Fully Homomorphic Encryption (FHE) - Part II

RNS and SIMD Batching

Chuanqi Zhang

The Compute Bottleneck: Giant Numbers

To guarantee security and support enough noise capacity, FHE requires a very large modulo Q.

The Compute Bottleneck: Giant Numbers

To guarantee security and support enough noise capacity, FHE requires a very large modulo Q.

- **The Scale:** In practice, Q might be a 800-bit integer!

The Compute Bottleneck: Giant Numbers

To guarantee security and support enough noise capacity, FHE requires a very large modulo Q.

- **The Scale:** In practice, Q might be a 800-bit integer!
- **The Hardware Problem:** Your CPU is designed for **64-bit** arithmetic.

The Compute Bottleneck: Giant Numbers

To guarantee security and support enough noise capacity, FHE requires a very large modulo Q.

- **The Scale:** In practice, Q might be a 800-bit integer!
- **The Hardware Problem:** Your CPU is designed for **64-bit** arithmetic.
- **The Consequence:** Adding or multiplying 800-bit numbers is slow because the CPU has to split them into smaller pieces and process them step by step.

The Compute Bottleneck: Giant Numbers

To guarantee security and support enough noise capacity, FHE requires a very large modulo Q.

- **The Scale:** In practice, Q might be a 800-bit integer!
- **The Hardware Problem:** Your CPU is designed for **64-bit** arithmetic.
- **The Consequence:** Adding or multiplying 800-bit numbers is slow because the CPU has to split them into smaller pieces and process them step by step.

How can we compute modulo an 800-bit Q using only 64-bit CPU instructions?

The Residue Number System (RNS)

The solution comes from ancient mathematics: *Chinese Remainder Theorem (CRT)*.

The Residue Number System (RNS)

The solution comes from ancient mathematics: *Chinese Remainder Theorem (CRT)*.

- Instead of using one giant modulus Q, we choose a set of small, coprime moduli: $q_1, q_2, \ldots, q_k$.

The Residue Number System (RNS)

The solution comes from ancient mathematics: *Chinese Remainder Theorem (CRT)*.

- Instead of using one giant modulus Q, we choose a set of small, coprime moduli: $q_1, q_2, \dots, q_k$.
- We ensure that $q_1 \times q_2 \times \cdots \times q_k = Q$, and every $q_i < 2^{64}$.

The Residue Number System (RNS)

The solution comes from ancient mathematics: *Chinese Remainder Theorem (CRT)*.

- Instead of using one giant modulus Q, we choose a set of small, coprime moduli: $q_1, q_2, \ldots, q_k$.
- We ensure that $q_1 \times q_2 \times \cdots \times q_k = Q$, and every $q_i < 2^{64}$.
- **The RNS Representation:** A giant number X is broken down into a tuple of small numbers:

The Residue Number System (RNS)

The solution comes from ancient mathematics: *Chinese Remainder Theorem (CRT)*.

- Instead of using one giant modulus Q, we choose a set of small, coprime moduli: $q_1, q_2, \ldots, q_k$.
- We ensure that $q_1 \times q_2 \times \cdots \times q_k = Q$, and every $q_i < 2^{64}$.
- **The RNS Representation:** A giant number X is broken down into a tuple of small numbers:

$$X \implies (x_1, \quad x_2, \quad \ldots, \quad x_k)$$

where $x_i = X \bmod q_i$.

The Residue Number System (RNS)

The solution comes from ancient mathematics: *Chinese Remainder Theorem (CRT)*.

- Instead of using one giant modulus Q, we choose a set of small, coprime moduli: $q_1, q_2, \ldots, q_k$.
- We ensure that $q_1 \times q_2 \times \cdots \times q_k = Q$, and every $q_i < 2^{64}$.
- **The RNS Representation:** A giant number X is broken down into a tuple of small numbers:

$$X \implies (x_1, \ \ x_2, \ \ \ldots, \ \ x_k)$$

  where $x_i = X \bmod q_i$.
- **The CRT Guarantee:** Math operations can be preserved! We can add or multiply giant numbers by computing their small components over small primes.

