

Memory Swapping based Number-Theoretic Transform for Fully Homomorphic Encryption

Jingwei Hu¹ and Unknown²

¹ Nanyang Technological University, Singapore, davidhu@ntu.edu.sg

² Unknown

Abstract. This document is a quick introduction to the L^AT_EX class for the IACR Transactions.

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

Contents

1 FHEW-like Fully Homomorphic Encryption Scheme	1
2 Number-theoretic transform with merged twiddle factors	1
2.1 Higher level description for NTT with merged twiddle factors	2
2.2 $\log_2 d$ -Dimensional Hypercube Multiprocessors	6
2.3 A Useful Equivalent Notation: PID Local M	7
2.4 First attempt: parallel in-place FFTs without inter-processor permutations	8
2.5 Second attempt: Parallel NTTs with Inter-processor permutations	9
2.6 Butterfly Processor	13
2.7 Implementation Experiments	16
3 System Integration on Xilinx MPSOC platform	16
4 Conclusions	17

Introduction

1 FHEW-like Fully Homomorphic Encryption Scheme

2 Number-theoretic transform with merged twiddle factors

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 $\{w_i[\cdot]\}_{i=0,\dots,\log_2 N-1}$, is prepared for the i -th round of butterfly computation ($\log N$ rounds in total) as described in Alg. 2. 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 $i \in [N], j = 2^0, 2^1, \dots, 2^{\log N - 1}$.

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. 1 which is also referred 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 $\log N$ iterations (loop- i) at outermost, where each iteration computes one layer of butterfly computations. The i ($i = 0, \dots, \log_2 N - 1$)-th layer of butterfly computation always has $\frac{N}{2^i}$ butterflies. These butterflies are bundled into 2^i groups (recorded by the variable $NumberOfGroups$) and each group 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. 1 is depicted in Fig. 1a when $N = 8$. The inputs are $a[0], \dots, a[7]$ 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. 1), 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 outputs from the NTT network is in bit-reversed order as $A[0], A[4], A[2], A[6], A[1], A[5], A[3], A[7]$.

```

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

1  $PairsInGroup \leftarrow N/2$ 
2  $NumOfGroups \leftarrow 1$ 
3  $Distance \leftarrow N/2$ 
4 for  $i \leftarrow 0$  to  $\log_2 N - 1$  do
5   for  $k \leftarrow 0$  to  $NumOfGroups - 1$  do
6      $JFirst \leftarrow 2 \cdot k \cdot PairsInGroup$ 
7      $JLast \leftarrow JFirst + PairsInGroup - 1$ 
8     for  $j \leftarrow JFirst$  to  $JLast$  do
9        $Temp \leftarrow w_i[k] * a[j + Distance]$ 
10       $a[j + Distance] \leftarrow a[j] - Temp$ 
11       $a[j] \leftarrow a[j] + Temp$ 
12    $PairsInGroup \leftarrow PairsInGroup/2$ 
13    $NumOfGroups \leftarrow NumOfGroups \cdot 2$ 
14    $Distance \leftarrow Distance/2$ 
15 return  $a[\cdot]$ 
```

Algorithm 1: Higher level description of NTT, a.k.a DIT_{NN → RN}

Next, we detail how to construct the twiddle factor LUT. Recall the NTT with

Input: a polynomial ring R_q , and NTT points N Output: Twiddle factors $\{w_i[\cdot]\}_{i=0,\dots,\log_2 N-1}$ used in Algorithm 1 1 $FirstPart \leftarrow [0 \dots 0]_2$ 2 $SecondPart \leftarrow [1 \dots 0]_2$ 3 for $i \leftarrow 0$ to $\log_2 N - 1$ do 4 for $j \leftarrow 0$ to $N - 1$ do 5 $[j_{\log_2 N-1}, \dots, j_0]_2 \leftarrow BinRepr(j)$ 6 $w_i[j] \leftarrow \phi^{FirstPart} \cdot \phi^{SecondPart}$ 7 $FirstPart \leftarrow RightRot(FirstPart, j_{\log_2 N-1-i})$ 8 $SecondPart \leftarrow SecondPart / 2$ 9 return $\{w_i[\cdot]\}_{i=0,\dots,\log_2 N-1}$

