Number-Theoretic Transform Architecture for Fully Homomorphic Encryption from Hypercube Topology

Jingwei Hu[1], Yuhong Fang[2] and Wangchen Dai[2]

[1] Nanyang Technological University, Singapore, davidhu@ntu.edu.sg
[2] Zhejiang Lab, China, w.dai@my.cityu.edu.hk

Abstract. This paper introduces a high-performance and scalable hardware architecture designed for the Number-Theoretic Transform (NTT), a fundamental component extensively utilized in lattice-based encryption and fully homomorphic encryption schemes.

The underlying rationale behind this research is to harness the advantages of the hypercube topology. This topology serves to significantly diminish the volume of data exchanges required during each iteration of the NTT, reducing it to a complexity of $\Omega(\log N)$. Concurrently, it enables the parallelization of N processing elements. This reduction in data exchange operations is of paramount importance. It not only facilitates the establishment of interconnections among the N processing elements but also lays the foundation for the development of a high-performance NTT design. This is particularly valuable when dealing with large values of N.

Keywords: Fully Homomorphic Encryption, Number Theoretic Transform, FPGA Implementation

Contents

Introduction

The contributions of this paper include:

- Pioneering Hypercube Topology in NTT Designs: This research introduces the innovative concept of applying hypercube topology to NTT designs. It successfully addresses the challenging issue of managing a substantial volume of data exchange within high-performance NTT designs.

- Prototyping a Hypercube-Based NTT Hardware: The study provides a practical implementation of NTT hardware based on the hypercube topology. Importantly, it allows users the flexibility to configure the degree of parallelization as per their requirements. The paper also offers theoretical estimations of the timing performance, which are subsequently validated through concrete implementation results.

1 FHEW-like Fully Homomorphic Encryption Scheme

TFHE [CGGI20] and FHEW [DM15] are examples of the third generation of fully homomorphic encryption (FHE) schemes. They offer advantages in terms of bootstrapping performance, which is a crucial operation in FHE schemes. The state-of-the-art advancements in TFHE/FHEW indicate that any discrete function f over a small domain $\mathbb{Z}_t$ can be bootstrapped. This unique capability is often referred to as functional or programmable bootstrapping. It allows for performing computations on encrypted data without decrypting it. In this work, we stay focused on one particular function called `NAND` gate which is proposed in the original FHEW paper as:

$$\texttt{NAND}(x, y) = \begin{cases} 0 & \text{if } x == y \\ 1 & \text{otherwise} \end{cases}$$

where $x, y \in \{0, 1\}$.

1.1 LWE and RLWE

TFHE ciphertext is essentially an LWE ciphertext. The LWE ciphertext is defined as follows:

$$(\mathbf{a}, b) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + \frac{q}{t}m + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$$

, where $\mathbf{s}$ is a secret key, $m \in \mathbb{Z}_t$ is the message, e is an error term extracted from a discrete Gaussian distribution with small variance $e \sim \mathcal{N}(0, \sigma^2)$. The parameters $n, q, \mathbf{s}$ characterize an LWE ciphertext. We use the notation $LWE_{\mathbf{s}}^{n,q}(m)$ (or $LWE_{\mathbf{s}}(m), LWE(m), LWE(\frac{q}{t}m)$ when the context is clear) to represent an LWE encryption of the message m.

In the TFHE bootstrapping algorithm, another format of ciphertext, called RLWE ciphertext, is used. An RLWE ciphertext can be viewed as an LWE ciphertext defined over polynomial ring $R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$:

$$(a(X), b(X)) = (a(X), a(X) \cdot z(X) + \frac{Q}{t}m(X) + e(X)) \in R_Q \times R_Q$$

where z is the secret key, m is the message, $e = \sum_i e_i X^i$ is the error polynomial where each coefficient is extracted from a discrete Gaussian distribution with small variance $e_i \sim \mathcal{N}(0, \sigma^2)$. Likewise, we use $RLWE_z^{N,Q}(m)$ (or $RLWE_{\mathbf{s}}(m), RLWE(m), RLWE(\frac{Q}{t}m)$ when the context is clear) to denote an RLWE encryption of the message $m(X)$.

1.2 Key Switch and Mod Switch

Key switching and modulus switching are two important homomorphic operations in TFHE. We skip the algorithmic detail for these two primitives but outline the functionality abstractly. `KeySwitch` takes as input an LWE encryption of message m under secret key $\mathbf{z}$ and outputs another LWE ciphertext encrypting the same message m but under a different secret key $\mathbf{s}$:

$$LWE_{\mathbf{z}}^{N,Q}(m) \xrightarrow{\texttt{KeySwitch}} LWE_{\mathbf{s}}^{n,Q}(m)$$

`KeySwitch` utilizes another publicly known key called key switching key **KSK** to perform this operation and `KeySwitch` introduces extra noise to the input LWE ciphertext. In practice, we may use another form of key switching, called LWE-to-RLWE keyswitch which homomorphically converts an LWE encryption to an RLWE encryption without changing the encrypted message:

$$LWE_{\mathbf{z}}^{N,Q}(m) \xrightarrow{\texttt{LWE-to-RLWE KeySwitch}} RLWE_{z}^{n,Q}(m)$$

`ModSwitch` takes input as an LWE encryption of message m defined over $\mathbb{Z}_Q^N \times \mathbb{Z}_Q$ and outputs another LWE encryption of the same message defined over $\mathbb{Z}_q^N \times \mathbb{Z}_q$ with $Q > q$:

$$LWE_{\mathbf{s}}^{N,Q}(m) \xrightarrow{\texttt{ModSwitch}} LWE_{\mathbf{s}}^{N,q}(m)$$

A unique feature of `ModSwitch` is that it reduces the noise within the LWE ciphertext. TFHE bootstrapping utilizes this feature to homomorphically evaluate a discrete function f while reducing the noise in the ciphertext.

1.3 Sample Extraction

Sample extraction is another classical technique that homomorphically extracts a term in an encrypted polynomial. More precisely, the input for sample extraction is an RLWE ciphertext $RLWE(m(X))$ which encrypts a polynomial $m(X) = \sum_{i=0}^{N-1} m_i X^i$ and sample extraction extracts the i-th term of $m(X)$ as an LWE encryption $LWE(m_i)$ denoted as:

$$LWE_{\mathbf{z}}^{N,Q}(m_i) \leftarrow \texttt{SampleExt}(RLWE_{z}^{N,Q}(m(X), i)$$

Note that the secret key $z = \sum_{i=0}^{N-1} z_i X^i$ is changed accordingly to its vector form $\mathbf{z} = (z_0, \cdots, z_{N-1})$. Sometimes, we abuse the notation above and use $LWE_{\mathbf{z}}^{N,Q}(m_0) \leftarrow \texttt{SampleExt}(RLWE_{z}^{N,Q}(m(X))$ to denote the extraction of the constant term. The computational overhead for `SampleExt` is trivial and the noise does not grow.

1.4 RGSW cryptosystem

In TFHE/FHEW schemes, a new encryption scheme called RGSW [GSW13] is used as the basis for homomorphic multiplication. We use the notation $RGSW_{z}^{N,Q}(m(X))$ to denote an RGSW encryption of a polynomial $m(X) \in R_Q = \mathbb{Z}_Q[X]/(X^N + 1)$ under the secret key z.

Before detailing RGSW, an extended form of RLWE encryption with respect to some basis B_g should be introduced, *i.e.*,

$$RLWE'(m) = (RLWE(m), \cdots, RLWE(B^i m), \cdots, RLWE(B^{d_g - 1} m))$$

where $d_g = \lceil log_{B_g}(Q) \rceil$.

The noise growth for the scalar multiplication $d \cdot RLWE'(m)$ defined in $RLWE'(\cdot)$ is well controlled since:

$$d \cdot RLWE'(m) = \sum_{i=0}^{d_g - 1} d_i \cdot RLWE(B^i m) = RLWE(\sum_i d_i B^i m) = RLWE(d \cdot m)$$

where $d = \sum_i d_i B^i$.

RSGW encryption is further constructed based on $RLWE'(\cdot)$ as:

$$RGSW_z(m) = (RLWE'_z(-z \cdot m), RLWE'_z(m))$$

In other terms, RGSW not only encrypts the message m but also encrypts $z \cdot m$ which encloses the secret key z.

The RGSW homomorphic multiplication between a RLWE ciphertext $RLWE(m_0) \stackrel{def}{=} (a, b)$ and another RSGW ciphertext $RGSW(m_1)$ $\odot : RLWE \times RGSW \to RLWE$ proceeds as follows:

$$\begin{aligned} RLWE(m_0) \odot RGSW(m_1) =& (a, b) \odot (RLWE'(-z \cdot m_1), RLWE'(m_1)) \\ =& a \cdot RLWE'(-z \cdot m_1) + b \cdot RLWE'(m_1) \\ =& RLWE((b - as) \cdot m_1) \\ =& RLWE(\frac{Q}{t} m_0 m_1 + e_0 m_1) \end{aligned}$$

since $e_0 m_1$ is a small component and can be treated as noise and thus we have $RLWE(\frac{Q}{t} m_0 m_1 + e_0 m_1) = RLWE(\frac{Q}{t} m_0 m_1)$. Also note that the inputs are asymmetric where one input is an RLWE encryption while the other is an RGSW encryption. This type of homomorphic multiplication is also known as external product. Each external product takes $4 d_g N \log N$ $\mathbb{F}_Q$ multiplications where $d_g \approx 6$ is a small constant used to describe RSGW encryption if number-theoretic transform (NTT) is applied.

1.5 Blind Rotation

Blind Rotation essentially homomorphically performs an inner product of a public known vector $\mathbf{a} \stackrel{def}{=} (a_0, \cdots, a_{n-1})$ and the private key vector $\mathbf{s} \stackrel{def}{=} (s_0, \cdots, s_{n-1})$ over the exponent of X by extensive use of RGSW external product. In order to control the noise growth during the homomorphic computation, a redundant encryption of the private key $\mathbf{s}$, *i.e.*, $\mathbf{Z}_{i,j,v} = RGSW(X^{v B_r^j \cdot s_i})$ for all i, j, v is used. Algorithm 1 formally describes the primitive `BlindRotate`.