The Residue Number System (RNS)

The solution comes from ancient mathematics: *Chinese Remainder Theorem (CRT)*.

- Instead of using one giant modulus Q, we choose a set of small, coprime moduli: $q_1, q_2, \ldots, q_k$.
- We ensure that $q_1 \times q_2 \times \cdots \times q_k = Q$, and every $q_i < 2^{64}$.
- **The RNS Representation:** A giant number X is broken down into a tuple of small numbers:

  $$X \implies (x_1,\ \ x_2,\ \ \ldots,\ \ x_k)$$

  where $x_i = X \bmod q_i$.
- **The CRT Guarantee:** Math operations can be preserved! We can add or multiply giant numbers by computing their small components over small primes. For example,

  $$X \times Y \implies (x_1 \cdot y_1 \bmod q_1,\ \ \ldots,\ \ x_k \cdot y_k \bmod q_k)$$

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5,\ \ 12 \bmod 7) = (2, 5)$

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5, \;\; 12 \bmod 7) = (2, 5)$
- $y \implies \mathbf{y} = (14 \bmod 5, \;\; 14 \bmod 7) = (4, 0)$

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5, \;\; 12 \bmod 7) = (2, 5)$
- $y \implies \mathbf{y} = (14 \bmod 5, \;\; 14 \bmod 7) = (4, 0)$

Step 2: Multiply in RNS (component-wise)

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5, \;\; 12 \bmod 7) = (2, 5)$
- $y \implies \mathbf{y} = (14 \bmod 5, \;\; 14 \bmod 7) = (4, 0)$

Step 2: Multiply in RNS (component-wise)

- $\mathbf{x} \cdot \mathbf{y} \implies (2 \times 4 \bmod 5, \;\; 5 \times 0 \bmod 7) = \color{red}{(3, 0)}$

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5, \ \ 12 \bmod 7) = (2, 5)$
- $y \implies \mathbf{y} = (14 \bmod 5, \ \ 14 \bmod 7) = (4, 0)$

Step 2: Multiply in RNS (component-wise)

- $\mathbf{x} \cdot \mathbf{y} \implies (2 \times 4 \bmod 5, \ \ 5 \times 0 \bmod 7) = (3, 0)$

Verification in the Big Modulo Q

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5, \;\; 12 \bmod 7) = (2, 5)$
- $y \implies \mathbf{y} = (14 \bmod 5, \;\; 14 \bmod 7) = (4, 0)$

Step 2: Multiply in RNS (component-wise)

- $\mathbf{x} \cdot \mathbf{y} \implies (2 \times 4 \bmod 5, \;\; 5 \times 0 \bmod 7) = (3, 0)$

Verification in the Big Modulo Q

- $x \times y = 12 \times 14 = 168 \bmod 35 = 28$.

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5,\ \ 12 \bmod 7) = (2, 5)$
- $y \implies \mathbf{y} = (14 \bmod 5,\ \ 14 \bmod 7) = (4, 0)$

Step 2: Multiply in RNS (component-wise)

- $\mathbf{x} \cdot \mathbf{y} \implies (2 \times 4 \bmod 5,\ \ 5 \times 0 \bmod 7) = (3, 0)$

Verification in the Big Modulo Q

- $x \times y = 12 \times 14 = 168 \bmod 35 = 28.$
- Does 28 match our RNS result? $28 \bmod 5 = 3$, and $28 \bmod 7 = 0$.

RNS: A Concrete Numerical Example

Let's simulate a toy "Giant Number" system. We take target modulo as $Q = 35$.

- We split Q into two small coprime factors: $q_1 = 5$ and $q_2 = 7$.
- Let's take two numbers: $x = 12$ and $y = 14$.

Step 1: Convert to RNS

- $x \implies \mathbf{x} = (12 \bmod 5,\ \ 12 \bmod 7) = (2, 5)$
- $y \implies \mathbf{y} = (14 \bmod 5,\ \ 14 \bmod 7) = (4, 0)$

Step 2: Multiply in RNS (component-wise)

- $\mathbf{x} \cdot \mathbf{y} \implies (2 \times 4 \bmod 5,\ \ 5 \times 0 \bmod 7) = (3, 0)$

Verification in the Big Modulo Q

- $x \times y = 12 \times 14 = 168 \bmod 35 = 28.$
- Does 28 match our RNS result? $28 \bmod 5 = 3$, and $28 \bmod 7 = 0$.
  It's a perfect match!