Algorithm 2: Construction of Twiddle Factor LUTs

pre-processing can be written together as a summation of N terms:

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

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

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

Now express A_i s into the first half A_i and the second half $A_{i+\frac{N}{2}}$ as follows:

$$\begin{aligned} A_i &= \sum_{j=0}^{\frac{N}{2}-1} a_{2j} \omega_{\frac{N}{2}}^{ij} \omega_N^j + \omega_N^i \omega_{2N} \sum_{j=0}^{\frac{N}{2}-1} a_{2j+1} \omega_{\frac{N}{2}}^{ij} \omega_N^j \bmod q \text{ for } i \in [0, \frac{N}{2}-1] \\ A_{i+\frac{N}{2}} &= \sum_{j=0}^{\frac{N}{2}-1} a_{2j} \omega_{\frac{N}{2}}^{ij} \omega_N^j - \omega_N^i \omega_{2N} \sum_{j=0}^{\frac{N}{2}-1} a_{2j+1} \omega_{\frac{N}{2}}^{ij} \omega_N^j \bmod q \end{aligned}$$

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 $\{a_{2j+1}\}_{j \in [\frac{N}{2}]}$) defined by

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

Therefore, the equation above is rewritten in a more compact form:

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

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 $j = j_{n-1}j_{n-2}\cdots j_1j_0$
$i = 0$	$\omega_N^{\overbrace{00\cdots 00}^{n-1 \text{ bits}}} \cdot \omega_{2N}^{\overbrace{10\cdots 00}^n}$
$i = 0$	$\omega_N^{\overbrace{j_{n-1}0\cdots 00}^{n-1 \text{ bits}}} \cdot \omega_{2N}^{\overbrace{01\cdots 00}^n}$
\vdots	\vdots
$i = n - 2$	$\omega_N^{\overbrace{j_2j_3\cdots 00}^{n-1 \text{ bits}}} \cdot \omega_{2N}^{\overbrace{00\cdots 10}^n}$
$i = n - 1$	$\omega_N^{\overbrace{j_1j_2\cdots j_{n-2}j_{n-1}}^{n-1 \text{ bits}}} \cdot \omega_{2N}^{\overbrace{00\cdots 01}^n}$

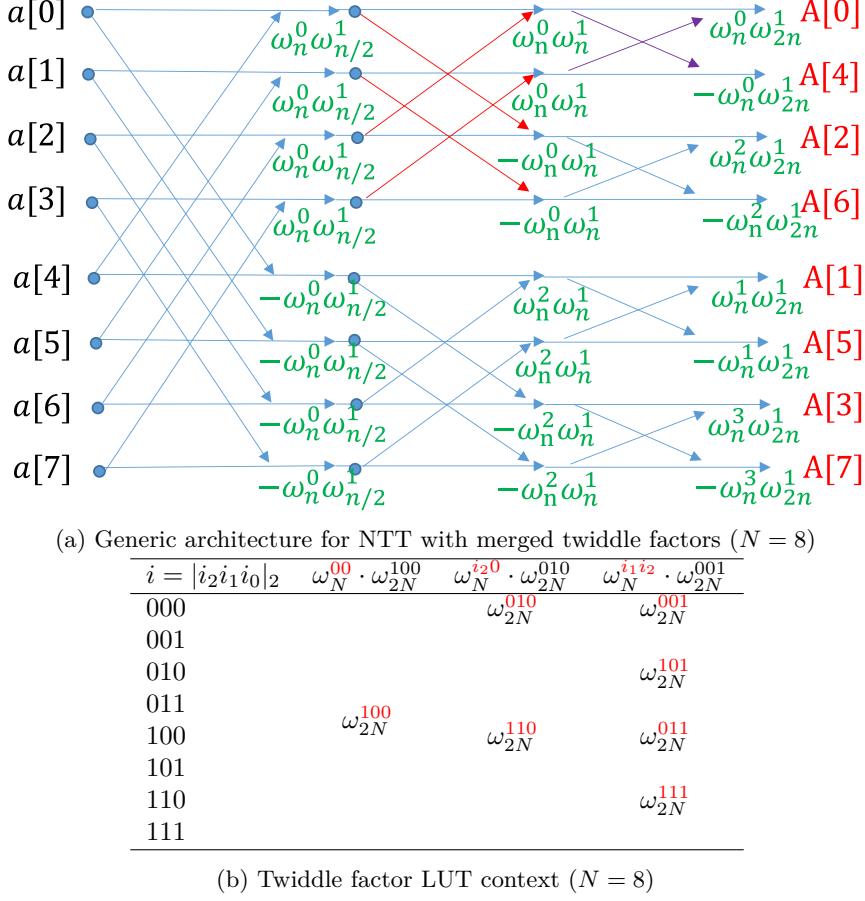
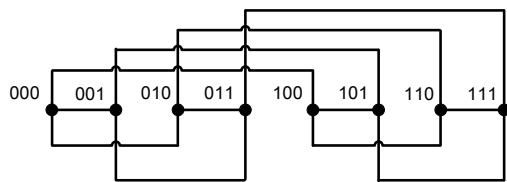
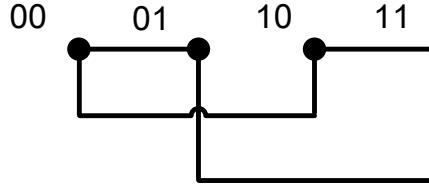
The key observation for the equation above is that $Y_i^{(n)}$ and $Z_i^{(n)}$ has a recursive structure: for example, $Y_i^{(n)}$ and $Z_i^{(n)}$ are computed from a butterfly computation of $Y_i^{(n-1)}$ and $Z_i^{(n-1)}$, $Y_i^{(n-1)}$ and $Z_i^{(n-1)}$ are computed from a butterfly computation of $Y_i^{(n-2)}$ 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 twiddle 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 no-