Input: bootstrapping key brK $\mathbf{Z}_{i,j,v} = RGSW(X^{v B_r^j \cdot s_i})$ for all i, j, v, RLWE ciphertext $ct = RLWE(X^b)$ and a public known vector $\mathbf{a}$.
Output: an RLWE encryption of $X^{b - \langle \mathbf{a}, \mathbf{s} \rangle}$.

for $i \leftarrow 0$ **to** $n - 1$ **do**
  Compute $a_i \leftarrow q - a_i$
  for $j \leftarrow 0$ **to** $log_{B_r} q - 1$ **do**
    Compute $a_{i,j} \leftarrow \lfloor a_i / B_r^j \rfloor$
    Compute $ACC \leftarrow ACC \odot \mathbf{Z}_{i,j,a_{i,j}}$
return ACC

Algorithm 1: the primitive blind rotation `BlindRotate`(brK, ct, $\mathbf{a}$)

Proposition 1. *Given an RLWE encryption of a monomial X^b, and a public known vector $\mathbf{a}$ an an pre-computed encryption of the secret keys, i.e., $\mathbf{Z}_{i,j,v} = RGSW(X^{v B_r^j \cdot s_i})$ for all i, j, v. Algorithm 1 correctly computes an RLWE encryption of $X^{b - \langle \mathbf{a}, \mathbf{s} \rangle}$.*

Proof. The algorithm proceeds in a nested loop with loop-i as the outer loop and loop-j as the inner loop. Each iteration of loop-i first flattens a_i to $\{a_{i,j}\}$ for all j and then

computes in the inner loop-j $RLWE(X^{b-\sum_{k=0}^{i} a_k s_k})$ from $RLWE(X^{b-\sum_{k=0}^{i-1} a_k s_k})$ since $-a_i s_i = \sum_j a_{i,j} B_r^j s_i$. By induction, ACC returns $RLWE(X^{b-\sum_{k=0}^{n-1} a_k s_k})$ at the end of the algorithm. □

1.6 Bootstrapping a NAND gate

A full description of homomorphic NAND gate operation is presented in Alg. 2. $acc_{\mathrm{BR},f} = RLWE_z^{N,Q}(rotP \cdot X^{\widetilde{m}})$ where $\widetilde{m} = b_N - \langle \mathbf{a}_N, \mathbf{s} \rangle = \frac{q}{t} m + e$, $ct_Q = LWE_{\mathbf{z}}^{N,Q}(f(m))$, and $ct_q = LWE_{\mathbf{s}}^{n,q}(f(m))$. HDB is relatively fast since it performs the most time-consuming operation `BlindRotate`, whose computational complexity is $\mathcal{O}(d_g N^2 \log N)$, only once. Here, d_g is a small constant used in the TFHE system parameter description. For 80-bit security, one HDB function evaluation takes less than 1 second by a typical software implementation on a desktop PC.

Input: bootstrapping key brK, LWE ciphertexts $ct_i = LWE_{\mathbf{s}}^{q/4}(m_i) = (\mathbf{a}, b)$ for $i \in \{0, 1\}$, LWE-to-LWE key-switching key ksK.

Output: an LWE encryption of $\mathtt{NAND}(m_0, m_1)$, *i.e.,* $ct_q = LWE_{\mathbf{s}}^{q/4}(\mathtt{NAND}(m_0, m_1))$.

1 Compute $ct \leftarrow ct_0 + ct_1$

2 Set $rotP = f(0)\left(\sum_{j=0}^{\frac{N}{4}} X^j + \sum_{j=N-\frac{N}{4}+1}^{N-1} X^j\right) - f(1)\left(\sum_{j=\frac{N}{4}+1}^{\frac{3N}{4}} X^j\right)$ where $f(0) = f(1) = \frac{q}{8}$

3 Set $ct_N \leftarrow \mathtt{ModSwitch}(ct, q, 2N) = (\mathbf{a}_N, b_N)$

4 Compute $acc_f \leftarrow rotP \cdot X^{b_N}$

5 Run $acc_{\mathrm{BR},f} \leftarrow \mathtt{BlindRotate}(brK, acc_f, ct_N)$

6 Run $ct_Q \leftarrow \mathtt{SampleExt}(acc_{\mathrm{BR},f}, 0)$

7 Run $ct_{Q,ksK} \leftarrow \mathtt{KeySwitch}(ct_Q, ksK)$

8 Run $ct_q \leftarrow \mathtt{ModSwitch}(ct_Q, Q, q)$

9 **return** $ct_q + (\mathbf{0}, \frac{q}{8})$

Algorithm 2: Half domain bootstrapping `Bootstrap`(brK, ct, $f(\cdot)$, ksK)

Proposition 2. *Given an LWE encryption of a binary message m_0, and an LWE encryption of another message m_1, Algorithm 2 correctly computes an LWE encryption of $\mathtt{NAND}(m_0, m_1)$.*

Proof. In line 1, the homomorphic addition returns $ct = LWE_{\mathbf{s}}^{q/4}(m_0 + m_1)$. In line 2, the rotation polynomial $rotP(x)$ is configured according to the anti-cyclic function $f(x)$ for $x \in \{0, 1, 2, 3\}$ *s.t.* $f(0) = f(1) = \frac{q}{8}$ and $f(2) = f(3) = -\frac{q}{8}$. In line 3, the primitive `ModSwitch` is invoked to scale the ciphertext modulus q down to $2N$ making $ct_N = LWE_{\mathbf{s}}^{2N/4}(m_0 + m_1)$. Line 4 essentially creates a noiseless and trivial RLWE encryption of $rotP(X) \cdot X^{b_N}$. In line 5, the primitive `BlindRotate` uses the bootstrapping key to compute $acc_{\mathrm{BR},f} = RLWE_z^{N,Q}(rotP \cdot X^{\widetilde{m}})$ where $\widetilde{m} = b_N - \langle \mathbf{a}_N, \mathbf{s} \rangle = \frac{2N}{t}(m_0 + m_1) + e \bmod 2N$. In line 6, the constant term in $rotP \cdot X^{\widetilde{m}}$ is extracted as a new LWE encryption over a relatively larger ciphertext modulus, *i.e.*, $LWE_{\mathbf{z}}^{N,Q}(f(m_0 + m_1))$. Line 7 switches the secret key $\mathbf{z}$ back to the original secret key $\mathbf{s}$ such that $ct_{Q,ksk} = LWE_{\mathbf{s}}^{n,Q}(f(m_0 + m_1))$. Line 8 performs again the primitive `ModSwitch` to switch back to the original ciphertext modulus q such that $ct_q = LWE_{\mathbf{s}}^{n,q}(f(m_0 + m_1))$. In line 9, the final result $LWE_{\mathbf{s}}^{n,q}(f(m_0 + m_1) + q/8)$ is returned. It is easy to verify that $LWE_{\mathbf{s}}^{n,q}(f(m_0 + m_1) + q/8) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$ which encrypts '0' if $m_0 = m_1 = 1$, and $LWE_{\mathbf{s}}^{n,q}(f(m_0 + m_1) + q/8) = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + \frac{q}{4} + e)$ which encrypts '1' otherwise. □

2 Number-theoretic Transform Architecture

This section describes NTT hardware at the bottom level. Firstly, The distinguishing feature is that a series of new twiddle factor LUTs is constructed and used: an independent twiddle factor LUT, denoted as $\{w_i[\cdot]\}_{i=0,\ldots,log_2 N-1}$, is prepared for the i-th round of butterfly computation ($logN$ rounds in total) as described in Alg. 4. Note that differing from the standard FFT which uses twiddle factor ω_N^i for $i \in [N]$, the NTT used in the ring $\mathcal{R}_q$ uses the modified twiddle factor $\omega_N^i \cdot \omega_{2N}^j$ for $i \in [N], j = 2^0, 2^1, \cdots, 2^{logN-1}$.

2.1 Higher level description for NTT with merged twiddle factors

In this subsection, we discuss the NTT algorithm with merged twiddle factors. No pre-processing or post-processing is required in this variant of NTT algorithm. At an abstract level, the structure of this NTT algorithm is identical to that of the classic FFT algorithm.

The formal description of this NTT variant is shown in Alg. 3 which is also refered to as Cooley-Tukey (CT) butterfly or decimation in time (DIT) in the open literature. It is essentially identical to the classic FFT algorithm except the twiddle factor array $w_i[\cdot]$. It has $logN$ iterations (loop-i) at outermost, where each iteration computes one layer of butterfly computations. The $i(i = 0, \cdots, logN - 1)$-th layer of butterfly computation always has $\frac{N}{2}$ butterflies. These butterflies are bundled into 2^i groups (recorded by the variable $NumberofGroups$) and each groups has $\frac{N}{2^i}$ pairs of butterflies (recorded by the variable $PairsInGroup$). The key feature is that at a particular iteration (say the i-th iteration), the butterflies in a particular group (say the k-th group) share the same twiddle factor $w_i[k]$. The variable $Distance$ is used to locate precisely two inputs of a particular pair of butterfly in loop-j, *i.e.*, $a[j]$ and $a[j + Distance]$. The variables $JFirst$ and $JLast$ indicate the starting and the ending position of the array $a[\cdot]$, respectively, used in the k-th group of the i-th iteration.

A visualization of Alg. 3 is depicted in Fig. 1a when $N = 8$. The inputs are $a[0], \cdots, a[7]$ where $a[i]$ represents the i-th coefficient a_i in the polynomial $a(X) = \sum_{i=0}^{N-1} a_i X^i$. The NTT computation has 3 layers of butterflies: In the first layer ($i = 0$ for loop-i in Alg. 3), only one butterfly group (associated with twiddle factor ω_{16}^4) exists; in the second layer, two butterfly groups (associated with twiddle factor ω_{16}^2 and ω_{16}^6) exist; in the third layer, four butterfly groups (associated with twiddle factors ω_{16}^1, ω_{16}^5, ω_{16}^3, and ω_{16}^7, respectively) exist. It is worth noting that the ouputs from the NTT network is in bit-reversed order as $A[0], A[4], A[2], A[6], A[1], A[5], A[3], A[7]$.

Next, we detail how to construct the twiddle factor LUT. Recall the NTT with pre-processing can be written together as a summation of N terms:

$$A_i = \sum_{j=0}^{N-1} a_j \omega_{2N}^j \omega_N^{ij} \bmod q, i \in [0, N-1] \tag{1}$$

Next, by splitting the summation above (Equation 1) into even and odd groups according to the index i of A_i, we obtain

$$\begin{aligned} A_i &= \sum_{j=0}^{\frac{N}{2}-1} a_{2j} \omega_N^{2ij} \omega_{2N}^{2j} + \sum_{j=0}^{\frac{N}{2}-1} a_{2j+1} \omega_N^{i(2j+1)} \omega_{2N}^{2j+1} \bmod q \\ &= \sum_{j=0}^{\frac{N}{2}-1} a_{2j} \omega_{\frac{N}{2}}^{ij} \omega_N^j + \omega_N^i \omega_{2N} \sum_{j=0}^{\frac{N}{2}-1} a_{2j+1} \omega_{\frac{N}{2}}^{ij} \omega_N^j \bmod q \end{aligned} \tag{2}$$

Now express A_is in Equation 2 into the first half A_i and the second half $A_{i+\frac{N}{2}}$ as

Input: polynomial $a(x) \in R_q$ represented in an array $a[\cdot]$, Twiddle factors $\{w_i[\cdot]\}_{i=0,\ldots,log_2 N-1}$
Output: NTT($a(x)$) represented in $a[\cdot]$ (in-place)

$PairsInGroup \leftarrow N/2$
$NumOfGroups \leftarrow 1$
$Distance \leftarrow N/2$
for $i \leftarrow 0$ **to** $log_2 N - 1$ **do**
  for $k \leftarrow 0$ **to** $NumOfGroups - 1$ **do**
    $JFirst \leftarrow 2 \cdot k \cdot PairsInGroup$
    $JLast \leftarrow JFirst + PairsInGroup - 1$
    for $j \leftarrow JFirst$ **to** $JLast$ **do**
      $Temp \leftarrow w_i[k] * a[j + Distance]$
      $a[j + Distance] \leftarrow a[j] - Temp$
      $a[j] \leftarrow a[j] + Temp$
  $PairsInGroup \leftarrow PairsInGroup/2$
  $NumOfGroups \leftarrow NumOfGroups \cdot 2$
  $Distance \leftarrow Distance/2$
return $a[\cdot]$

Algorithm 3: Higher level description of NTT, *a.k.a* $DIT_{NN\rightarrow RN}$

Input: a polynomial ring R_q, and NTT points N
Output: Twiddle factors $\{w_i[\cdot]\}_{i=0,\ldots,log_2 N-1}$ used in Algorithm 3

