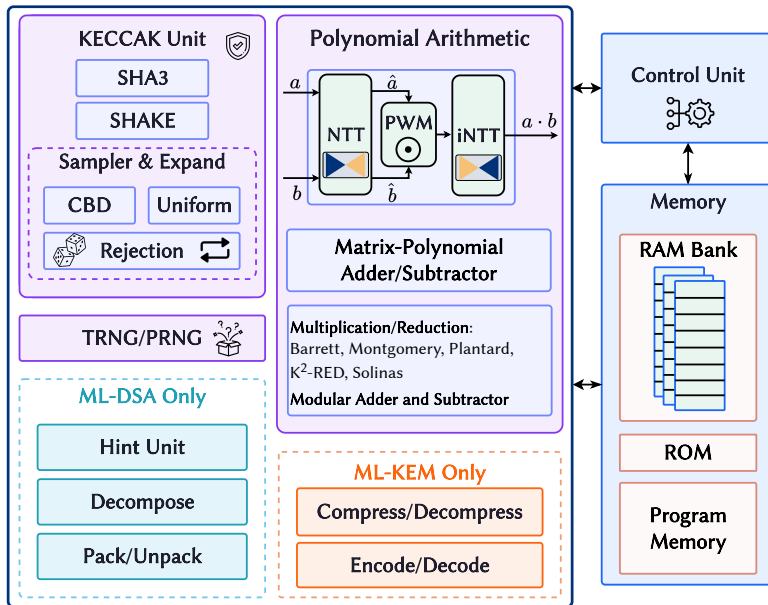
and we get:

$$\hat{c} = (\hat{c}_0, \hat{c}_1, \dots, \hat{c}_{255})$$

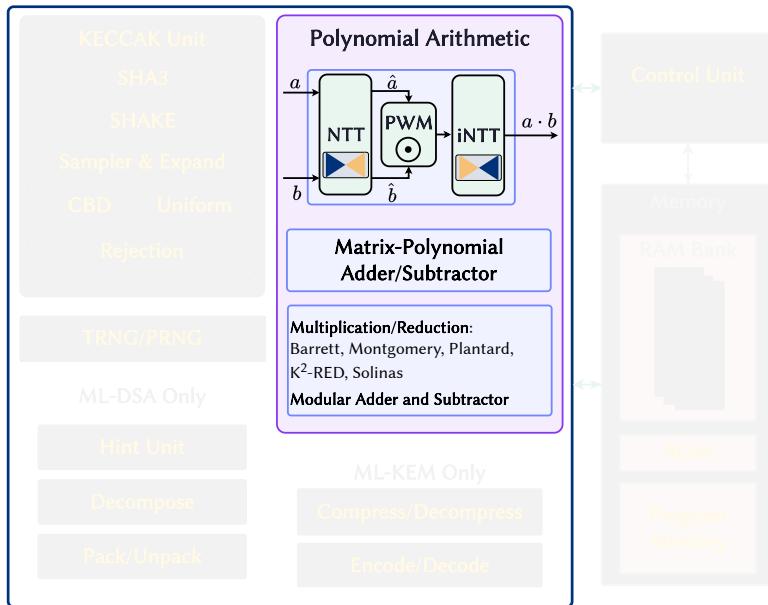
✧ Compute the **inverse incomplete NTT** of $\hat{c}$:

$$c = \text{iNTT}(\hat{c})$$

A UNIFIED PQC ROOT OF TRUST HARDWARE BLOCK



A UNIFIED PQC ROOT OF TRUST HARDWARE BLOCK



HARDWARE IMPLEMENTATION JOURNEY

OUR RESEARCH JOURNEY

✧ Evolution of NTT Hardware Designs:

1. **LightNTT**: Lightweight NTT for ML-DSA
2. **KNightCore**: Lightweight NTT for ML-KEM (with PWM support)
3. **HyperPoly**: Unified high-performance architecture for both

✧ Each design addresses different constraints and requirements

✧ Today's focus: **High-performance unified design** (HyperPoly)