tations here and do not repeat the proof. It has the form $\omega_N^{\overbrace{00\cdots 00}^{n-1 \text{ bits}}}, \omega_N^{\overbrace{j_{n-1}0\cdots 00}^{n-1 \text{ bits}}}, \dots, \omega_N^{\overbrace{j_1j_2\cdots j_{n-2}j_{n-1}}^{n-1 \text{ bits}}}$, for $i = 0, 1, \dots, n - 1$, respectively. The second part of the merged twiddle factor is ω_{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 $\omega_{2\cdot 2}^1, \omega_{2\cdot 4}^1, \dots, \omega_{2N}^1$ for $i = 0, 1, \dots, n - 1$. Further to unify the notation, the second part is rewritten in binary form as $\omega_{2N}^{\overbrace{10\cdots 00}^n}, \omega_{2N}^{\overbrace{01\cdots 00}^n}, \dots, \omega_{2N}^{\overbrace{00\cdots 01}^n}$ for $i = 0, 1, \dots, n - 1$.

Alg. 2 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 first impression on Alg. 2 might be that the size of twiddle factor LUTs is about $\mathcal{O}(N \log N)$: It has $\log N$ rounds and each round consumes $N/2$ twiddle factor for the $N/2$ pairs of input points. 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 $w_1[N/2]$, $w_2[\cdot\cdot\cdot]$ 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 $\{w_i[\cdot]\}$ equal to

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

Figure 1: DIT instance with $N = 8$ Figure 2: Hypercubes of Dimension $\log_2 d = 2$ and $\log_2 d = 3$

```

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

```

Algorithm 3: Construction of the $\log_2 d$ -dimensional hypercube

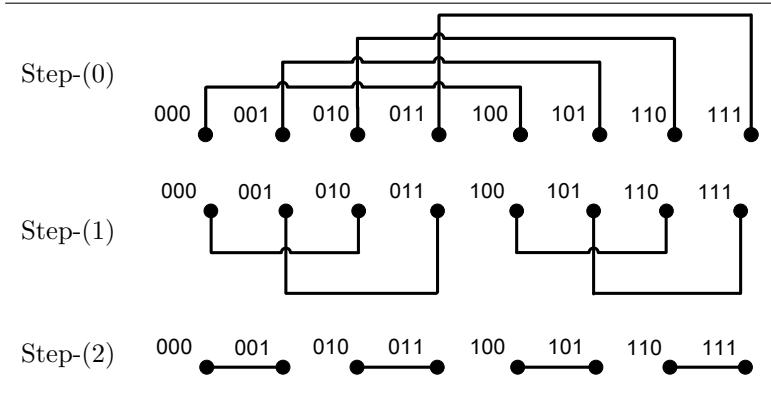
```

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