$FirstPart \leftarrow 0$ where $[0 \cdots 0]_2 == BinRepr(0)$
$SecondPart \leftarrow 2^{N-1}$ where $[1 \cdots 0]_2 == BinRepr(2^{N-1})$
for $i \leftarrow 0$ **to** $log_2 N - 1$ **do**
  for $j \leftarrow 0$ **to** $N - 1$ **do**
    $[j_{log_2 N-1,\cdots,j_0}]_2 \leftarrow BinRepr(j)$
    $Firstpart \leftarrow \sum_{k=0}^{i} j_{log_2 N-i-1+k} \cdot 2^{log_2 N-1-k}$
    $w_i[j] \leftarrow \phi^{Firstpart} \cdot \phi^{SecondPart}$
  $SecondPart \leftarrow SecondPart/2$
return $\{w_i[\cdot]\}_{i=0,\ldots,log_2 N-1}$

Algorithm 4: Construction of Twiddle Factor LUTs

follows:

$$A_i = \sum_{j=0}^{\frac{N}{2}-1} a_{2j}\omega_{\frac{N}{2}}^{ij}\omega_N^j + \omega_N^i\omega_{2N}\sum_{j=0}^{\frac{N}{2}-1} a_{2j+1}\omega_{\frac{N}{2}}^{ij}\omega_N^j \bmod q \text{ for } i \in [0, \frac{N}{2} - 1]$$
$$A_{i+\frac{N}{2}} = \sum_{j=0}^{\frac{N}{2}-1} a_{2j}\omega_{\frac{N}{2}}^{ij}\omega_N^j - \omega_N^i\omega_{2N}\sum_{j=0}^{\frac{N}{2}-1} a_{2j+1}\omega_{\frac{N}{2}}^{ij}\omega_N^j \bmod q \quad (3)$$

Assume $N = 2^n$, let $Y_i^{(n-1)}$ and $Z_i^{(n-1)}$ be solutions to the two half-sized subproblems (NTT of size of $\frac{N}{2} = 2^{n-1}$ for the even terms $\{a_{2j}\}_{j\in[\frac{N}{2}]}$ and the odd terms $\{a_{2j+1}\}_{j\in[\frac{N}{2}]}$)

Table 1: Merged Twiddle Factor used in N-point NTT. The exponent is expressed in binary form.

NTT iteration i	twiddle factor associated with $a[j]$ and $a[j+distance]$ where $j=[j_{n-1}j_{n-2}\cdots j_1j_0]_2$
$i=0$	$\omega_N^{\overbrace{00\cdots00}^{n-1 \text{ bits}}} \cdot \omega_{2N}^{\overbrace{10\cdots00}^{n \text{ bits}}}$
$i=0$	$\omega_N^{j_{n-1}0\cdots00} \cdot \omega_{2N}^{01\cdots00}$
$\vdots$	$\vdots$
$i=n-2$	$\omega_N^{j_2j_3\cdots00} \cdot \omega_{2N}^{00\cdots10}$
$i=n-1$	$\omega_N^{j_1j_2\cdots j_{n-2}j_{n-1}} \cdot \omega_{2N}^{00\cdots01}$

defined by

$$Y_i^{(n-1)} = \sum_{j=0}^{\frac{N}{2}-1} a_{2j}\omega_{\frac{N}{2}}^{ij}\omega_N^j \bmod q \text{ for } i \in [0, \frac{N}{2}-1]$$

$$Z_i^{(n-1)} = \sum_{j=0}^{\frac{N}{2}-1} a_{2j+1}\omega_{\frac{N}{2}}^{ij}\omega_N^j \bmod q$$

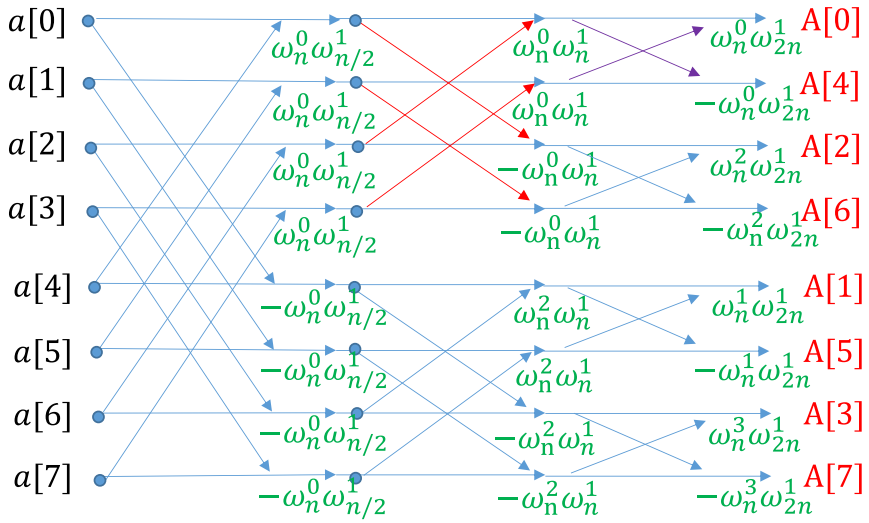
To unify the notation systems, we define $Y_i^{(n)}$ and $Z_i^{(n)}$ as the first half part and the second half part, respectively, of A_i, *i.e.*, $Y_i^{(n)} \stackrel{def}{=} A_i$ for $i \in [0, \frac{N}{2}-1]$ and $Y_i^{(n)} \stackrel{def}{=} A_i$ for $i \in [\frac{N}{2}, N-1]$. Therefore, the Equation 3 is rewritten in a more compact form:

$$Y_i^{(n)} = Y_i^{(n-1)} + \omega_N^i\omega_{2N} \cdot Z_i^{(n-1)} = A_i \bmod q \text{ for } i \in [0, \frac{N}{2}-1]$$

$$Z_i^{(n)} = Y_i^{(n-1)} - \omega_N^i\omega_{2N} \cdot Z_i^{(n-1)} = A_{i+\frac{N}{2}} \bmod q$$

The key observation for the equation above is that $Y_i^{(n)}$ and $Z_i^{(n)}$ has a recursive structure: for example, $Y_i^{(n)}$ and $Z_i^{(n)}$ are computed from a butterfly computation of $Y_i^{(n-1)}$ and $Z_i^{(n-1)}$, $Y_i^{(n-1)}$ and $Z_i^{(n-1)}$ are computed from a butterfly computation of $Y_i^{(n-2)}$ and $Z_i^{(n-2)}$, and so on so forth. Note that in the k-th iteration of such recursion (*i.e.*, $Y_i^{(k)}$ and $Z_i^{(k)}$, and let $K=2^k$), the twiddle factor always has the form $\omega_K^i\omega_{2K}$. As we have known from the standard FFT, the index i in ω_K^i appears in bit-reversed order, therefore, we generalize the modified twiddle factor in our case as shown in Table 1.

As shown in Table 1, the updated twidlde factor is composed of two multiplicative factors which is called the first part and the second part in this paper. The first part of the merged twiddle factor is identical to the standard NTT. We keep the same notations here and do not repeat the proof. It has the form $\omega_N^{\overbrace{00\cdots00}^{n-1 \text{ bits}}}$, $\omega_N^{\overbrace{j_{n-1}0\cdots00}^{n-1 \text{ bits}}}$, $\cdots$, $\omega_N^{\overbrace{j_1j_2\cdots j_{n-2}j_{n-1}}^{n-1 \text{ bits}}}$, for $i=0,1,\cdots,n-1$, respectively. The second part of the merged twiddle factor is $\omega_{2N'}^1$. However, the value of N' depends on the recursive structure of butterfly, for the i-th layer, noted as $N'=N/2^{n-1-i}$ where $N=2^n$. In other words, the second part equals to $\omega_{2\cdot2}^1, \omega_{2\cdot4}^1$, $\cdots$, ω_{2N}^1 for $i=0,1,\cdots,n-1$. Further to unify the second part is rewritten in binary form as $\omega_{2N}^{\overbrace{10\cdots00}^{n \text{ bits}}}$, $\omega_{2N}^{\overbrace{01\cdots00}^{n \text{ bits}}}$, $\cdots$, $\omega_{2N}^{\overbrace{00\cdots01}^{n \text{ bits}}}$ for $i=0,1,\cdots,n-1$.



(a) Generic architecture for NTT with merged twiddle factors ($N = 8$)

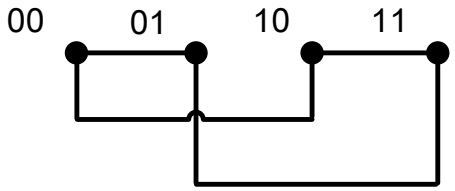
$i = \lvert i_2 i_1 i_0 \rvert_2$	$\omega_N^{00} \cdot \omega_{2N}^{100}$	$\omega_N^{i_2 0} \cdot \omega_{2N}^{010}$	$\omega_N^{i_1 i_2} \cdot \omega_{2N}^{001}$
000		ω_{2N}^{010}	ω_{2N}^{001}
001			
010			ω_{2N}^{101}
011			
100	ω_{2N}^{100}	ω_{2N}^{110}	ω_{2N}^{011}
101			
110			ω_{2N}^{111}
111			

(b) Twiddle factor LUT context ($N = 8$)

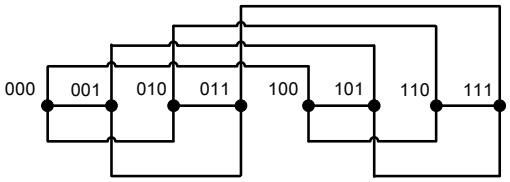
Figure 1: DIT instance with $N = 8$

Alg. 4 formally describes how to construct the twiddle factors for each round of butterfly computation based on our first-part-second-part concept mentioned above. The key variable `Firstpart` is updated in line 6 within the inner j-loop to maintain the desired binary form `Firstpart` $= [j_{\log_2 N-1-i}, \cdots, j_{\log_2 N-1}, 0, \cdots, 0]_2$. The other key variable `Secondpart` is updated in line 8 at the end of the outer i-loop. The first impression on Alg. 4 might be that the size of twiddle factor LUTs is about $\mathcal{O}(NlogN)$: It has $logN$ rounds and each round cosumes $N/2$ twiddle factor for the $N/2$ pairs of input points. However, we deploy a simplified twiddle factor LUT with only $N-1$ elements for our actual hardware design. The key observation for reducing the size of twiddle factor LUT $\{w_i[\cdot]\}_i$ is that many entries in $w_i[\cdot]$ are duplicates and thus redundant. In particular, $w_0[\cdot]$ has only one distinct element $w_0[0]$, $w_1[\cdot]$ has two distinct elements $w_1[0]$ and $w_1[N/2]$, $w_2[\cdots]$ has four distinct elements $w_2[0]$, $w_2[N/4]$, $w_2[2N/4]$, and $w_2[3N/4]$ and so on so forth. Therefore, the total valid entries used in $\{w_i[\cdot]\}$ after eliminating duplicates equal to

$$\sum_{i=0}^{log_2 N-1} 2^i = N - 1 = \mathcal{O}(N)$$



(a) $d = 4$, $ID \in \{P_0, P_1, P_2, P_3\}$



(b) $d = 8$, $ID \in \{P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7\}$