OUR RESEARCH JOURNEY

✧ Evolution of NTT Hardware Designs:

1. **LightNTT**: Lightweight NTT for ML-DSA
2. **KNightCore**: Lightweight NTT for ML-KEM (with PWM support)
3. **HyperPoly**: Unified high-performance architecture for both

✧ Each design addresses different constraints and requirements

✧ Today's focus: **High-performance unified design** (HyperPoly)

OUR RESEARCH JOURNEY

✧ Evolution of NTT Hardware Designs:

1. **LightNTT**: Lightweight NTT for ML-DSA
2. **KNightCore**: Lightweight NTT for ML-KEM (with PWM support)
3. **HyperPoly**: Unified high-performance architecture for both

✧ Each design addresses different constraints and requirements

✧ Today's focus: **High-performance unified design** (HyperPoly)

LIGHTNTT: LIGHTWEIGHT NTT FOR ML-DSA

- ✧ **Goal:** Efficient NTT/iNTT for ML-DSA signatures
- ✧ **Target:** Resource-constrained applications
- ✧ **Key Features:**
 - ▷ Optimized for $\mathcal{R}_{8380417}$
 - ▷ Shared NTT/iNTT datapath
 - ▷ Ping-pong RAM architecture for efficient data flow
 - ▷ Constant Geometry NTT and iNTT
 - ▷ **Smallest possible footprint**

Minimal Resource Use: ▷ Smallest possible footprint.	Constant-Geometry Dataflow: ▷ For simplified control and better security.
Deeply Pipelined Architecture: ▷ With a single butterfly unit to maximize throughput.	Optimized Modular Arithmetic: ▷ Using an efficient, hardware friendly, Barrett multiplier.

MODULAR ADDITION/SUBTRACTION

- ✧ Modular addition and subtraction are quite straightforward.
- ✧ The only tricky part is multiplying by $2^{-1} \bmod q$ in iNTT after each stage.

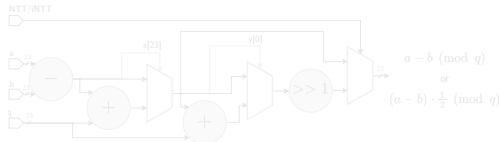
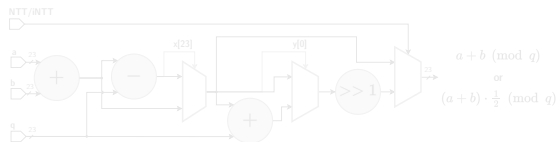
Algorithm 1: Multiply by $2^{-1} \pmod{q}$

Input: $a \in \mathbb{Z}_q$

Output: $out = 2^{-1} \cdot a \pmod{q}$

1 $out \leftarrow a[0] ? a \gg 1 : (a + q) \gg 1$

- ▷ If a is even, just divide by 2.
- ▷ If a is odd, add q and then divide by 2.



MODULAR ADDITION/SUBTRACTION

- ✧ Modular addition and subtraction are quite straightforward.
- ✧ The only tricky part is multiplying by $2^{-1} \bmod q$ in iNTT after each stage.

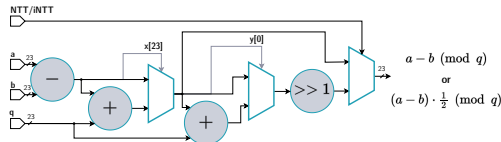
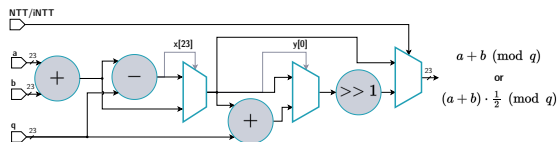
Algorithm 1: Multiply by $2^{-1} \bmod q$

Input: $a \in \mathbb{Z}_q$

Output: $out = 2^{-1} \cdot a \bmod q$

1 $out \leftarrow a[0] ? a \gg 1 : (a + q) \gg 1$

- ▷ If a is even, just divide by 2.
- ▷ If a is odd, add q and then divide by 2.