Why RNS is a Practical Game Changer

By breaking Q into $(q_1, q_2, \ldots, q_k)$, we achieve three architectural advantages:

Why RNS is a Practical Game Changer

By breaking Q into $(q_1, q_2, \ldots, q_k)$, we achieve three architectural advantages:

1. **Small-number arithmetic:** Every operation fits perfectly inside standard 64-bit CPU registers. (E.g., no "BigInt" software overhead.)

Why RNS is a Practical Game Changer

By breaking Q into $(q_1, q_2, \ldots, q_k)$, we achieve three architectural advantages:

1. **Small-number arithmetic:** Every operation fits perfectly inside standard 64-bit CPU registers. (E.g., no "BigInt" software overhead.)
2. **Massive parallelism:** The operations for q_1 have zero dependency on q_2. We can compute all k branches simultaneously across multiple CPU cores!

Why RNS is a Practical Game Changer

By breaking Q into $(q_1, q_2, \ldots, q_k)$, we achieve three architectural advantages:

1. **Small-number arithmetic:** Every operation fits perfectly inside standard 64-bit CPU registers. (E.g., no "BigInt" software overhead.)
2. **Massive parallelism:** The operations for q_1 have zero dependency on q_2. We can compute all k branches simultaneously across multiple CPU cores!
3. **Unlocking fast polynomial multiplication:**

Why RNS is a Practical Game Changer

By breaking Q into $(q_1, q_2, \ldots, q_k)$, we achieve three architectural advantages:

1. **Small-number arithmetic:** Every operation fits perfectly inside standard 64-bit CPU registers. (E.g., no "BigInt" software overhead.)
2. **Massive parallelism:** The operations for q_1 have zero dependency on q_2. We can compute all k branches simultaneously across multiple CPU cores!
3. **Unlocking fast polynomial multiplication:**
   - If each polynomial has N coefficients, naive multiplication takes $O(N^2)$ operations.

Why RNS is a Practical Game Changer

By breaking Q into $(q_1, q_2, \ldots, q_k)$, we achieve three architectural advantages:

1. **Small-number arithmetic:** Every operation fits perfectly inside standard 64-bit CPU registers. (E.g., no "BigInt" software overhead.)
2. **Massive parallelism:** The operations for q_1 have zero dependency on q_2. We can compute all k branches simultaneously across multiple CPU cores!
3. **Unlocking fast polynomial multiplication:**
    - If each polynomial has N coefficients, naive multiplication takes $O(N^2)$ operations.
    - The *Number Theoretic Transform* (NTT) speeds it up to $O(N \log N)$.

Why RNS is a Practical Game Changer

By breaking Q into $(q_1, q_2, \ldots, q_k)$, we achieve three architectural advantages:

1. **Small-number arithmetic:** Every operation fits perfectly inside standard 64-bit CPU registers. (E.g., no "BigInt" software overhead.)
2. **Massive parallelism:** The operations for q_1 have zero dependency on q_2. We can compute all k branches simultaneously across multiple CPU cores!
3. **Unlocking fast polynomial multiplication:**
    - If each polynomial has N coefficients, naive multiplication takes $O(N^2)$ operations.
    - The *Number Theoretic Transform* (NTT) speeds it up to $O(N \log N)$.
    - NTT *requires* small prime moduli. RNS natively provides exactly what NTT needs!

The Data Bottleneck: Wasted Space

RNS solved the compute speed, but what about memory?

The Data Bottleneck: Wasted Space

RNS solved the compute speed, but what about memory?

- In modern FHE (like BGV, BFV, CKKS), a ciphertext is a huge polynomial with N coefficients (e.g., $N = 8192$).

The Data Bottleneck: Wasted Space

RNS solved the compute speed, but what about memory?