Figure 2: Hypercubes of Dimension $log_2 d = 2$ and $log_2 d = 3$

Input: d node processors
Output: d node processors with connections
Denote the processor IDs as $\{P_0, P_1, \cdots, P_{d-1}\}$ where d is a power-of-2
for $i \leftarrow 0$ **to** $d-1$ **do**
  $[i_{log_2 d-1, \cdots, i_0}]_2 \leftarrow BinRepr(i)$
  for $j \leftarrow 0$ **to** $log_2 d - 1$ **do**
    flip the bit i_j in $[i_{log_2 d-1}, \cdots, i_j, \cdots, i_0]_2$ to make $[i_{log_2 d-1}, \cdots, \bar{i_j}, \cdots, i_0]_2$
    if $[i_{log_2 d-1}, \cdots, \bar{i_j}, \cdots, i_0]_2 > [i_{log_2 d-1}, \cdots, i_j, \cdots, i_0]_2$ **then**
      connect $P_{[i_{log_2 d-1}, \cdots, i_j, \cdots, i_0]_2}$ and $P_{[i_{log_2 d-1}, \cdots, \bar{i_j}, \cdots, i_0]_2}$
return $(P_0, \cdots, P_{d-1})$

Algorithm 5: Construction of the $log_2 d$-dimensional hypercube

Input: $log_2 d$-dimensional hypercubes
Output: communication pattern
Denote the processor IDs as $\{P_0, P_1, \cdots, P_{d-1}\}$
for $k \leftarrow 0$ **to** $log_2 d - 1$ **do**
  /*$d/2$ pairs of processors exchange data in step-k*/
  exchange data between processors $P_{[i_{log_2 d-1}, \cdots, i_{log_2 d-1-k}, \cdots, 0]_2}$ and $P_{[i_{log_2 d-1}, \cdots, \overline{i_{log_2 d-1-k}}, \cdots, 0]_2}$ which differ at $i_{log_2 d-1-k}$
return $c(x)$

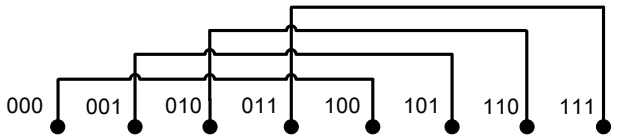
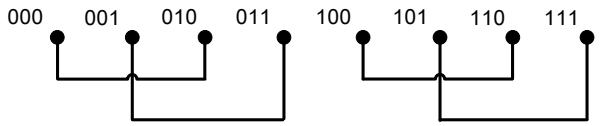
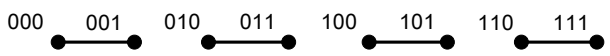
Algorithm 6: Subcube-doubling communication in $log_2 d$-dimensional hypercube

2.2 $log_2 d$-Dimensional Hypercube Multiprocessors

In this subsection, we introduce the hypercube topology which fits the parallelized version of NTT algorithm. The hypercube is also the basis for hardware architecture proposed in this work.

Before detailing the parallel NTT algorithm and hardware, the computing model used in this paper must be clarified. There are d identical node processors organized in a hypercube of dimension $log_2 d$. Each node processor includes one butterfly unit and some

Table 2: Subcube-doubling communication in 3-dimensional hypercube

steps	connections
Step-(0)	
Step-(1)	
Step-(2)	

storage (N/d NTT points). Roughly speaking, this $log_2 d$-dimensional hypercube structure should increase the speed of sequential NTT algorithm by d times. Fig. 2a illustrates the hypercube architecture of dimension 2 where 4 node processors (*i.e.*, P_0, P_1, P_2, P_3 labeled as it binary form '00', '01', '10', and '11') are implemented. The node processors are sparsely connected with each other where any one of them are connected to the other 2 processors. For example, P_0 is only connected to P_1 and P_2, P_1 is only connected to P_0 and P_3. Fig. 2b illustrates the hypercube architecture of dimension 3 where 8 node processors (*i.e.*, $P_0, P_1, \cdots, P_7$ labeled as it binary form '000', '001', ..., and '111') are implemented. The node processors are sparsely connected with each other where any one of them are connected to the other 3 processors. For example, P_0 is only connected to P_1, $P2$ and $P4$, P_1 is only connected to P_0, P_3 and P_5.

Alg. 5 formally describes how to construct the $log_2 d$-dimensional hypercube by sparsely connecting d node processors. The key idea here is that for each processor P_i, rewrite the index i in binary form as $[i_{log_2 d-1}, \cdots, i_0]_2$, and connects those processors P_j whose index $j = [j_{log_2 d-1}, \cdots, j_0]_2$ differs only 1 bit compared with i. In particular, each node processor connects only to $log_2 d$ other node processors in this $log_2 d$-dimensional hypercube topology. The if condition in the for-loop in Alg. 5 helps rule out the possibility of connecting the same pair of nodes repeatedly. When the computation continues in the hypercube, the intermediate data generated in each round of computations typically requires exchange between node processors. This type of data exchange is referred to as 'subcube-doubling' communication in the literature. There are in total $log_2 d$ rounds of exchange during the communication as described in Alg. 6: In step-(k), each node processor P_i with index $i = [i_{log_2 d-1}, \cdots, i_0]_2$ exchanges data with P_j whose index j differs at the $log_2 d - 1 - k$-th bit.

An illustration instance with $d = 8$ for subcube-doubling algorithm (Alg. 6) is given in Table 2. $log_2 d = 3$ communication steps are required in this example: In step-(0), processor $P_{[i_2,i_1,i_0]_2}$ connects processor $P_{[\overline{i_2},i_1,i_0]_2}$ which differs at i_2, and there are $d/2 = 4$ such pairs of connections, *i.e.*, $P_0 - P_4, P_1 - P_5, P_2 - P_6, P_3 - P_7$; In step-(1), processor $P_{[i_2,i_1,i_0]_2}$ connects processor $P_{[i_2,\overline{i_1},i_0]_2}$ which differs at i_1; Finally, in step-(2), processor $P_{[i_2,i_1,i_0]_2}$ connects processor $P_{[i_2,i_1,\overline{i_0}]_2}$ which differs at i_0.

2.3 A Useful Equivalent Notation: |PID|Local M

Assume that N points are stored in the global array $a[\cdot] = \{a_{N-1}, \cdots, 0\}$ or simplified as $a[\cdot] = \{a_i\}_{i=N-1,\cdots,0}$, and the elements in the array are assigned evenly to d node

processors for storage and processing. Then the array address based notation uses a $logN$-bit integer $i = i_{logN-1} \cdots i_{0_2}$:

$$i_{logN-1} \cdots i_{k+1} | i_k \cdots i_{k-logd+1} | i_{k-logd} \cdots i_0$$

to indicate that cosecutive $logd$ bits $i_k \cdots i_{k-logd+1}$ are chosen to specify the data-to-processor allocation.

In general, since any $logd$ bits can be used to form the processor ID number, it is easier to concatenate the bits representing the processor ID into one group denoted by 'PID', and refers to the remaining $logN - logd$ bits, which are concatenated to form the local array address, as 'Local M'. This paper uses the following equivalent notation, where the leading d bits are always used to identify the processor ID number.

$$|\text{PID}|\text{Local } M = |\underbrace{i_k \cdots i_{k-logd+1}}_{logd}|\overbrace{i_{N-1} \cdots i_{k+2} i_{k+1}}^{N-k-1}\overbrace{i_{k-d} \cdots i_1 i_0}^{k-logd+1}$$

Table 3 shows the details about the data allocation for hypercube processor array after a naturally ordered input series of $N = 32$ elements are divided among $d = 4$ processors using one particular cyclic block mapping $i_4 i_3 | i_2 i_1 i_0$. For instance, to locate $a_m = a_{26}$, one writes down $m = 26 = 11010_2 = i_4 i_3 i_2 i_1 i_0$, from which one knows that a_{26} is stored in $a[r]$, $r = i_4 i_3 | i_2 i_1 i_0 = 11 | 010_2 = 26$, meaning that $a[26] = a_{26}$ (the element a_{26} is located in $a[26]$) is allocated by processor $P_{i_4 i_3} = P_{01}$.

Table 3: Local data in processor $P_{i_4 i_3}$ expressed in terms of global array element $a[m], m = i_4 i_3 i_2 i_1 i_0$ for the notation $i_4 i_3 | i_2 i_1 i_0$

|PID|Local M $i_4 i_3 \| i_2 i_1 i_0$	$P_{i_4 i_3} = P_{00}$ $a[m]$	|PID|Local M $i_4 i_3 \| i_2 i_1 i_0$	$P_{i_4 i_3} = P_{01}$ $a[m]$	|PID|Local M $i_4 i_3 \| i_2 i_1 i_0$	$P_{i_4 i_3} = P_{10}$ $a[m]$	|PID|Local M $i_4 i_3 \| i_2 i_1 i_0$	$P_{i_4 i_3} = P_{11}$ $a[m]$
00|000	$a[0]$	01|000	$a[8]$	10|000	$a[16]$	11|000	$a[24]$
00|001	$a[1]$	01|000	$a[9]$	10|000	$a[17]$	11|000	$a[25]$
00|010	$a[2]$	01|000	$a[10]$	10|000	$a[18]$	11|000	$a[26]$
00|011	$a[3]$	01|000	$a[11]$	10|000	$a[19]$	11|000	$a[27]$
00|100	$a[4]$	01|000	$a[12]$	10|000	$a[20]$	11|000	$a[28]$
00|101	$a[5]$	01|000	$a[13]$	10|000	$a[21]$	11|000	$a[29]$
00|110	$a[6]$	01|000	$a[14]$	10|000	$a[22]$	11|000	$a[30]$
00|111	$a[7]$	01|000	$a[15]$	10|000	$a[23]$	11|000	$a[31]$

On the other hand, when the input elements are stored in **a** in bit-reversed order, *i.e.,* $a[r] = a_m$ where $m = i_{n-1} i_{n-2} \cdots i_0$, and $r = i_0 \cdots i_{n-2} i_{n-1}$, then the equivalent notation is as follows:

$$|\text{PID}|\text{Local } M = |\underbrace{i_{k-logd+1} \cdots i_k}_{logd}|\overbrace{i_0 \cdots i_{k-logd}}^{k-logd+1}\overbrace{i_{k+1} \cdots i_{n-1}}^{N-k-1}$$

Table 4 shows the details about the data allocation for hypercube processor array after an inverse ordered input series of $N = 32$ elements are divided among $d = 4$ processors using one particular cyclic block mapping $i_0 i_1 | i_2 i_3 i_4$. For instance, to locate $a_m = a_{26}$, one writes down $m = 26 = 11010_2 = i_4 i_3 i_2 i_1 i_0$, from which one knows that a_{26} is stored in $a[r]$, $r = i_0 i_1 | i_2 i_3 i_4 = 01 | 011_2 = 11$, meaning that $a[11] = a_{26}$ (the element a_{26} is located in $a[11]$) is allocated by processor $P_{i_0 i_1} = P_{01}$.