MODULAR MULTIPLICATION

$$\diamond \mu = \left\lfloor \frac{2^{2t}}{q} \right\rfloor = 2^{23} + 2^{13} + 2^3 - 1.$$

$$\diamond q = 2^{23} - 2^{13} + 1.$$

Algorithm 2: Hardware-friendly Optimized Barrett Modular Multiplication

Input: $a, b \in \mathbb{Z}_q, t = \lceil \log_2 q \rceil = 23$

Output: $out = a \cdot b \pmod{q}$

```
1  $product \leftarrow a \cdot b$ 
2  $x_1 \leftarrow product \gg (t - 2)$ 
3  $x_2 \leftarrow (x_1 \ll 23) + (x_1 \ll 13) + (x_1 \ll 3) - x_1$ 
4  $s \leftarrow x_2 \gg (t + 2)$ 
5  $out \leftarrow product - ((s \ll 23) - (s \ll 13) + s) - q$ 
6 if  $out < 0$  then
7    $out \leftarrow out + q$ 
```

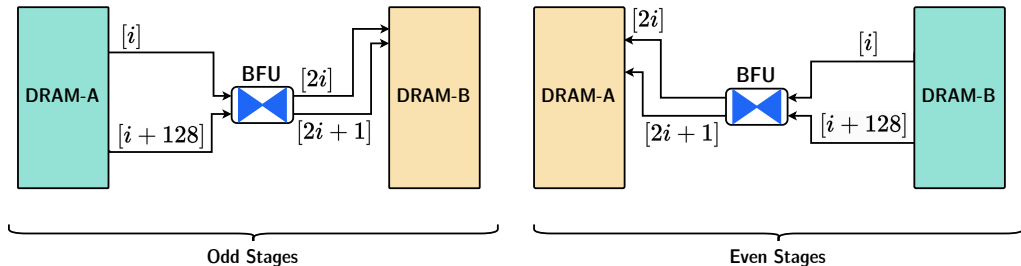
LIGHTNTT: MEMORY ARCHITECTURE

❖ Ping-Pong RAM:

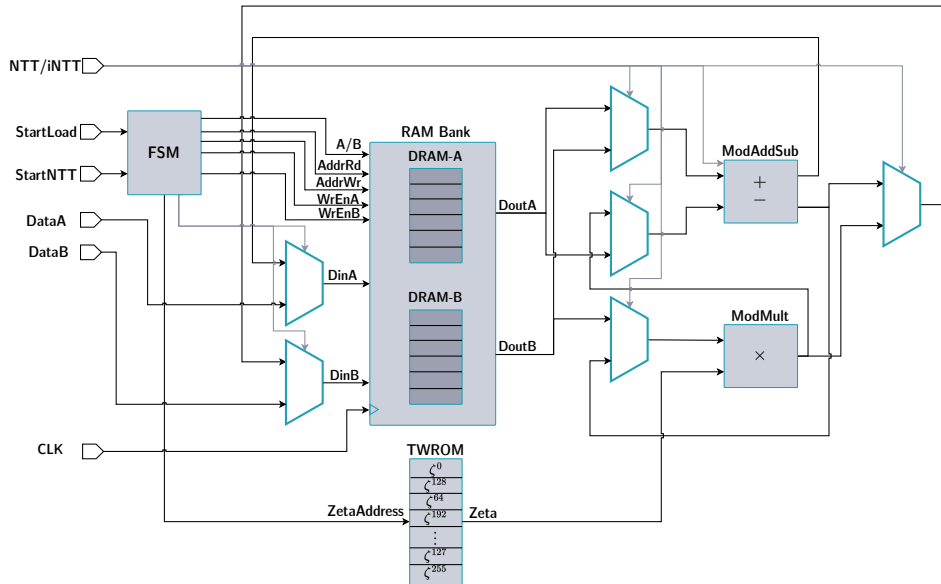
- ▷ Dual-port memory for parallel access
- ▷ Enables pipelined processing