```

Algorithm 4: Subcube-doubling communication in $\log_2 d$ -dimensional hypercube

Table 2: Subcube-doubling communication in 3-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 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 its 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, \dots, P_7 labeled as its 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, P_2 and P_4 , P_1 is only connected to P_0, P_3 and P_5 .

Alg. 3 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 $[i_{\log_2 d-1}, \dots, i_0]_2$, and connects those processors P_j whose index $j = [j_{\log_2 d-1}, \dots, j_0]_2$ 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. 3 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. 4: In step-(k), each node processor P_i with index $i = [i_{\log_2 d-1}, \dots, i_0]_2$ 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. 4) 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 $P_{[\bar{i}_2, i_1, i_0]_2}$ 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 $P_{[i_2, \bar{i}_1, i_0]_2}$ which differs at i_1 ; Finally, in step-(2), processor $P_{[i_2, i_1, i_0]_2}$ connects processor $P_{[i_2, i_1, \bar{i}_0]_2}$ which differs at i_0 .

2.3 A Useful Equivalent Notation: |PID|Local M

Assume that N points are stored in the global array $a[\cdot] = \{a_{N-1}, \dots, 0\}$ or simplified as $a[\cdot] = \{a_i\}_{i=N-1, \dots, 0}$, 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 $\log N$ -bit integer $i = i_{\log N-1} \dots i_0$:

$$i_{\log N-1} \dots i_{k+1} | i_k \dots i_{k-\log d+1} | i_{k-\log d} \dots i_0$$

to indicate that consecutive $\log d$ bits $i_k \dots i_{k-\log d+1}$ are chosen to specify the data-to-processor allocation.

In general, since any $\log d$ 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 $\log N - \log d$ 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.

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

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

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 $M_{i_4 i_3 i_2 i_1 i_0}$	$P_{i_4 i_3} = P_{00}$	PID	Local $M_{i_4 i_3 i_2 i_1 i_0}$	$P_{i_4 i_3} = P_{01}$	PID	Local $M_{i_4 i_3 i_2 i_1 i_0}$	$P_{i_4 i_3} = P_{10}$	PID	Local $M_{i_4 i_3 i_2 i_1 i_0}$	$P_{i_4 i_3} = P_{11}$
	$a[m]$			$a[m]$			$a[m]$			$a[m]$	
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	$a[31]$				

$a[r]$, $r = i_4i_3|i_2i_1i_0 = 11|010_2 = 26$, meaning that $a[26] = a_{26}$ (the element a_{26} is located in $a[26]$) is allocated by processor $P_{i_4i_3} = P_{01}$.

On the other hand, when the input elements are stored in \mathbf{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:

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

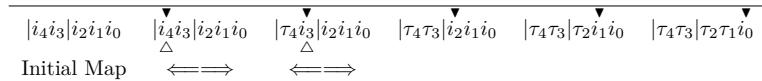
Table 4: Local data in processor $P_{i_0 i_1}$ (bit reversed) expressed in terms of global array element $a[r] = a_m, r = i_0 i_1 i_2 i_3 i_4, m = i_4 i_3 i_2 i_1 i_0$ for the notation $i_0 i_1 | i_2 i_3 i_4$

$ PID $	$Local\ M$	$P_{i_0 i_1} = P_{00}$	$ PID $	$Local\ M$	$P_{i_0 i_1} = P_{01}$	$ PID $	$Local\ M$	$P_{i_0 i_1} = P_{10}$	$ PID $	$Local\ M$	$P_{i_0 i_1} = P_{11}$
$i_0 i_1 i_2 i_3 i_4$	$a[r]$		$i_0 i_1 i_2 i_3 i_4$	$a[r]$		$i_0 i_1 i_2 i_3 i_4$	$a[r]$		$i_0 i_1 i_2 i_3 i_4$	$a[r]$	
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	$a[31]$	