2.4 First attempt: parallel in-place FFTs without inter-processor permutations

Consider the $DIT_{NN \to RN}$ algorithm (Alg. 3) and use the cyclic block mapping introduced in the last subsection. For $N = 32, d = 4$, the computation is depicted below:

Table 4: Local data in processor $P_{i_0 i_1}$ (bit reversed) expressed in terms of global array element $a[r] = a_m, r = i_0 i_1 i_2 i_3 i_4, m = i_4 i_3 i_2 i_1 i_0$ for the notation $i_0 i_1 | i_2 i_3 i_4$

|PID|Local M $i_0 i_1 \| i_2 i_3 i_4$	$P_{i_0 i_1} = P_{00}$ $a[r]$	|PID|Local M $i_0 i_1 \| i_2 i_3 i_4$	$P_{i_0 i_1} = P_{01}$ $a[r]$	|PID|Local M $i_0 i_1 \| i_2 i_3 i_4$	$P_{i_0 i_1} = P_{10}$ $a[r]$	|PID|Local M $i_0 i_1 \| i_2 i_3 i_4$	$P_{i_0 i_1} = P_{11}$ $a[r]$
00|000	$a[0]$	01|000	$a[8]$	10|000	$a[16]$	11|000	$a[24]$
00|001	$a[1]$	01|000	$a[9]$	10|000	$a[17]$	11|000	$a[25]$
00|010	$a[2]$	01|000	$a[10]$	10|000	$a[18]$	11|000	$a[26]$
00|011	$a[3]$	01|000	$a[11]$	10|000	$a[19]$	11|000	$a[27]$
00|100	$a[4]$	01|000	$a[12]$	10|000	$a[20]$	11|000	$a[28]$
00|101	$a[5]$	01|000	$a[13]$	10|000	$a[21]$	11|000	$a[29]$
00|110	$a[6]$	01|000	$a[14]$	10|000	$a[22]$	11|000	$a[30]$
00|111	$a[7]$	01|000	$a[15]$	10|000	$a[23]$	11|000	$a[31]$

$\|i_4 i_3 \| i_2 i_1 i_0$	$\|i_4 i_3 \| i_2 i_1 i_0$	$\|\tau_4 i_3 \| i_2 i_1 i_0$	$\|\tau_4 \tau_3 \| i_2 i_1 i_0$	$\|\tau_4 \tau_3 \| \tau_2 i_1 i_0$	$\|\tau_4 \tau_3 \| \tau_2 \tau_1 i_0$
Initial Map	$\Longleftrightarrow$	$\Longleftrightarrow$			

The initial map indicates that the processor P_k initially holds the elements $a[8k], \cdots, a[8k+7]$ for $k = 0, 1, 2, 3$. The shorthand notation previously used for sequential NTT is augmented by two additional symbols. The double-headed arrow $\Longleftrightarrow$ indicates that $\frac{N}{d}$ data elements must be exchanged between processors in advance of butterfly computation. In our example, the NTT takes 5 rounds where the first two rounds require data exchange among proessors and the last three rounds do not require data exchange. The symbol i_k identifies two things:

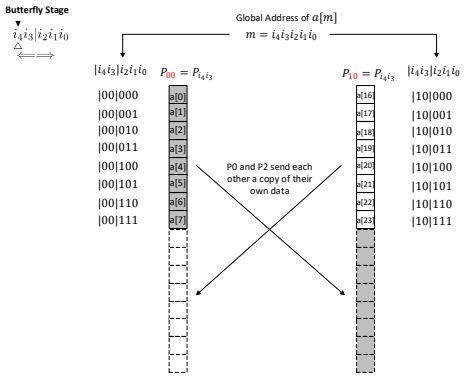
- First, it indicates the input source of external data: the incoming data from another processor are the elements whose addresses differ from a processor's own data in bit i_k.
- Second, it indicates that all pairs of processors whose binary ID number differ in bit i_k send each other a copy of their own data.

The required data communications before the first stage of butterfly computation (step-0) are explicitly depicted in Fig. 3a and Fig. 3b: P_0 swaps data with P_2 such that $a[i]$ pairs with $a[i + 16]$ to perform the required butterfly computation in the same processor for $i = 0, \cdots, 7$, and P_1 swaps data with P_3 such that $a[i]$ pairs with $a[i + 16]$ to perform the required butterfly computation in the same processor for $i = 8, \cdots, 15$; the required data communications before the second stage of butterfly computation (step-1) are depicted in Fig. 3c and Fig. 3d: P_0 swaps data with P_1 such that $a[i]$ pairs with $a[i+8]$ to perform the required butterfly computation in the same processor for $i = 0, \cdots, 7$, and P_2 swaps data with P_3 such that $a[i]$ pairs with $a[i + 8]$ to perform the required butterfly computation in the same processor for $i = 16, \cdots, 23$. The last three steps (step-2,3,4) do not require data swaps since all elements needed for butterfly computataion are already within the precessor: for example, in step-2, $a[0]$ pairs $a[4]$, $a[1]$ pairs $a[5]$, $a[2]$ pairs $a[6]$, $a[3]$ pairs $a[7]$, which are all located within processor P_0.

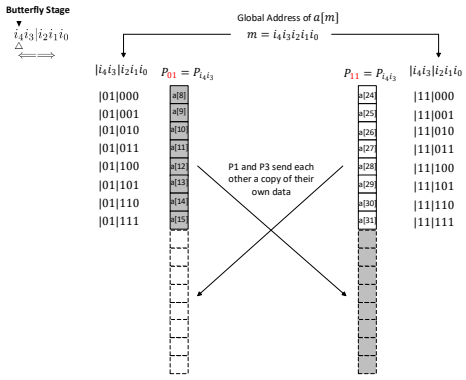
Remarks The parallel in-place NTT without inter-processor permutations approach employs *data exchange between a pair of processors*. That is, one processor's initial complement of data may swap with that of another processor. With use of this type of data exchange, N/d butterfly computations are performed in parallel at the cost of a number of N/d data swaps per processor.

2.5 Second attempt: Parallel NTTs with Inter-processor permutations

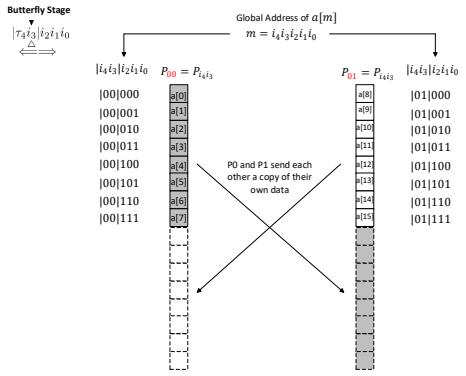
In this subsection, we discuss the class of parallel NTTs which employ inter-processor data permutations. Similar to the one presented in the previous subsection which evenly



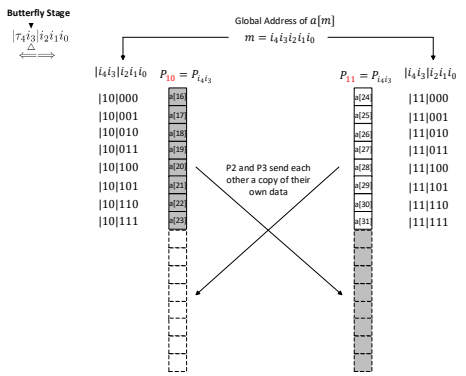
(a) In round-0, Data sent and received by processors P_0 and P_2



(b) In round-0, Data sent and received by processors P_1 and P_3



(c) In round-1, Data sent and received by processors P_0 and P_1



(d) In round-1, Data sent and received by processors P_2 and P_3

Figure 3: An illustrative example for parallelizing in-place NTT($N = 32, d = 4$) without inter-processor permutations

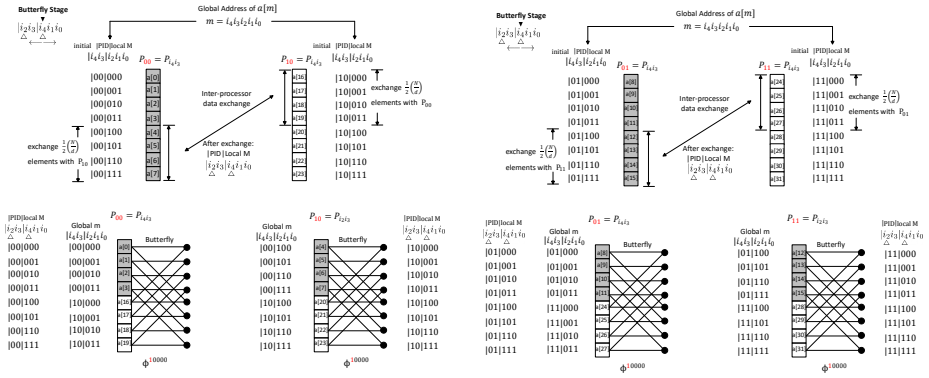
distributes all butterfly computations among the processors, the new method also reduces the message length from $\frac{N}{d}$ elements to $\frac{1}{2}\frac{N}{d}$ in each of the $log_2 d + 1$ concurrent message exchanges.

The complete algorithm We use the shorthand notation we have developed with symbols $\triangle$ and $\blacktriangledown$, the complete parallel algorithm corresponding to DIT_{NR} NTT is represented below for the $N = 32$ example.

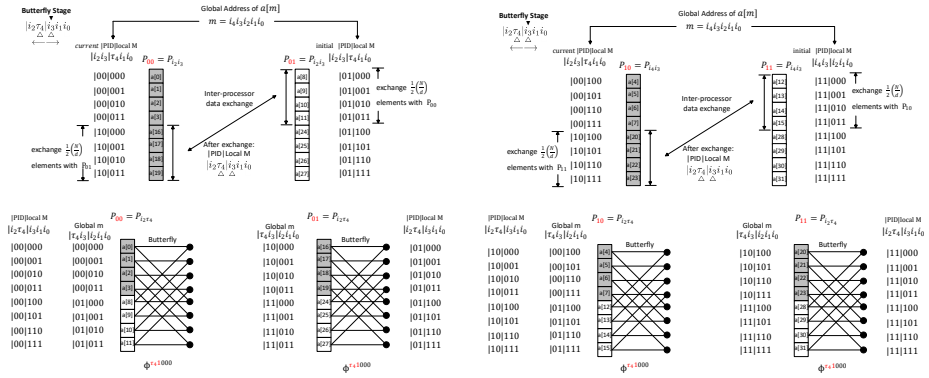
$\lvert i_4i_3\rvert i_2i_1i_0$	$\lvert i_2i_3\rvert i_4i_1i_0$	$\lvert i_2\tau_4\rvert i_3i_1i_0$	$\lvert\tau_3\tau_4\rvert i_2i_1i_0$	$\lvert\tau_3\tau_4\rvert\tau_2i_1i_0$	$\lvert\tau_3\tau_4\rvert\tau_2\tau_1i_0$
Initial Map	$\triangle\longleftrightarrow\triangle$	$\triangle\longleftrightarrow\triangle$	$\triangle\longleftrightarrow\triangle$		

To provide complete information for this example, in the initial map (before performing the first stage butterfly computation), the input data are distributed as the element $a_{i_4i_3i_2i_1i_0}$ can be found in $A[i_2i_1i_0]$ in processor $P_{i_4i_3}$. For example, $a[19] = a_{19}$ is shown to be initially in $A[3]$ in P_2 and $A[14] = a_{14}$ in $A[6]$ in P_1 in Fig. 4a since $19 = 10|011_2$ and $14 = 01|110_2$.