❖ Shared Datapath:

- ▷ NTT and iNTT share same hardware
- ▷ Reduces area overhead



LIGHTNTT: HARDWARE ARCHITECTURE



KNIGHTCORE: NTT FOR ML-KEM

✧ **Goal:** Efficient NTT/iNTT/**PWM** for ML-KEM

✧ **Challenge:**

- ▷ Adapting the CG NTT to the ML-KEM (the final order is different from the in-place NTT)
- ▷ Handling the 5 multiplications required to compute 2 elements of the product polynomial in the NTT domain
- ▷ Still smallest possible footprint is desired, so we need a way to handle the additional twiddle factors for the PWM ($\zeta^{2\text{BitRev}_7(i)+1}$ for $i = 0, 1, \dots, 127$)

Solution

We have to modify the hardware as little as possible, so we can still use the same memory architecture and the same arithmetic unit.

KNIGHTCORE: NTT FOR ML-KEM

✧ **Goal:** Efficient NTT/iNTT/**PWM** for ML-KEM

✧ **Challenge:**

- ▷ Adapting the CG NTT to the ML-KEM (the final order is different from the in-place NTT)
- ▷ Handling the 5 multiplications required to compute 2 elements of the product polynomial in the NTT domain
- ▷ Still smallest possible footprint is desired, so we need a way to handle the additional twiddle factors for the PWM ($\zeta^{2\text{BitRev}_7(i)+1}$ for $i = 0, 1, \dots, 127$)

Solution

We have to modify the hardware as little as possible, so we can still use the same memory architecture and the same arithmetic unit.

Adapting the CG NTT to the ML-KEM

- ✧ Store the result of the final order of the NTT in a different memory locations to match the correct order.
- ✧ Read the input of the first stage of the iNTT from the different memory locations.
- ✧ So, just add some additional logic to the control unit.

The capability of PWM

- ✧ Barrett reduction can compute $a \cdot b \pmod{q}$ without any scaling factor (like in Montgomery multiplication)
- ✧ So, we can use the same arithmetic unit for the PWM as for the other operations, just add some additional logic to the control unit.

Adapting the CG NTT to the ML-KEM

- ✧ Store the result of the final order of the NTT in a different memory locations to match the correct order.
- ✧ Read the input of the first stage of the iNTT from the different memory locations.
- ✧ So, just add some additional logic to the control unit.

The capability of PWM

- ✧ Barrett reduction can compute $a \cdot b \pmod{q}$ without any scaling factor (like in Montgomery multiplication)
- ✧ So, we can use the same arithmetic unit for the PWM as for the other operations, just add some additional logic to the control unit.

Additional twiddle factors for the PWM

We discovered a mathematical property that allows us to generate the required PWM twiddle factors from the existing NTT twiddle factors stored in ROM:

$$\begin{aligned}\zeta^{2\text{brv}_l(2i)+1} &\equiv \zeta^{\text{brv}_l(n/4+i)} \pmod{q}, \\ \zeta^{2\text{brv}_l(2i+1)+1} &\equiv -\zeta^{\text{brv}_l(n/4+i)} \pmod{q}.\end{aligned}$$

This allows us to reuse the NTT's **ROM**, and also the multiplier, adder, and subtractor to perform the entire PWM operation with **zero additional hardware**.

✧ **Result:** We can have a core for NTT/iNTT/PWM without any additional hardware, just some additional logic to the control unit.