- In modern FHE (like BGV, BFV, CKKS), a ciphertext is a huge polynomial with N coefficients (e.g., $N = 8192$).
- **The Inefficiency:** If we use a ciphertext of size $N = 8192$ to encrypt just one single integer, we are wasting 99.99% of the mathematical capacity.

The Data Bottleneck: Wasted Space

RNS solved the compute speed, but what about memory?

- In modern FHE (like BGV, BFV, CKKS), a ciphertext is a huge polynomial with N coefficients (e.g., $N = 8192$).
- **The Inefficiency:** If we use a ciphertext of size $N = 8192$ to encrypt just one single integer, we are wasting 99.99% of the mathematical capacity.
- **Data Expansion:** A 4-byte integer expands into a 1-Megabyte ciphertext!

The Data Bottleneck: Wasted Space

RNS solved the compute speed, but what about memory?

- In modern FHE (like BGV, BFV, CKKS), a ciphertext is a huge polynomial with N coefficients (e.g., $N = 8192$).
- **The Inefficiency:** If we use a ciphertext of size $N = 8192$ to encrypt just one single integer, we are wasting 99.99% of the mathematical capacity.
- **Data Expansion:** A 4-byte integer expands into a 1-Megabyte ciphertext!

Can we pack multiple messages into a single polynomial?

SIMD: Single Instruction, Multiple Data

The answer is **Batching** or **SIMD**.

SIMD: Single Instruction, Multiple Data

The answer is **Batching** or **SIMD**.

- Instead of encrypting one number, we encrypt a vector of N numbers simultaneously into a single ciphertext.

SIMD: Single Instruction, Multiple Data

The answer is **Batching** or **SIMD**.

- Instead of encrypting one number, we encrypt a vector of N numbers simultaneously into a single ciphertext.
- **The Magic:** When you homomorphically add or multiply two SIMD ciphertexts, the operation happens to all N elements independently at the same time!

SIMD: Single Instruction, Multiple Data

The answer is **Batching** or **SIMD**.

- Instead of encrypting one number, we encrypt a vector of N numbers simultaneously into a single ciphertext.
- **The Magic:** When you homomorphically add or multiply two SIMD ciphertexts, the operation happens to all N elements independently at the same time!

SIMD Example

$$Enc([1,2,3]) \; + \; Enc([10,20,30]) \implies Enc([{\color{red}11,22,33}])$$

SIMD: Single Instruction, Multiple Data

The answer is **Batching** or **SIMD**.

- Instead of encrypting one number, we encrypt a vector of N numbers simultaneously into a single ciphertext.
- **The Magic:** When you homomorphically add or multiply two SIMD ciphertexts, the operation happens to all N elements independently at the same time!

SIMD Example

$$Enc([1,2,3]) \; + \; Enc([10,20,30]) \implies Enc([11,22,33])$$

$$Enc([2,3,4]) \; \times \; Enc([5,5,5]) \implies Enc([10,15,20])$$

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?

Answer: Polynomial ring!

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?
Answer: Polynomial ring!

- Let our plaintext be a polynomial $P(X)$ in the ring $\mathbb{Z}_q[X]/(X^N - 1)$.

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?

Answer: Polynomial ring!

- Let our plaintext be a polynomial $P(X)$ in the ring $\mathbb{Z}_q[X]/(X^N - 1)$.
- The polynomial $X^N - 1$ has N distinct roots: $r_1, r_2, \ldots, r_N$.

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?
Answer: Polynomial ring!

- Let our plaintext be a polynomial $P(X)$ in the ring $\mathbb{Z}_q[X]/(X^N - 1)$.
- The polynomial $X^N - 1$ has N distinct roots: $r_1, r_2, \ldots, r_N$.
- **The Slots:** We encode our messages by defining them as the evaluations of the polynomial at these roots:

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?
Answer: Polynomial ring!

- Let our plaintext be a polynomial $P(X)$ in the ring $\mathbb{Z}_q[X]/(X^N - 1)$.
- The polynomial $X^N - 1$ has N distinct roots: $r_1, r_2, \ldots, r_N$.
- **The Slots:** We encode our messages by defining them as the evaluations of the polynomial at these roots:

$$P(r_1) = m_1, \quad P(r_2) = m_2, \quad \ldots, \quad P(r_N) = m_N$$

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?
Answer: Polynomial ring!