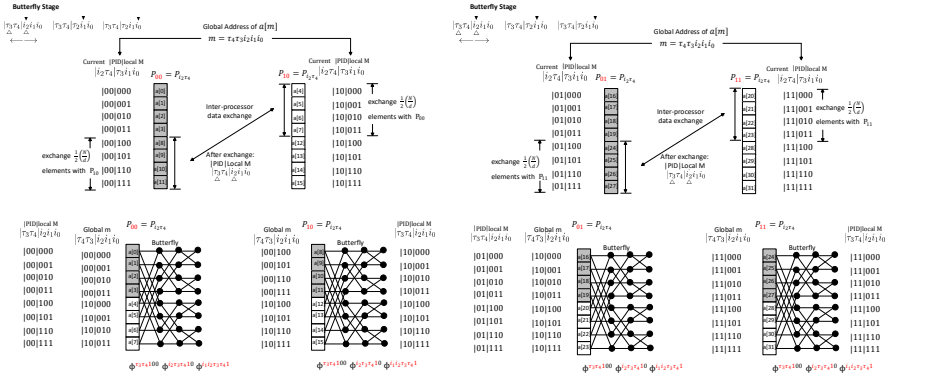
To prepare data for each processor in the first round of butterfly computation where P_0 connects P_2 and P_1 connects P_3 due to the hypercube structure, P_0 swaps the second half of his local array with the first half of P_2's local array, and P_1 swaps the second half



(a) In round-0, DIT_{NR} butterfly computation with data migration between processors P_0 and P_2, and P_1 and P_3, respectively



(b) In round-1, DIT_{NR} butterfly computation with data migration between processors P_0 and P_1, and P_2 and P_3, respectively



(c) In round-2/3/4, DIT_{NR} butterfly computation with data migration between processors P_0 and P_2, and P_1 and P_3, respectively

Figure 4: An illustrative example for parallelizing in-place NTT($N = 32, d = 4$) with inter-processor permutations

of his local array with the first half of P_3's local array as dipicted in Fig. 4a. The symbols $\overset{\triangle}{}$ are used to locate the exact position of the element a_i after such data swap: the bit i_k, which has just been permuted from PID to Local M, and the bit i_ℓ, which has just

been permuted from Local M to the PID. In our case, the notation $|\underset{\triangle}{i_2}i_3|\overset{\blacktriangledown}{\underset{\triangle}{i_4}}i_1i_0$ is used which means bit i_4 in the PID and bit i_2 in the local M switch their positions in the shorthand notation (denoted by the symbols $\triangle$) making the memory mapping changed to $i_2i_3|i_4i_1i_0$, which means that the data in $a[i_4i_3i_2i_1i_0]$ can now be found in $A[i_4i_1i_0]$ in $P_{i_2i_3}$. For example, $a[19] = a_{19}$ is relocated to $A[7]$ in P_0 ($A[7]$ means the 7-th element for P_0's local array) after the inter-processor permutation shown in Fig. 4a since $10|011_2$ is changed to $00|111_2$; $a[14] = a_{14}$ is relocated to $A[2]$ in P_3 after the inter-processor permutation shown in Fig. 4a since $01|110_2$ is changed to $11|010_2$. After the memory swap, all data are located correctly in the corresponding processors to perfom the first round of butter fly computation. The symbol $\blacktriangledown$ is used to indicate the pairs of elements for the butterfly computation in each processor, *i.e.*, $|\underset{\triangle}{i_2}i_3|\overset{\blacktriangledown}{\underset{\triangle}{i_4}}i_1i_0$ means $A[i_2i_30i_1i_0]$ should pair with $A[i_2i_31i_1i_0]$ to complete elementary butterfly unit computation for $P_{i_2i_3}$. Also, the index i_k that the symbol $\blacktriangledown$ points to is changed to τ_k for showing that this particular round of butterfly computation is completed. A quick observation is that there are $i-1$ indices changed to τ in the notation for the i-th round of computation: for example, $|\underset{\triangle}{i_2}i_3|\overset{\blacktriangledown}{\underset{\triangle}{i_4}}i_1i_0$ represents the first round where no τ indices exist; $|\underset{\triangle}{i_2}\tau_4|\overset{\blacktriangledown}{\underset{\triangle}{i_3}}i_1i_0$ represents the second round where one τ index (i_4 changed to τ_4) exists and *etc.*

For the second round of butterfly computation where P_0 connects P_1 and P_2 connects P_3, P_0 swaps the second half of his array with the first half of P_1's local array, and P_2 swaps the second half of his local array with the first half of P_3's local array as dipicted in Fig. 4b. a similar notation $|\underset{\triangle}{i_2}\tau_4|\overset{\blacktriangledown}{\underset{\triangle}{i_3}}i_1i_0$ is used to denote the memory swap: bit τ_4 in the PID and bit i_3 in the local M switch their positions in the shorthand notation making the memory mapping changed from previous $i_2i_3|i_4i_1i_0$ to $i_2i_4|i_3i_1i_0$. For example, a_{19} which is previously stored in $A[7]$ from P_0 is now changed to $A[3]$ from P_1 since $19 = 10|011_2$ is rearranged to $01|011_2$. Moreover, $A[i_2i_40i_1i_0]$ pairs with $A[i_2i_41i_1i_0]$ to complete elementary butterfly unit computation for $P_{i_2i_4}$ for all i_2, i_4, i_1, i_0.

For the third round of butterfly computation where P_0 connects P_2 and P_1 connects P_3, P_0 swaps the second half of his local array with the first half of P_2's local array, and P_1 swaps the second half of his local array with the first half of P_3's local array as dipicted in Fig. 4c. This swapping pattern is captured in the notation $|\underset{\triangle}{\tau_3}\tau_4|\overset{\blacktriangledown}{\underset{\triangle}{i_2}}i_1i_0$ indicating the previous $i_2i_4|i_3i_1i_0$ is changed to $i_3i_4|i_2i_1i_0$ due to the $\triangle$ annotations. For example, a_{19} which is previously stored in $A[3]$ from P_1 is still preserved in $A[3]$, P_1 since $19 = 10|011_2$ is rearranged to $01|011_2$. The $\blacktriangledown$ annotation indicates that $A[i_3i_40i_1i_0]$ pairs with $A[i_3i_41i_1i_0]$ to complete elementary butterfly unit computation for $P_{i_3i_4}$ for all i_3, i_4, i_1, i_0.

There are no inter-processor data swapping for the last two rounds, *i.e.*, the fourth and the fifth round of butterfly computation. However, the $\blacktriangledown$ annotation helps distinguish which two data elements should pair to complete the elementary butterfly unit computation inside the processor: in the fourth round, $A[i_3i_4i_20i_0]$ pairs with $A[i_3i_4i_21i_0]$ for $P_{i_3i_4}$, and in the fifth round, $A[i_3i_4i_2i_10]$ pairs with $A[i_3i_4i_2i_11]$ for $P_{i_3i_4}$. The computations in the fourth and fifth round are merged to Fig. 4c.

After all five rounds of butterfly computation are completed, the NTT results are stored in the array $a[\cdot]$ but the position is rearranged: the output data element $A_{i_0i_1i_2i_3i_4}$, which overwrites the data in $a[i_4i_3i_2i_1i_0]$, is finally contained in $A[i_2i_1i_0]$ in $P_{i_3i_4}$. Such arrangement for the data mapping for the output elements is observed as following:

- The *in-place* butterfly computation in the DIT_{NR} algorithm ensures $a[i_4i_3i_2i_1i_0] = a^{(5)}_{i_4i_3i_2i_1i_0} = A_{i_0i_1i_2i_3i_4}$ where $A_{i_0i_1i_2i_3i_4} = \sum_j a_j(\omega_N^{i_0i_1i_2i_3i_4} \cdot \omega_{2N}^1)^j$
- The final mapping $|\tau_3\tau_4|\tau_2\tau_1\tau_0$ indicates that the final content in $a[i_4i_3i_2i_1i_0]$ is now

located in $a[i_2 i_1 i_0]$ in processor $P_{i_3 i_4}$ (rather than the initially assigned processor $P_{i_4 i_3}$)

For example, P_0 has stored $a[0] - a[7]$ where $a[0]$ computes $A[0]$, $a[1]$ computes $A[16]$, and *etc.*; P_1 has stored $a[16] - a[23]$ where $a[16]$ computes $A[1]$, $a[17]$ computes $A[17]$, and *etc.*; P_2 has stored $a[8] - a[15]$ where $a[8]$ computes $A[2]$, $a[9]$ computes $A[18]$, and *etc.*; P_3 has stored $a[24] - a[31]$ where $a[24]$ computes $A[3]$, $a[25]$ computes $A[19]$, and *etc.*

Remarks on the correctness of the notation Because i_k was in PID session before the switch, $i_k = 1$ in one processor, and $i_k = 0$ in the other processor. On the other hand, because i_ℓ was in Local M session before the switch, $i_\ell = 0$ for half of the data, and $i_\ell = 1$ for another half of the data. Consequently, the value of i_k, the PID bit, is equal to i_ℓ, the local M bit, for half of the data elements in each processor, and the notation which represents the switch of these two bits identifies both the PID of the other processor as well as the data to be sent out or received. To depict exactly what happens, the data exchange between two processors and the butterfly computation represented by $|i_2 i_3|i_4 i_1 i_0$ is shown in its entirety in Fig. 4a and 4b.

Input: a polynomial ring R_q, and NTT points N, input $\mathbf{a} = (a[0], \cdots, a[N-1])$
Output: $NTT(\mathbf{a})) = \mathbf{A} = (A[0], \cdots, A[N-1])$

Initialize the hypercube connections between d processors as described in Alg. 5
Initialize the merged twiddle factor look-up table $\{w_i\}$ as described in Alg. 4
/*arrange the data array $a[\cdot]$ in natural order*/
Initialize the data $a[i_{log_2 N - log_2 d - 1} \cdots i_1 i_0]$ in $P_{i_{log_2 N - 1} \cdots i_{log_2 N - log_2 d}}$ with $a[i_{log_2 N - 1} \cdots i_1 i_0]$ for all $i_{log_2 N - 1}, \cdots, i_0$
/*perform the first $log_2 d + 1$ round of computations where inter-processor data swapping is required*/
for $j \leftarrow 0$ **to** $log_2 d$ **do**
  if $j \neq log_2 d$ **then**
    exchange the first half of data in $P_{i_{log_2 N - 1} \cdots i_{log_2 N - 1 - j} \cdots i_{log_2 N - log_2 d}}$ w.r.t. $i_{log_2 N - 1 - j} = 0$ with the second half of data in $P_{i_{log_2 N - 1} \cdots i_{log_2 N - 1 - j} \cdots i_{log_2 N - log_2 d}}$ w.r.t. $i_{log_2 N - 1 - j} = 1$
  else
    exchange the first half of data in $P_{i_{log_2 N - 1} \cdots i_{log_2 N - 1 - j} \cdots i_{log_2 N - log_2 d}}$ w.r.t. $i_{log_2 N - 1} = 0$ with the second half of data in $P_{i_{log_2 N - 1} \cdots i_{log_2 N - 1 - j} \cdots i_{log_2 N - log_2 d}}$ w.r.t. $i_{log_2 N - 1} = 1$
  perform within each processor $P_{i_{log_2 N - 1} \cdots i_{log_2 N - 1 - j} \cdots i_{log_2 N - log_2 d}}$ the $\frac{N}{2d}$ butterfly computations (round-j butterfly)
/*perform the first $log_2 N - log_2 d - 1$ round of computations where inter-processor data swapping is not required*/
for $j \leftarrow log_2 d + 1$ **to** $log_2 N - 1$ **do**
  perform within each processor $P_{i_{log_2 N - 1} \cdots i_{log_2 N - 1 - j} \cdots i_{log_2 N - log_2 d}}$ the $\frac{N}{2d}$ butterfly computations (round-j butterfly)
return *the data in all d processors as* **A**

Algorithm 7: Parallel Hypercube NTT

Twiddle Factor LUT Distribution Let us discuss in details on the distribution of the twiddle factor LUT within each butterfly processor here. In the first $logd + 1$ rounds of butterfly computations, memory swapping occurs and each butterfly processor utilizes only 1 twiddle factor; in the next $logN - logd - 1$ rounds, no memory swapping occurs and the number of twiddle factors utilized in eah butterfly processor increases exponentially

(starting with 2). Therefore, the total number of twiddle factors (the depth of twiddle factor LUT) in each processor is:

$$\sum_{i=1}^{logd+1} 1 + \sum_{i=1}^{logN-logd-1} 2^i = \frac{N}{d} + logd - 1$$

A concrete example for the twiddle factor LUT distribution can also be found in Fig 4. In the first 3 rounds, each processor uses only 1 twiddle factor, *i.e.*, Φ^{10000} where Φ denotes the $2N$-th primitive root of unity ω_{2N}. In the 4-th round, each processor uses 2 twiddle factors, *i.e.*, $\Phi^{\tau_4 1000}$ for $\tau_4 \in \{0, 1\}$. In the 5-th round, each processor uses 4 twiddle factors, *i.e.*, $\Phi^{\tau_3 \tau_4 100}$ for $\tau_4 \in \{0, 1\}, \tau_3 \in \{0, 1\}$. Therefore, each processor stores $1 + 2 + 4 = 7$ twiddle factors for the proposed hypercube NTT architecture.