Additional twiddle factors for the PWM

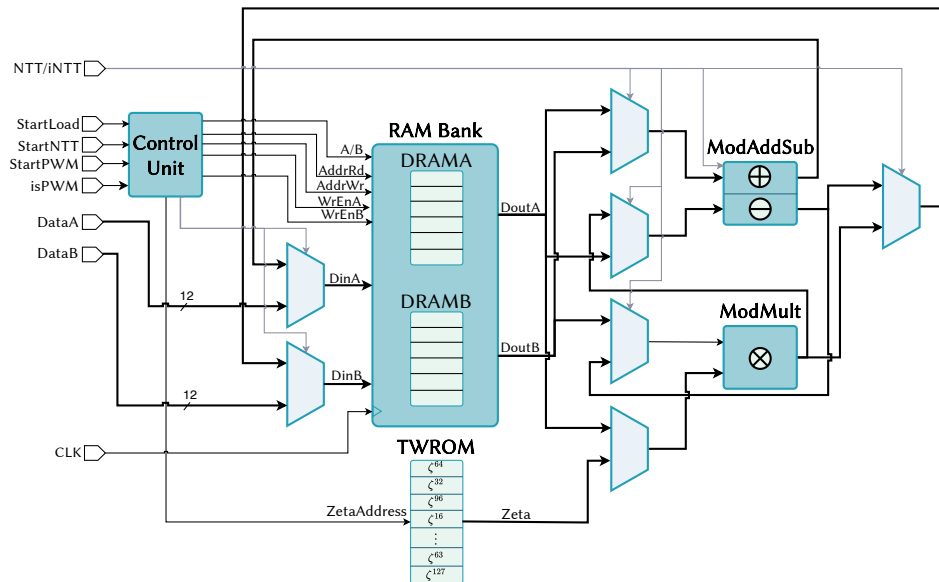
We discovered a mathematical property that allows us to generate the required PWM twiddle factors from the existing NTT twiddle factors stored in ROM:

$$\begin{aligned}\zeta^{2\text{brv}_l(2i)+1} &\equiv \zeta^{\text{brv}_l(n/4+i)} \pmod{q}, \\ \zeta^{2\text{brv}_l(2i+1)+1} &\equiv -\zeta^{\text{brv}_l(n/4+i)} \pmod{q}.\end{aligned}$$

This allows us to reuse the NTT's **ROM**, and also the multiplier, adder, and subtractor to perform the entire PWM operation with **zero additional hardware**.

✧ **Result:** We can have a core for NTT/iNTT/PWM without any additional hardware, just some additional logic to the control unit.

KNIGHTCORE: HARDWARE ARCHITECTURE



HYPERPOLY: UNIFIED HIGH-PERFORMANCE ARCHITECTURE

- ✧ **Goal:** Single architecture supporting both ML-KEM and ML-DSA, performing as fast as possible
- ✧ **Challenge:** Different moduli, different operations, but same polynomial structure
- ✧ **Performance Target:** High-throughput for both schemes

Design Philosophy

"Two Birds, One Stone": One architecture, multiple applications

HYPERPOLY: KEY DESIGN CHALLENGES

Modulus Flexibility

- ✧ ML-KEM: $q = 3329$ (12 bits)
- ✧ ML-DSA: $q = 8380417$ (23 bits)
- ✧ Need a datapath that can handle both moduli

Operation Diversity

- ✧ NTT, iNTT, PWM, ADD, SUB
- ✧ Unified datapath design
- ✧ Supports both schemes