- Let our plaintext be a polynomial $P(X)$ in the ring $\mathbb{Z}_q[X]/(X^N - 1)$.
- The polynomial $X^N - 1$ has N distinct roots: $r_1, r_2, \ldots, r_N$.
- **The Slots:** We encode our messages by defining them as the evaluations of the polynomial at these roots:

$$P(r_1) = m_1, \quad P(r_2) = m_2, \quad \ldots, \quad P(r_N) = m_N$$

- **The Magic:** If we multiply two polynomials $R(X) = P(X)Q(X)$, the evaluation at any root r_i is naturally:

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?

Answer: Polynomial ring!

- Let our plaintext be a polynomial $P(X)$ in the ring $\mathbb{Z}_q[X]/(X^N - 1)$.
- The polynomial $X^N - 1$ has N distinct roots: $r_1, r_2, \ldots, r_N$.
- **The Slots:** We encode our messages by defining them as the evaluations of the polynomial at these roots:

$$P(r_1) = m_1, \quad P(r_2) = m_2, \quad \ldots, \quad P(r_N) = m_N$$

- **The Magic:** If we multiply two polynomials $R(X) = P(X)Q(X)$, the evaluation at any root r_i is naturally:

$$R(r_i) = P(r_i) \cdot Q(r_i) = m_i \cdot m_i'$$

The Mathematical Engine: Polynomial CRT

How do we find a mathematical object that supports this batching property?
Answer: Polynomial ring!

- Let our plaintext be a polynomial $P(X)$ in the ring $\mathbb{Z}_q[X]/(X^N - 1)$.
- The polynomial $X^N - 1$ has N distinct roots: $r_1, r_2, \ldots, r_N$.
- **The Slots:** We encode our messages by defining them as the evaluations of the polynomial at these roots:

$$P(r_1) = m_1, \quad P(r_2) = m_2, \quad \ldots, \quad P(r_N) = m_N$$

- **The Magic:** If we multiply two polynomials $R(X) = P(X)Q(X)$, the evaluation at any root r_i is naturally:

$$R(r_i) = P(r_i) \cdot Q(r_i) = m_i \cdot m_i'$$

Conclusion: By Using SIMD

1. Storage: Encrypting ONE polynomial $\Rightarrow$ encrypting N messages simultaneously!
2. Compute: ONE polynomial multiplication $\Rightarrow$ N point-wise multiplications!

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

- **The Ring Rule:** We use a polynomial ring bounded by $X^2 - 1$. This means whenever we see X^2, it wraps around and becomes 1 (i.e., $X^2 = 1$).

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

- **The Ring Rule:** We use a polynomial ring bounded by $X^2 - 1$. This means whenever we see X^2, it wraps around and becomes 1 (i.e., $X^2 = 1$).
- **The Slots:** Because $X^2 - 1 = 0$ has roots at $X = 1$ and $X = -1$, these two roots are our "Message Slots".

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

- **The Ring Rule:** We use a polynomial ring bounded by $X^2 - 1$. This means whenever we see X^2, it wraps around and becomes 1 (i.e., $X^2 = 1$).
- **The Slots:** Because $X^2 - 1 = 0$ has roots at $X = 1$ and $X = -1$, these two roots are our "Message Slots".

Encoding into Polynomials (Plaintexts):

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

- **The Ring Rule:** We use a polynomial ring bounded by $X^2 - 1$. This means whenever we see X^2, it wraps around and becomes 1 (i.e., $X^2 = 1$).
- **The Slots:** Because $X^2 - 1 = 0$ has roots at $X = 1$ and $X = -1$, these two roots are our "Message Slots".

Encoding into Polynomials (Plaintexts):

- For $[4, 2]$, we need a polynomial $P(X)$ where $P(1) = 4$ and $P(-1) = 2$.

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

- **The Ring Rule:** We use a polynomial ring bounded by $X^2 - 1$. This means whenever we see X^2, it wraps around and becomes 1 (i.e., $X^2 = 1$).
- **The Slots:** Because $X^2 - 1 = 0$ has roots at $X = 1$ and $X = -1$, these two roots are our "Message Slots".