2.6 Butterfly Processor

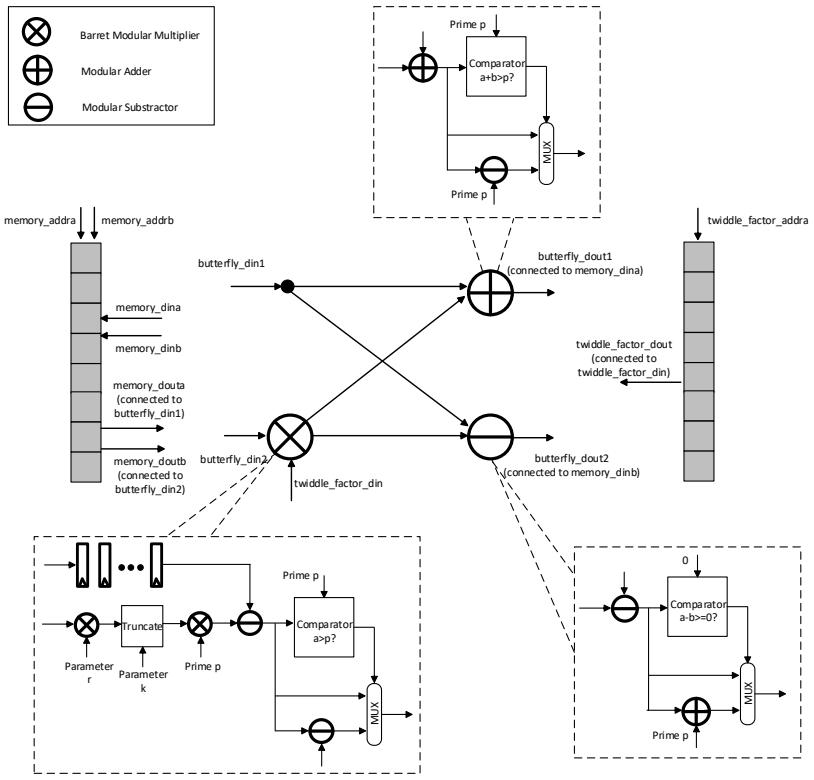


Figure 5: Internal structure of butterfly processor

Design Overview To perform the butterfly computations and related memory access in each processor efficiently as illustrated in Fig. 4, a butterfly processor architecture is proposed. Fig. 5 depicts the internal structure of the butterfly processor. Two memory blocks are instantiated: one dual-port RAM for the $\frac{N}{d}$ points, namely $a_{\frac{N}{d} \cdot i} - a_{\frac{N}{d} \cdot i + \frac{N}{d} - 1}$ for $i \in [d]$, and one single-port ROM for the $\frac{N}{d} + logd - 1$ precomputed twiddle factors. At first, two points which forms the pair for the elementary butterfly computation unit, *e.g.,* a_i and a_j are simultaneously extracted on `memory_douta` and `memory_doutb` from the dual-port RAM. Then a_i and a_j are fed to the input ports `butterfly_din1` and `butterfly_din2` of the butterfly structure. This butterfly structure consists of one Barret modular multiplier (apply Alg. 8), one modular adder (apply Alg. 9), and one modular subtractor (apply

Alg. 10). After the butterfly computation is completed, the results $a_i + a_j \cdot w$ and $a_i - a_j \cdot w$ appear at the output ports `butterfly_dout1` and `butterfly_dout2`. Finally, the two results are simultaneously written back to the RAM through the ports `memory_dina` and `memory_dinb`. It is worth mentioning that the butterfly processor is fully pipelined such that a pair of valid data `butterfly_dout1` and `butterfly_dout2` is written back to the RAM every clock cycle, which maintains a relatively high throughput of butterfly computation. This characteristic is crucial for high speed implementation of FHE scheme since the parameter N (the number of NTT points) is typically set to be large (typical value is around 1k) for maintaining the hardness of the (Ring-) LWE problem.

Input: two integers a and b over $\mathbb{Z}_q$
Output: $a \cdot b \in \mathbb{Z}_q$
Precompute an integer $k = \lceil log_2 q \rceil$
Precompute an integer $r = \lfloor \frac{4^k}{q} \rfloor$
Calculate $x = a \cdot b$
Calculate $t = x - \lfloor \frac{xr}{4^k} \rfloor \cdot q$
if $t < q$ **then**
  return t
else
  return $t - q$

Algorithm 8: Barret-Reduction based Modular Multiplication

Input: two integers a and b over $\mathbb{Z}_q$
Output: $a + b \in \mathbb{Z}_q$
Calculate $t = a + b$
if $t < q$ **then**
  return t
else
  return $t - q$

Algorithm 9: Modular Addition

Input: two integers a and b over $\mathbb{Z}_q$
Output: $a - b \in \mathbb{Z}_q$
Calculate $t = a - b$
if $t \geq 0$ **then**
  return t
else
  return $t + q$

Algorithm 10: Modular Subtraction

Timing analysis Let one unit denote the delay of one clock cycle, T_{mul} denote the delay of standard integer multiplication, T_{modmul} denote the delay of Barret reduction based modular multiplication algorithm, and $T_{modadd}(T_{modsub})$ denote the delay of modular addition(subtraction) algorithm. The delay of one butterfly computation is calculated as

$$T_{butterfly} = T_{swap} + T_{mul} + T_{modmul} + T_{modadd}$$

Note that the proposed butterfly processor is fully pipelined and therefore it takes $T_{butterfly} + \frac{N}{2d} - 1$ to process $\frac{N}{2d}$ butterfly computations.

Fully pipelined computation The key point for fully pipelined butterfly computation is to streamline the generation of memory address, *i.e.*, `memory_addra` and `memory_addrb` in Fig. 5. Note that the NTT butterfly address generation pattern is rather complicated: it varies distinctly in different butterfly computation round. It is desirable to implement some other simpler patterns and later combine these simple patterns to create the address generation. In our design, we use five registers, `cntb`, `roundi`, `dist`, `cnt`, and `base` to assist the generation of `memory_addra` and `memory_addrb` in every clock cycle:

- `cntb`: base counter register, used to generate the basic logic pattern, *i.e.*, a square wave signal with period of $\frac{N}{d}$ cycles
- `roundi`: butterfly round register, used to indicate the current round of butterfly computation
- `dist`: distance register, used to record the distance between `memory_addra` and `memory_addrb` s.t. `memory_addrb`=`memory_addra`+`dist`
- `cnt`: counter register, used to indicate the incremental offset value for generating `memory_addra`
- `base`: the (basis) starting address for `memory_addra` in each round of butterfly calculation

Moreover, we use two pre-computed arrays **blk** and **dist** to help generate the correct values in the five registers mentioned above. **blk** is related to the variable `NumOfGroups` in Alg. 3, and indicates the number of butterfly blocks in every round of butterfly calculation and has $log_2 N$ elements; **dist** is related to the variable `Distance` in Alg. 3, and indicates the distance between `memory_addra` and `memory_addrb` in every round of butterfly calculation and has $log_2 N$ elements. The construction **blk** goes like this: The first $logd$ elements are always 1; starting from the $(logd+1)$-th element down to the last one, *i.e.* the last $logN - logd$ elements formulate a geometric sequence with initial value 1 and common ratio 2. The construction **dist** goes like this: The first $logd$ elements are always $\frac{N}{2d}$; Then the last $logN - logd$ elements formulate a geometric sequence with initial value $\frac{N}{2d}$ and common ratio $\frac{1}{2}$. For example, if $N = 32, d = 4$, then $\mathbf{blk} = \{1, 1, 1, 2, 4\}$ and $\mathbf{dist} = \{4, 4, 4, 2, 1\}$.

The generation of `memory_addra` and `memory_addrb` in Fig. 5 is formally described in Alg. 11. The generated addresses basically map to the memory location of two butterfly inputs (`butterfly_din1` and `butterfly_din2` shown in Fig. 5). A more concrete example for when $N = 32, d = 4$ is depicted in Fig. 6. Every register including `cntb`, `roundi`, `dist`, `cnt`, and `base` has 5 phases each of which corresponds to one of the $logN = 5$ rounds of butterfly computation. Each phase costs 4 clock cycles. For example, `cntb` updates as $0, 1, 2, 3$ in every phase; whereas `roundi` updates as i in phase-$i (i = 0, 1, 2, 3, 4)$. We also assume the calculation of memory address (step6-step7 in Alg. 11) takes one clock cycle delay and thus the result appearing in `memory_addra` and `memory_addrb` is delayed by one clock cycle as shown in Fig. 6. The sequence of `memory_addra` and `memory_addrb` can be interpreted as follows: In the first clock cycle of phase-0, `memory_addra` outputs 0 and `memory_addrb` output 4 (extracting $a[0]$ and $a[4]$ from the local memory $a[\cdot]$ within the node processor); in the second clock cycle, `memory_addra` outputs 1 and `memory_addrb` outputs 5, and so on so forth. Finally, in the first clock cycle of phase-4, `memory_addra` outputs 0 and `memory_addrb` outputs 1; in the second clock cycle, `memory_addra` outputs 2 and `memory_addrb` outputs 3, and so on so forth.

Based on the memory address generation pattern described in Fig. 6, we can finally introduce the complete memory address control logic (See Fig. 7) used in the proposed

butterfly processor. Again, all registers are represented in 5 phases where each phase costs 3 clock cycles. The register `current_state` indicates one of the three current status in each phase as follows:

- `ADDR_RD`: In this state, butterfly processor reads the corresponding butterfly inputs (`butterfly_din1` and `butterfly_din2` in Fig. 5) from memory in a pipelined fashion
- `IDLE`: This state is optional, and is used only if N is relatively small. For more details, refer to the next section.
- `ADDR_WR`: In this state, butterfly processor writes back the computed results (`butterfly_dout1` and `butterfly_dout2` in Fig. 5) to memory in a pipelined fashion.

Note that the entire butterfly computation takes $logN = 5$ iterations. If N is relatively small and d is relatively large, the state register transits by `ADDR_RD` $\rightarrow$ `IDLE` $\rightarrow$ `ADDR_WR` in each iteration; otherwise, the state register transits by `ADDR_RD` $\rightarrow$ `ADDR_WR`. A more detailed analysis on the delay of the state `IDLE` for prescribed parameters N, d is given in the next subsection.