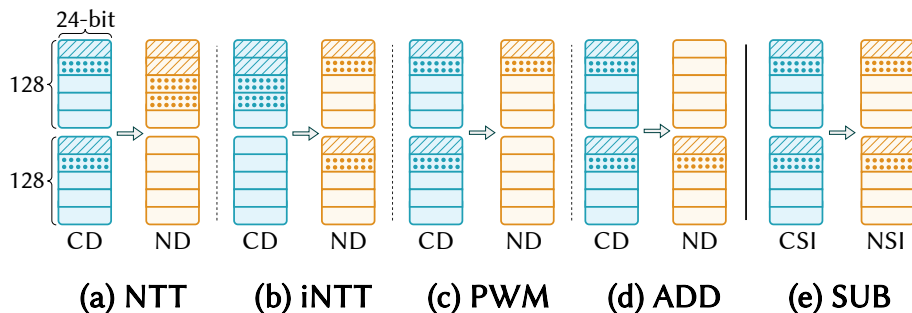
Performance vs Area

- ✧ High performance for both schemes
- ✧ Reasonable area overhead

Memory Management

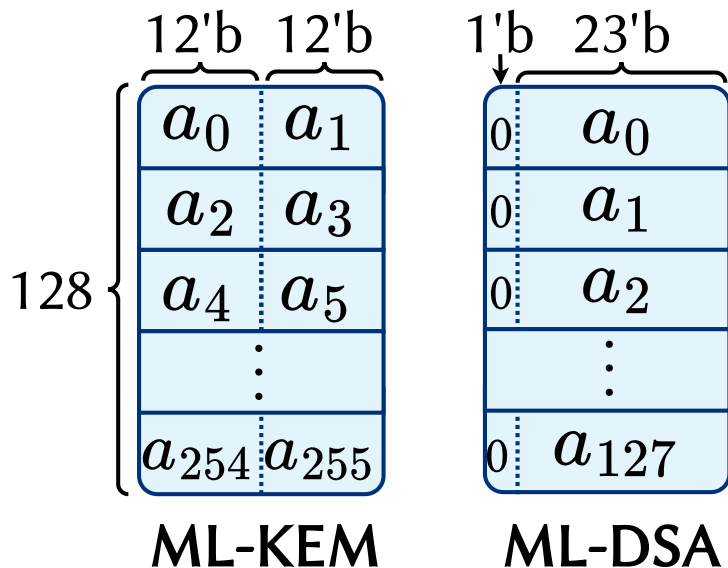
- ✧ Efficient data storage and access
- ✧ Maximum utilization of the memory

HYPERPOLY: DATAFLOW PATTERN



- ✧ Using two Dual-port RAMs of 128x24 bits: 1 ML-DSA or 2 ML-KEM coefficients in each cell
- ✧ All coefficients of one vector of the ML-DSA or two vectors of the ML-KEM will be read at the same time (in parallel) $\rightarrow \mathcal{O}(\log_2 n)$ time complexity
- ✧ Each stage requires one clock cycle for modular addition/subtraction and one clock cycle for modular multiplication $\rightarrow 2$ clock cycles per stage

HYPERPOLY: DATA STORAGE STRATEGY



HYPERPOLY: MODULAR MULTIPLICATION STRATEGY

- ✧ Using the same modular multiplication strategy as in LightNTT and KNightCore for mod q
- ✧ To compute $a \cdot b \pmod{q}$ where a_i and b_i are 24-bit numbers, we can use the following strategy:

- ▷ $a = \{a_l, a_r\}$ and $b = \{b_l, b_r\}$
- ▷ $p_l = a_l \cdot b_l \pmod{q}$ and $p_r = a_r \cdot b_r \pmod{q}$
- ▷ $p_{rl} = p_l + p_r \pmod{q}$
- ▷ $p_c = (a_r + a_l) \cdot (b_r + b_l) \pmod{q}$
- ▷ $p_{crl} = p_c - p_{rl} \pmod{q}$

✧ Now, for **ML-KEM** we have the result: p_l and p_r for the left and right coefficients, respectively.

✧ For **ML-DSA**, using Karatsuba's algorithm:

$$\triangleright p = (p_l \ll 24) + (p_{crl} \ll 12) + p_r \pmod{q}$$

HYPERPOLY: MODULAR MULTIPLICATION STRATEGY

- ✧ Using the same modular multiplication strategy as in LightNTT and KNightCore for $\text{mod } q$
- ✧ To compute $a \cdot b \pmod{q}$ where a_i and b_i are 24-bit numbers, we can use the following strategy:

- ▷ $a = \{a_l, a_r\}$ and $b = \{b_l, b_r\}$
- ▷ $p_l = a_l \cdot b_l \pmod{q}$ and $p_r = a_r \cdot b_r \pmod{q}$
- ▷ $p_{rl} = p_l + p_r \pmod{q}$
- ▷ $p_c = (a_r + a_l) \cdot (b_r + b_l) \pmod{q}$
- ▷ $p_{crl} = p_c - p_{rl} \pmod{q}$