Encoding into Polynomials (Plaintexts):

- For $[4, 2]$, we need a polynomial $P(X)$ where $P(1) = 4$ and $P(-1) = 2$. This is $P(X) = X + 3$.

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

- **The Ring Rule:** We use a polynomial ring bounded by $X^2 - 1$. This means whenever we see X^2, it wraps around and becomes 1 (i.e., $X^2 = 1$).
- **The Slots:** Because $X^2 - 1 = 0$ has roots at $X = 1$ and $X = -1$, these two roots are our "Message Slots".

Encoding into Polynomials (Plaintexts):

- For $[4, 2]$, we need a polynomial $P(X)$ where $P(1) = 4$ and $P(-1) = 2$. This is $P(X) = X + 3$.
- For $[5, 1]$, we need a polynomial $Q(X)$ where $Q(1) = 5$ and $Q(-1) = 1$.

SIMD in Action: Step 1 - Encoding the Arrays

Let's try a toy example and see the full magic! Say we want to multiply two arrays: $[4, 2] \times [5, 1]$.

- **The Ring Rule:** We use a polynomial ring bounded by $X^2 - 1$. This means whenever we see X^2, it wraps around and becomes 1 (i.e., $X^2 = 1$).
- **The Slots:** Because $X^2 - 1 = 0$ has roots at $X = 1$ and $X = -1$, these two roots are our "Message Slots".

Encoding into Polynomials (Plaintexts):

- For $[4, 2]$, we need a polynomial $P(X)$ where $P(1) = 4$ and $P(-1) = 2$. This is $P(X) = X + 3$.
- For $[5, 1]$, we need a polynomial $Q(X)$ where $Q(1) = 5$ and $Q(-1) = 1$. This is $Q(X) = 2X + 3$.

SIMD in Action: Step 2 - The Single Encryption

Here is where the “SIMD” concept truly begins.

SIMD in Action: Step 2 - The Single Encryption

Here is where the "SIMD" concept truly begins.

- We do NOT encrypt the numbers 4, 2, 5, and 1 individually.

SIMD in Action: Step 2 - The Single Encryption

Here is where the “SIMD” concept truly begins.

- We do NOT encrypt the numbers 4, 2, 5, and 1 individually.
- FHE schemes natively take *an entire polynomial* as a single atomic input.

SIMD in Action: Step 2 - The Single Encryption

Here is where the "SIMD" concept truly begins.

- We do NOT encrypt the numbers 4, 2, 5, and 1 individually.
- FHE schemes natively take *an entire polynomial* as a single atomic input.
- We run the encryption algorithm exactly ONCE for each array by:

SIMD in Action: Step 2 - The Single Encryption

Here is where the "SIMD" concept truly begins.

- We do NOT encrypt the numbers 4, 2, 5, and 1 individually.
- FHE schemes natively take *an entire polynomial* as a single atomic input.
- We run the encryption algorithm exactly ONCE for each array by:

$$C_A = Enc(P(X)) \implies Enc(X + 3)$$
$$C_B = Enc(Q(X)) \implies Enc(2X + 3)$$

SIMD in Action: Step 2 - The Single Encryption

Here is where the "SIMD" concept truly begins.

- We do NOT encrypt the numbers 4, 2, 5, and 1 individually.
- FHE schemes natively take *an entire polynomial* as a single atomic input.
- We run the encryption algorithm exactly ONCE for each array by:

$$C_A = Enc(P(X)) \implies Enc(X+3)$$
$$C_B = Enc(Q(X)) \implies Enc(2X+3)$$

The Core Idea

C_A or C_B is just **one** ciphertext matrix, but its underlying mathematical structure has secretly locked in **two** independent messages in its slots!

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

What happens inside the encrypted math?