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 $i_0i_1|i_2i_3i_4$. For instance, to locate $a_m = a_{26}$, one writes down $m = 26 = 11010_2 = i_4i_3i_2i_1i_0$, from which one knows that a_{26} is stored in $a[r]$, $r = i_0i_1|i_2i_3i_4 = 01|011_2 = 11$, meaning that $a[11] = a_{26}$ (the element a_{26} is located in $a[11]$) is allocated by processor $P_{i_0i_1} = P_{01}$.

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

Consider the $DIT_{NN \rightarrow RN}$ algorithm (Alg. 1) and use the cyclic block mapping introduced in the last subsection. For $N = 32, d = 4$, the computation is depicted below:



The initial map indicates that the processor P_k initially holds the elements $a[8k], \dots, a[8k+7]$ for $k = 0, 1, 2, 3$. The shorthand notation previously used for sequential NTT is augmented by two additional symbols. The double-headed arrow \iff 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 processors 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 $i = 0, \dots, 7$, 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 $i = 8, \dots, 15$; 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 $i = 0, \dots, 7$, 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 $i = 16, \dots, 23$. The last three steps (step-2,3,4) do not require data swaps since all elements needed for butterfly computation are already within the processor: 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 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 Δ and ∇ , the complete parallel algorithm corresponding to DIT_{NR} NTT is represented below for the $N = 32$ example.

$ i_4i_3 i_2i_1i_0$	$ i_2i_3 \overset{\nabla}{\Delta}i_4i_1i_0$	$ i_2\tau_4 \overset{\nabla}{\Delta}i_3i_1i_0$	$ \tau_3\tau_4 \overset{\nabla}{\Delta}i_2i_1i_0$	$ \tau_3\tau_4 \tau_2i_1i_0$	$ \tau_3\tau_4 \tau_2\tau_1i_0$
Initial Map	\iff				

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 $14 = 01|110_2$.