- ✧ Now, for **ML-KEM** we have the result: p_l and p_r for the left and right coefficients, respectively.
- ✧ For **ML-DSA**, using Karatsuba's algorithm:

- ▷ $p = (p_l \ll 24) + (p_{crl} \ll 12) + p_r \pmod{q}$

✧ We need to compute:

$$\begin{aligned}\hat{c}_{2i} &= \hat{a}_{2i} \cdot \hat{b}_{2i} + \hat{a}_{2i+1} \cdot \hat{b}_{2i+1} \cdot \zeta^{2\text{BitRev}_7(i)+1} \pmod{q} \\ \hat{c}_{2i+1} &= \hat{a}_{2i} \cdot \hat{b}_{2i+1} + \hat{a}_{2i+1} \cdot \hat{b}_{2i} \pmod{q} \\ &= (\hat{a}_{2i} + \hat{a}_{2i+1}) \cdot (\hat{b}_{2i} + \hat{b}_{2i+1}) - (\hat{a}_{2i} \cdot \hat{b}_{2i} + \hat{a}_{2i+1} \cdot \hat{b}_{2i+1}) \pmod{q}\end{aligned}$$

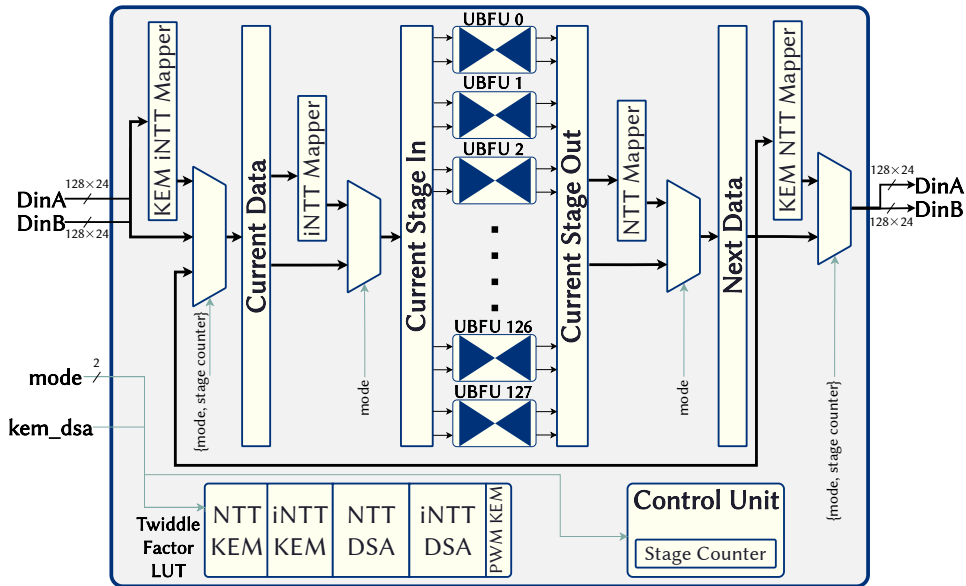
✧ When we want to compute $a \cdot b \pmod{q}$, $\hat{c}_{2i+1}$ has already been computed as p_{crl} .

✧ For $\hat{c}_{2i}$, we already computed $\hat{a}_{2i+1} \cdot \hat{b}_{2i+1}$

✧ So in one cycle, we multiply this by $\zeta^{2\text{BitRev}_7(i)+1}$

✧ And in the next cycle, we add it to $\hat{a}_{2i} \cdot \hat{b}_{2i}$ to get $\hat{c}_{2i}$.