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

What happens inside the encrypted math?

$$P(X) \cdot Q(X) = (X + 3)(2X + 3)$$

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

What happens inside the encrypted math?

$$
\begin{aligned}
P(X) \cdot Q(X) &= (X + 3)(2X + 3) \\
&= 2X^2 + 3X + 6X + 9
\end{aligned}
$$

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

What happens inside the encrypted math?

$$\begin{aligned} P(X) \cdot Q(X) &= (X+3)(2X+3) \\ &= 2X^2 + 3X + 6X + 9 \\ &= 2X^2 + 9X + 9 \end{aligned}$$

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

What happens inside the encrypted math?

$$
\begin{aligned}
P(X) \cdot Q(X) &= (X + 3)(2X + 3) \\
&= 2X^2 + 3X + 6X + 9 \\
&= 2X^2 + 9X + 9 \\
&= 2\textcolor{red}{(1)} + 9X + 9 \quad \text{(Applying the Ring Rule: } X^2 = 1\text{)}
\end{aligned}
$$

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

What happens inside the encrypted math?

$$\begin{aligned} P(X) \cdot Q(X) &= (X+3)(2X+3) \\ &= 2X^2 + 3X + 6X + 9 \\ &= 2X^2 + 9X + 9 \\ &= 2(1) + 9X + 9 \quad \text{(Applying the Ring Rule: } X^2 = 1) \\ &= 9X + 11 \end{aligned}$$

SIMD in Action: Step 3 - Homomorphic Computation

The Cloud receives C_A and C_B. It does not know the arrays, because it does not know the original polynomials.

- The Cloud performs a single homomorphic multiplication: $C_{mult} = C_A \otimes C_B$.
- By the laws of FHE, this homomorphically multiplies the underlying polynomials: $Enc(P(X) \cdot Q(X))$.

What happens inside the encrypted math?

$$\begin{aligned} P(X) \cdot Q(X) &= (X+3)(2X+3) \\ &= 2X^2 + 3X + 6X + 9 \\ &= 2X^2 + 9X + 9 \\ &= 2(1) + 9X + 9 \quad \text{(Applying the Ring Rule: } X^2 = 1\text{)} \\ &= 9X + 11 \end{aligned}$$

So, the new ciphertext is $C_{mult} = Enc(9X + 11)$.

SIMD in Action: Step 4 - Decoding the Result

The Client downloads C_{mult} and decrypts it to reveal the resulting polynomial:

SIMD in Action: Step 4 - Decoding the Result

The Client downloads C_{mult} and decrypts it to reveal the resulting polynomial:

$$Dec(C_{mult}) = R(X) = 9X + 11$$

SIMD in Action: Step 4 - Decoding the Result

The Client downloads C_{mult} and decrypts it to reveal the resulting polynomial:

$$Dec(C_{mult}) = R(X) = 9X + 11$$

Now, the Client checks the "Slots" of this polynomial:

SIMD in Action: Step 4 - Decoding the Result

The Client downloads C_{mult} and decrypts it to reveal the resulting polynomial:

$$Dec(C_{mult}) = R(X) = 9X + 11$$

Now, the Client checks the "Slots" of this polynomial:

- **Slot 1**: $R(1) = 9(1) + 11 = 20$.

SIMD in Action: Step 4 - Decoding the Result

The Client downloads C_{mult} and decrypts it to reveal the resulting polynomial:

$$Dec(C_{mult}) = R(X) = 9X + 11$$

Now, the Client checks the "Slots" of this polynomial:

- **Slot 1**: $R(1) = 9(1) + 11 = 20$.
- **Slot 2**: $R(-1) = 9(-1) + 11 = -9 + 11 = 2$.

SIMD in Action: Step 4 - Decoding the Result

The Client downloads C_{mult} and decrypts it to reveal the resulting polynomial:

$$Dec(C_{mult}) = R(X) = 9X + 11$$

Now, the Client checks the "Slots" of this polynomial:

- **Slot 1**: $R(1) = 9(1) + 11 = 20$.
- **Slot 2**: $R(-1) = 9(-1) + 11 = -9 + 11 = 2$.

Conclusion: By Using SIMD

This matches the correct result of $[4, 2] \times [5, 1] = [20, 2]$ that we want to compute at the beginning.