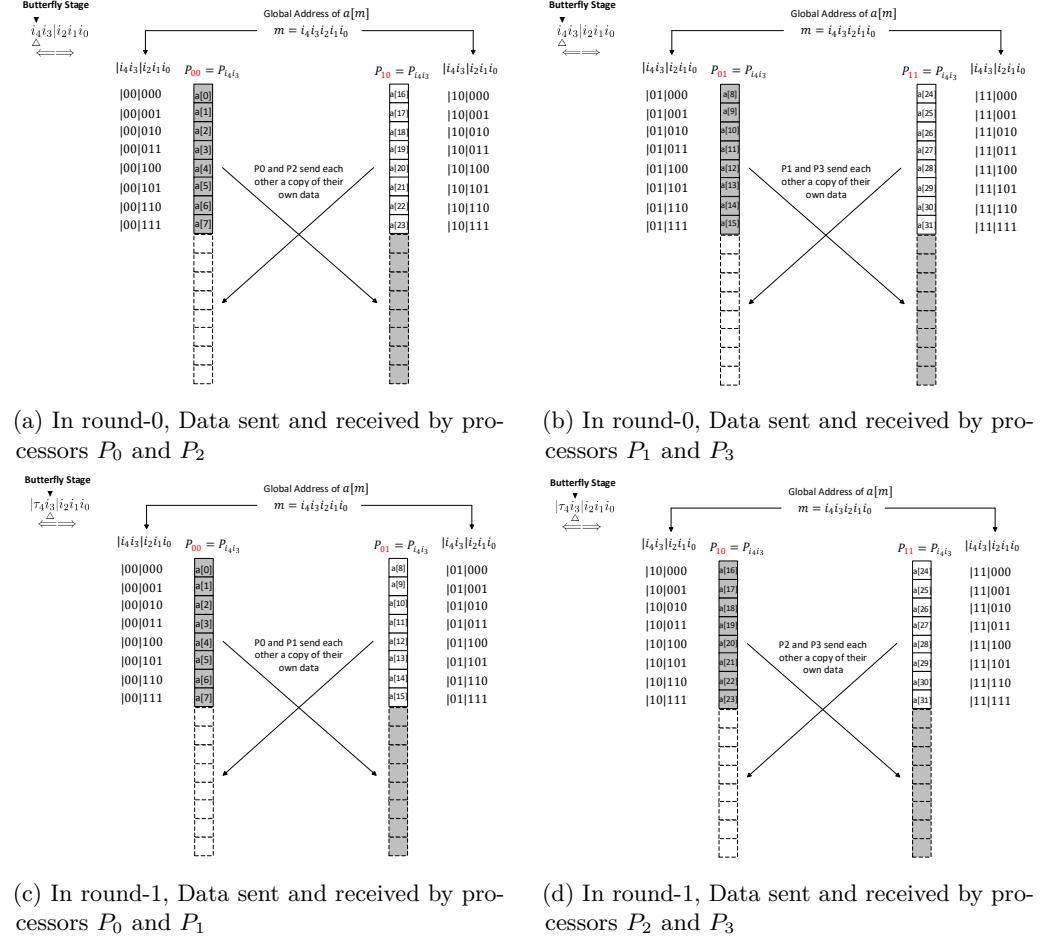


Figure 3: An illustrative example for parallelizing in-place NTT($N = 32, d = 4$) without inter-processor permutations

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 of his local array with the first half of P_3 's local array as depicted in Fig. 4a. The symbols \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 \triangle

been permuted from Local M to the PID. In our case, the notation $|i_2i_3|_{\triangle}^{i_4}i_1i_0$ 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 perform the first round of butterfly computation. The symbol \blacktriangledown is used to indicate the pairs of elements for the