HYPERPOLY: ARCHITECTURE OVERVIEW



CHALLENGES AND CONCLUSIONS

KEY CHALLENGES IN UNIFIED DESIGN

Modulus Heterogeneity

- ✧ ML-KEM: 12-bit modulus (3329)
- ✧ ML-DSA: 23-bit modulus (8380417)
- ✧ Need flexible datapath without excessive area overhead

NTT Incompleteness

- ✧ ML-KEM uses incomplete NTT (7 stages)
- ✧ ML-DSA uses complete NTT (8 stages)
- ✧ Different PWM requirements

Memory Efficiency

- ✧ Store 1 ML-DSA or 2 ML-KEM coefficients per cell
- ✧ Maximize memory utilization
- ✧ Parallel access patterns for high performance

Performance vs Area

- ✧ High throughput for both schemes
- ✧ Reasonable area overhead
- ✧ Efficient resource sharing

KEY CHALLENGES IN UNIFIED DESIGN

Modulus Heterogeneity

- ✧ ML-KEM: 12-bit modulus (3329)
- ✧ ML-DSA: 23-bit modulus (8380417)
- ✧ Need flexible datapath without excessive area overhead

Memory Efficiency

- ✧ Store 1 ML-DSA or 2 ML-KEM coefficients per cell
- ✧ Maximize memory utilization
- ✧ Parallel access patterns for high performance

NTT Incompleteness

- ✧ ML-KEM uses incomplete NTT (7 stages)
- ✧ ML-DSA uses complete NTT (8 stages)
- ✧ Different PWM requirements

Performance vs Area

- ✧ High throughput for both schemes
- ✧ Reasonable area overhead
- ✧ Efficient resource sharing

KEY CHALLENGES IN UNIFIED DESIGN

Modulus Heterogeneity

- ✧ ML-KEM: 12-bit modulus (3329)
- ✧ ML-DSA: 23-bit modulus (8380417)
- ✧ Need flexible datapath without excessive area overhead

NTT Incompleteness

- ✧ ML-KEM uses incomplete NTT (7 stages)
- ✧ ML-DSA uses complete NTT (8 stages)
- ✧ Different PWM requirements

Memory Efficiency

- ✧ Store 1 ML-DSA or 2 ML-KEM coefficients per cell
- ✧ Maximize memory utilization
- ✧ Parallel access patterns for high performance

Performance vs Area

- ✧ High throughput for both schemes
- ✧ Reasonable area overhead
- ✧ Efficient resource sharing

KEY CHALLENGES IN UNIFIED DESIGN

Modulus Heterogeneity

- ✧ ML-KEM: 12-bit modulus (3329)
- ✧ ML-DSA: 23-bit modulus (8380417)
- ✧ Need flexible datapath without excessive area overhead

NTT Incompleteness

- ✧ ML-KEM uses incomplete NTT (7 stages)
- ✧ ML-DSA uses complete NTT (8 stages)
- ✧ Different PWM requirements

Memory Efficiency

- ✧ Store 1 ML-DSA or 2 ML-KEM coefficients per cell
- ✧ Maximize memory utilization
- ✧ Parallel access patterns for high performance

Performance vs Area

- ✧ High throughput for both schemes
- ✧ Reasonable area overhead
- ✧ Efficient resource sharing

FUTURE WORK DIRECTIONS

Side-Channel Resistance

- ✧ Required for small implementation
- ✧ Masking/shuffling countermeasures
- ✧ Power analysis resistance

Performance Optimization

- ✧ Further pipelining opportunities
- ✧ Multi-core architectures
- ✧ ASIC implementation and benchmarking

Integration and Standards

- ✧ Full ML-KEM and ML-DSA implementations
- ✧ Hardware/software co-design
- ✧ Compliance with FIPS 203/204

CONCLUSIONS

1. **NTT is the key** to efficient PQC hardware implementations
2. **Unified designs** can efficiently support multiple PQC schemes
3. **Mathematical insights** enable hardware optimizations with zero overhead
4. **Memory architecture** is crucial for high-performance polynomial operations
5. **Post-quantum transition** requires efficient and flexible hardware solutions

Thank You!

Questions?