In state `IDLE`, the address is invalid since the purpose of `IDLE` is to wait for the correct results from the butterfly computing module and thus does not need the address signal to interact with memory. The address pattern used in state `ADDR_RD` is identical to that used in `ADDR_WR`: our butterfly processor is fully pipelined and, therefore, whenever it reads some data from some specific address in state `ADDR_RD`, it must write back to the same location later in state `ADDR_WR`.

Input: the number of NTT points N and the number of butterfly processors d
Output: memory address `memory_addra` and `memory_addrb` for butterfly computation

```
1 Precompute blk and dist
2 for roundi ← 0 to logN − 1 do
3     dist ← dist[roundi]
4     for blk ← 0 to blk[i] − 1 do
5         base ← N/(d·blk[i]) · blk
6         for cnt← 0 to N/(2d·blk[i]) − 1 do
7             memory_addra ← base + cnt
8             memory_addrb ← base + cnt + dist
```

Algorithm 11: Memory address generation for butterfly computation

2.7 Microbench Implementations

Timing analysis The main states we used are `ADDR_RD` and `ADDR_WR` which are used for memory read and memory write, respectively. If the delay of butterfly computation is longer than that of `ADDR_RD`, then an auxillary state called `IDLE` is inserted in between because the node processor cannot write valid data back to memory until the butterfly unit outputs the NTT results. Precisely speaking, if $\frac{N}{2d} - T_{\texttt{ADDR\_RD}} + 1 < T_{\texttt{butterfly}}$, then `IDLE` with delay $T_{\texttt{IDLE}} = T_{\texttt{butterfly}} - \frac{N}{2d} + T_{\texttt{ADDR\_RD}} - 1$ is required. The delay for the state `ADDR_RD` and the state `ADDR_WR` are $\frac{N}{2d}$ respectively, *i.e.*, $T_{\texttt{ADDR\_RD}} = T_{\texttt{ADDR\_WR}} = \frac{N}{2d}$. In summary, the total delay for the hypercube NTT with d processors is:

$$logN \cdot \left(\frac{N}{d} + max(T_{\texttt{IDLE}}, 0) \right)$$

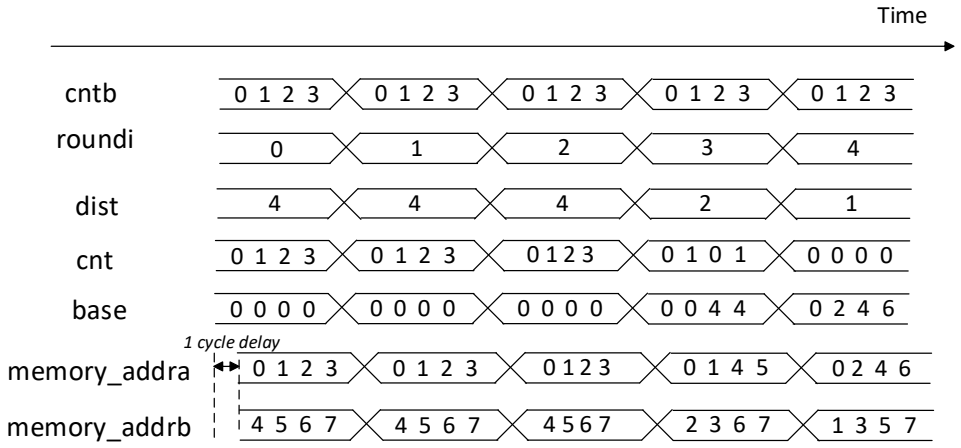


Figure 6: Illustrative timing diagram for memory address generation in line with Alg. 11($N = 32, d = 4$)

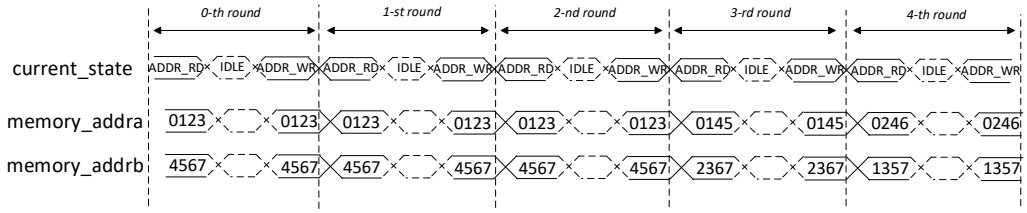


Figure 7: Top-level timing diagram for hypercube NTT in 5 rounds($N = 32, d = 4$)

Table 5: Performance of the configurable hypercube NTT hardware for FHEW-like FHE schemes on Xilinx Artix-7 FPGA

Instance	# of processors	freq	cycle	CLB/LUT/Reg	memory	DSPs
$N = 1024, q \approx 2^{32}$	2	100	5120	451/2309/1756	3	30
	4	100	2560	760/3581/2940	6	60
	8	100	1280	1402/4238/5480	12	120
	16	100	640	2174/10669/10591	24	240
	32	100	430	3937/19250/18738	48	480
	64	80	350	7835/39009/37060	96	960

In our concrete experiment, $T_{\texttt{ADDR\_RD}}$ set to 1 and $T_{\texttt{butterfly}}$ set to 27. Therefore the total delay for the hypercube NTT is further simplified to $logN \cdot (\frac{N}{d} + max(27 - \frac{N}{2d}, 0))$.

Experimental data The proposed design is implemented on Xilinx Zynq UltraScale+ ZCU106 evaluation board using Vivado 2018.1. The number of NTT points is set to 1024, a typical value used in FHE schemes. The number of NTT processors is configured to 2,4,8,16,32, and 64 to fully demonstrate the scalability of our hypercube NTT design. It is worth mentioning that our implementation follows the parameterized design approach, *i.e.*, our NTT hardware can be customized and auto-generated on the fly from a script file by inputting core parameters of hypercube NTT, for example, N and d. The experimental results are collected in Table. 5. As the parameter d increases, the clock frequency is rather stable around 100 MHz, which indicates the hypercube memory swapping strategy is successful to maintain a good critical path delay. If the number of processors is smaller than 32, the cycle delay equals to $logN \cdot \frac{N}{d}$ and thus the increase of d reduces significantly the cycle delay: for example, doubling d suggests cycle delay reduced by half. As the number of processors gets even bigger (≥ 32), the `IDLE` state is inserted and the cycle

delay equals to $logN \cdot (\frac{N}{2d} + 27)$ for which the performance boost by increasing d is rather marginal. For this case, we can optimize the performance of butterfly computation (more concretely, modular reduction) to further improve it. However, we do not push the limit on this direction which is not the focus in this paper.

2.8 System Integration on Xilinx MPSOC platform

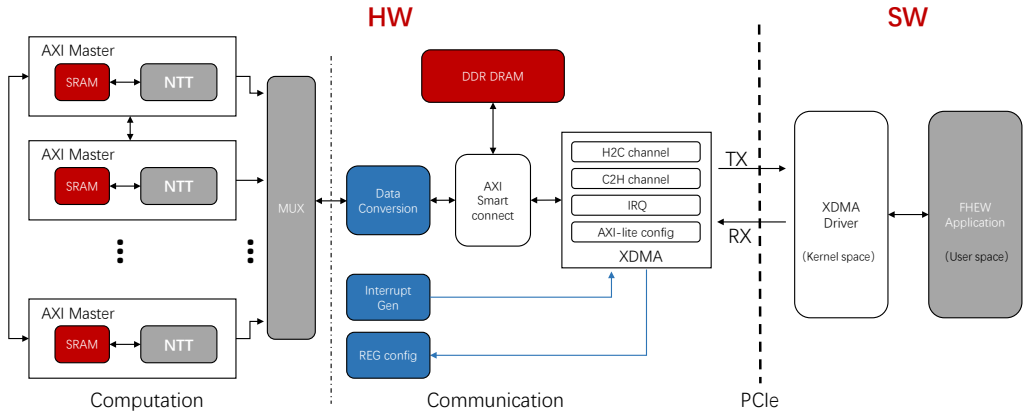


Figure 8: A software-hardware co-design for FHEW where the NTT module is implemented in PL logic and the software runs in PS logic

We also conduct an FPGA implementation experiment for the entire FHE bootstrapping where the NTT module is implemented in PL logic and the FHEW software (written in C++) runs on ARM PS logic. The target hardware platform is Xilinx Versal VMK180 development board. Figure 8 illustrates how a software-hardware co-design for a NTT-accelerated FHEW is built. The integration between FHEW software and NTT hardware is achieved via Xilinx's XDMA core. As a DMA, the core can be used for high performance block data movement between the PCIe address space and the AXI address space using the provided character driver [Inc23]. In this architecture, the NTT hardware is divided into computational and communication segments. The NTT computation module is composed of several NTT computation nodes that execute distinct NTT calculations. The NTT communication module is responsible for data exchange with the software.

In the specific process of software-hardware collaboration, in order to execute a single NTT hardware computation, the host needs to transmit $1,024$ data segments for calculation to the FPGA, exemplified by an NTT configuration of $N = 1024$ points and $q < 2^{32}$. The host sets up buffer space in system memory and creates descriptors that the DMA engine use to move the data. When the hardware DMA core receives the data, it stores the data into the DDR memory and informs the NTT computing module that the data is ready. The data conversion module subsequently reads the DDR data, converts it to the format required by the NTT computing module, and forwards it for processing. At this time, the NTT computation module starts to calculate. The calculation speed varies with different numbers of processors. After the computation is completed, the output data will be written back to DDR, and the processing is completed through the interrupt reporting host. Consequently, the DMA engine relocates the data from the DDR back to the host, completing an NTT calculation process.

Performance The performance of the NTT-accelerated FHEW solution is presented in Table 6. We configured the FHEW hardware with two different NTT setups: one featuring two NTT processor cores and the other with four NTT processor cores. While there is a noticeable speed improvement compared to the pure software solution, the advantage

Table 6: Performance of FHEW-like FHE schemes with and without NTT hardware module on Xilinx Artix-7 FPGA

Instance		Computing Time	Freq	LUT/Reg	Memory	DSPs
Pure Software	Overall	1.3s		n.a.	n.a.	n.a.
	Excluding NTT	0.24s	2.5 GHz	n.a.	n.a.	n.a.
	NTT (9484 times)	1.06s		n.a.	n.a.	n.a.
NTT-accelerated FHEW (NTT of 2 processors)	Overall	—	n.a.	n.a.	3	30
	FHEW Software	—	2.5 GHz	n.a.	n.a.	n.a.
	NTT commputation	—	250 MHz	2377/1879	3	30
	NTT communication	—	2.5 GHz	123090/135107	142	6
NTT-accelerated FHEW (NTT of 4 processors)	Overall	1.2s	n.a.	n.a.	3	30
	FHEW Software	0.25s	2.5 GHz	n.a.	n.a.	n.a.
	NTT commputation	0.14s	250 MHz	3702/3191	6	60
	NTT communication	0.81s	2.5 GHz	123090/135107	142	6

is not particularly significant. It is apparent that the hardware NTT computation speed surpasses that of software, yet a substantial amount of time is consumed in NTT communication, involving the transfer of encrypted data between the NTT hardware and the FHEW software. In summary, the primary performance bottleneck for FHEW software-hardware integration stems from the extensive data movement between the NTT hardware and the FHEW software. Therefore, optimizing NTT communication to minimize data transfer is essential for significantly enhancing FHEW performance.

References

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[DM15] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*, pages 617–640. Springer, 2015.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 75–92. Springer, 2013.

[Inc23] Xilinx Inc. Dma/bridge subsystem for pci express pg195, 2023. https://docs.amd.com/r/en-US/pg195-pcie-dma/Feature-Summary.