butterfly computation in each processor, *i.e.*, $|i_2 i_3|_{\Delta}^{\blacktriangledown} i_4 i_1 i_0$ means $A[i_2 i_3 0 i_1 i_0]$ should pair with $A[i_2 i_3 1 i_1 i_0]$ to complete elementary butterfly unit computation for $P_{i_2 i_3}$. 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, $|i_2 i_3|_{\Delta}^{\blacktriangledown} i_4 i_1 i_0$ represents the first round where no τ indices exist; $|i_2 \tau_4|_{\Delta}^{\blacktriangledown} i_3 i_1 i_0$ 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 depicted in Fig. 4b. a similar notation $|i_2 \tau_4|_{\Delta}^{\blacktriangledown} i_3 i_1 i_0$ 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 $i_2 i_3 | i_4 i_1 i_0$ to $i_2 i_4 | i_3 i_1 i_0$. 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, $A[i_2 i_4 0 i_1 i_0]$ pairs with $A[i_2 i_4 1 i_1 i_0]$ to complete elementary butterfly unit computation for $P_{i_2 i_4}$ 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 depicted in Fig. 4c. This swapping pattern is captured in the notation $|\tau_3 \tau_4|_{\Delta}^{\blacktriangledown} i_2 i_1 i_0$ indicating the previous $i_2 i_4 | i_3 i_1 i_0$ is changed to $i_3 i_4 | i_2 i_1 i_0$ due to the Δ 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 $A[i_3 i_4 0 i_1 i_0]$ pairs with $A[i_3 i_4 1 i_1 i_0]$ to complete elementary butterfly unit computation for $P_{i_3 i_4}$ 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, $A[i_3 i_4 i_2 0 i_0]$ pairs with $A[i_3 i_4 i_2 1 i_0]$ for $P_{i_3 i_4}$, and in the fifth round, $A[i_3 i_4 i_2 i_1 0]$ pairs with $A[i_3 i_4 i_2 i_1 1]$ for $P_{i_3 i_4}$. 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 $A_{i_0 i_1 i_2 i_3 i_4}$, which overwrites the data in $a[i_4 i_3 i_2 i_1 i_0]$, is finally contained in $A[i_2 i_1 i_0]$ in $P_{i_3 i_4}$. 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 $a[i_4 i_3 i_2 i_1 i_0] = a_{i_4 i_3 i_2 i_1 i_0}^{(5)} = A_{i_0 i_1 i_2 i_3 i_4}$ where $A_{i_0 i_1 i_2 i_3 i_4} = \sum_j a_j (\omega_N^{i_0 i_1 i_2 i_3 i_4} \cdot \omega_{2N}^1)^j$
- The final mapping $|\tau_3 \tau_4| \tau_2 \tau_1 \tau_0$ indicates that the final content in $a[i_4 i_3 i_2 i_1 i_0]$ 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 $|i_2 i_3| \overset{\nabla}{\triangle} i_4 i_1 i_0$ is shown in its entirety in Fig. 4a and 4b.

```

Input: a polynomial ring  $R_q$ , and NTT points  $N$ , input  $\mathbf{a} = (a[0], \dots, a[N - 1])$ 
Output:  $NTT(\mathbf{a}) = \mathbf{A} = (A[0], \dots, A[N - 1])$ 
1 Initialize the hypercube connections between  $d$  processors as described in Alg. 3
2 Initialize the merged twiddle factor look-up table  $\{w_i\}$  as described in Alg. 2
3 /*arrange the data array  $a[\cdot]$  in natural order*/
4 Initialize the data  $a[i_{log_2 N - log_2 d} \dots i_1 i_0]$  in  $P_{i_{log_2 N - 1} \dots i_{log_2 N - log_2 d}}$  with
    $a[i_{log_2 N - 1} \dots i_1 i_0]$  for all  $i_{log_2 N - 1}, \dots, i_0$ 
5 /*perform the first  $log_2 d + 1$  round of computations where inter-processor data
   swapping is required*/
6 for  $j \leftarrow 0$  to  $log_2 d$  do
7   if  $j \neq log_2 d$  then
8     exchange the first half of data in  $P_{i_{log_2 N - 1} \dots i_{log_2 N - 1-j} \dots i_{log_2 N - log_2 d}}$  w.r.t.
        $i_{log_2 N - 1-j} = 0$  with the second half of data in
        $P_{i_{log_2 N - 1} \dots i_{log_2 N - 1-j} \dots i_{log_2 N - log_2 d}}$  w.r.t.  $i_{log_2 N - 1-j} = 1$ 
9   else
10    exchange the first half of data in  $P_{i_{log_2 N - 1} \dots i_{log_2 N - 1-j} \dots i_{log_2 N - log_2 d}}$  w.r.t.
        $i_{log_2 N - 1} = 0$  with the second half of data in
        $P_{i_{log_2 N - 1} \dots i_{log_2 N - 1-j} \dots i_{log_2 N - log_2 d}}$  w.r.t.  $i_{log_2 N - 1} = 1$ 
11   perform within each processor  $P_{i_{log_2 N - 1} \dots i_{log_2 N - 1-j} \dots i_{log_2 N - log_2 d}}$  the  $\frac{N}{2d}$  butterfly
       computations (round- $j$  butterfly)
12 /*perform the first  $log_2 N - log_2 d - 1$  round of computations where inter-processor
   data swapping is not required*/
13 for  $j \leftarrow log_2 d + 1$  to  $log_2 N - 1$  do
14   perform within each processor  $P_{i_{log_2 N - 1} \dots i_{log_2 N - 1-j} \dots i_{log_2 N - log_2 d}}$  the  $\frac{N}{2d}$  butterfly
       computations (round- $j$  butterfly)
15 return the data in all  $d$  processors as  $\mathbf{A}$ 
```

Algorithm 5: 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 $log_2 d + 1$ rounds of butterfly computations, memory swapping occurs and each butterfly processor utilizes only 1 twiddle factor; in the next $log_2 N - log_2 d - 1$ rounds, no memory swapping occurs and the number of twiddle factors utilized in each butterfly processor increases exponentially (starting with 2). Therefore, the total number of twiddle factors (the depth of twiddle factor LUT) in each processor is:

$$\sum_{i=1}^{log_2 d + 1} 1 + \sum_{i=1}^{log_2 N - log_2 d - 1} 2^i = \frac{N}{d} + log_2 d - 1$$