SIMD in Action: Step 4 - Decoding the Result

The Client downloads C_{mult} and decrypts it to reveal the resulting polynomial:

$$Dec(C_{mult}) = R(X) = 9X + 11$$

Now, the Client checks the "Slots" of this polynomial:

- **Slot 1**: $R(1) = 9(1) + 11 = 20$.
- **Slot 2**: $R(-1) = 9(-1) + 11 = -9 + 11 = 2$.

Conclusion: By Using SIMD

This matches the correct result of $[4, 2] \times [5, 1] = [20, 2]$ that we want to compute at the beginning.

By encrypting just **one** polynomial and asking the cloud to perform **one** multiplication, we simultaneously processed **multiple** messages and their operations!

Why is this called “SIMD”?

SIMD stands for *Single Instruction, Multiple Data*.

Why is this called "SIMD"?

SIMD stands for *Single Instruction, Multiple Data*.

- **Single Instruction:** The Cloud executed exactly ONE operation ($C_A \otimes C_B$).

Why is this called “SIMD”?

SIMD stands for *Single Instruction, Multiple Data*.

- **Single Instruction:** The Cloud executed exactly ONE operation ($C_A \otimes C_B$).
- **Multiple Data:** That single operation simultaneously multiplied the data hidden in Slot 1 ($4 \times 5 = 20$) and Slot 2 ($2 \times 1 = 2$).

Why is this called "SIMD"?

SIMD stands for *Single Instruction, Multiple Data*.

- **Single Instruction:** The Cloud executed exactly ONE operation ($C_A \otimes C_B$).
- **Multiple Data:** That single operation simultaneously multiplied the data hidden in Slot 1 ($4 \times 5 = 20$) and Slot 2 ($2 \times 1 = 2$).

The Real World Scale

Why is this called "SIMD"?

SIMD stands for *Single Instruction, Multiple Data*.

- **Single Instruction:** The Cloud executed exactly ONE operation ($C_A \otimes C_B$).
- **Multiple Data:** That single operation simultaneously multiplied the data hidden in Slot 1 ($4 \times 5 = 20$) and Slot 2 ($2 \times 1 = 2$).

The Real World Scale

- In our Toy Example, we used $X^2 - 1$, which gave us **2 slots**.

Why is this called "SIMD"?

SIMD stands for *Single Instruction, Multiple Data*.

- **Single Instruction:** The Cloud executed exactly ONE operation ($C_A \otimes C_B$).
- **Multiple Data:** That single operation simultaneously multiplied the data hidden in Slot 1 ($4 \times 5 = 20$) and Slot 2 ($2 \times 1 = 2$).

The Real World Scale

- In our Toy Example, we used $X^2 - 1$, which gave us **2 slots**.
- In OpenFHE, the polynomial ring is often bounded by $X^{8192} + 1$.

Why is this called "SIMD"?

SIMD stands for *Single Instruction, Multiple Data*.

- **Single Instruction:** The Cloud executed exactly ONE operation ($C_A \otimes C_B$).
- **Multiple Data:** That single operation simultaneously multiplied the data hidden in Slot 1 ($4 \times 5 = 20$) and Slot 2 ($2 \times 1 = 2$).

The Real World Scale

- In our Toy Example, we used $X^2 - 1$, which gave us **2 slots**.
- In OpenFHE, the polynomial ring is often bounded by $X^{8192} + 1$.
- This gives us **8,192 slots**!

Why is this called "SIMD"?

SIMD stands for *Single Instruction, Multiple Data*.

- **Single Instruction:** The Cloud executed exactly ONE operation ($C_A \otimes C_B$).
- **Multiple Data:** That single operation simultaneously multiplied the data hidden in Slot 1 ($4 \times 5 = 20$) and Slot 2 ($2 \times 1 = 2$).

The Real World Scale

- In our Toy Example, we used $X^2 - 1$, which gave us **2 slots**.
- In OpenFHE, the polynomial ring is often bounded by $X^{8192} + 1$.
- This gives us **8,192 slots**!

Conclusion: One homomorphic multiplication processes 8,192 data points simultaneously. This is how FHE overcomes the data bottleneck for more applications!

Summary

1. RNS: Solving the Compute Bottleneck

2. SIMD: Solving the Data Bottleneck