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

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 $\frac{N}{d} + \log d - 1$ 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. 6), one modular adder (apply Alg. 7), and one modular subtractor (apply Alg. 8). 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:  $a \cdot b \in \mathbb{Z}_q$ 
1 Precompute an integer  $k = \lceil \log_2 q \rceil$ 
2 Precompute an integer  $r = \lfloor \frac{4^k}{q} \rfloor$ 
3 Calculate  $x = a \cdot b$ 
4 Calculate  $t = x - \lfloor \frac{xr}{4^k} \rfloor \cdot q$ 
5 if  $t < q$  then
6   | return  $t$ 
7 else
8   | return  $t - q$ 
```

Algorithm 6: Barret-Reduction based Modular Multiplication

```

Input: two integers  $a$  and  $b$  over  $\mathbb{Z}_q$ 
Output:  $a + b \in \mathbb{Z}_q$ 
1 Calculate  $t = a + b$ 
2 if  $t < q$  then
3   | return  $t$ 
4 else
5   | return  $t - q$ 
```

Algorithm 7: Modular Addition

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

Input: two integers a and b over \mathbb{Z}_q

Output: $a - b \in \mathbb{Z}_q$

```

1 Calculate  $t = a - b$ 
2 if  $t \geq 0$  then
3   return  $t$ 
4 else
5   return  $t + q$ 
```

Algorithm 8: Modular Subtraction

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_addr` 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_addr` 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_addr` and `memory_addrb` s.t. `memory_addrb=memory_addr+dist`
- `cnt`: counter register, used to indicate the incremental offset value for generating `memory_addr`
- `base`: the (basis) starting address for `memory_addr` 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. 1, 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. 1, and indicates the distance between `memory_addr` and `memory_addrb` in every round of butterfly calculation and has $\log_2 N$ elements. The construction `blk` goes like this: The first $\log d$ elements are always 1; starting from the $(\log d + 1)$ -th element down to the last one, *i.e.* the last $\log N - \log d$ elements formulate a geometric sequence with initial value 1 and common ratio 2. The construction `dist` goes like this: The first $\log d$ elements are always $\frac{N}{2d}$; Then the last $\log N - \log d$ 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 `blk` = {1, 1, 1, 2, 4} and `dist` = {4, 4, 4, 2, 1}.

The generation of `memory_addr` and `memory_addrb` in Fig. 5 is formally described in Alg. 9. 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 $\log N = 5$ 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. 9) takes one clock cycle delay and thus the result appearing in `memory_addr` and `memory_addrb` is delayed by one clock cycle as shown in Fig. 6. The sequence of `memory_addr` and `memory_addrb` can be interpreted as follows: In the first clock cycle of phase-0, `memory_addr` 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_addr` outputs 1 and `memory_addrb` outputs 5, and so on so forth. Finally, in the first clock cycle of phase-4, `memory_addr` outputs 0 and `memory_addrb` outputs 1; in the second clock cycle, `memory_addr` 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 $\log N = 5$ 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_addr` and `memory_addrb` for butterfly computation

```

1 Precompute blk and dist
2 for roundi  $\leftarrow 0$  to  $\log N - 1$  do
3   | dist  $\leftarrow \text{dist}[\text{roundi}]$ 
4   | for blk  $\leftarrow 0$  to  $\text{blk}[i] - 1$  do
5     |   | base  $\leftarrow \frac{N}{d \cdot \text{blk}[i]} \cdot \text{blk}$ 
6     |   | for cnt  $\leftarrow 0$  to  $\frac{N}{2d \cdot \text{blk}[i]} - 1$  do
7       |     |   memory_addr  $\leftarrow \text{base} + \text{cnt}$ 
8       |     |   memory_addrb  $\leftarrow \text{base} + \text{cnt} + \text{dist}$ 

```

Algorithm 9: Memory address generation for butterfly computation

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

2.7 Implementation Experiments

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 auxiliary 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 $\frac{N}{2d} - T_{\text{ADDR_RD}} + 1 < T_{\text{butterfly}}$, then `IDLE` with delay $T_{\text{IDLE}} = T_{\text{butterfly}} - \frac{N}{2d} + T_{\text{ADDR_RD}} - 1$ is required. The delay for the state `ADDR_RD` and the state `ADDR_WR` are $\frac{N}{2d}$ respectively, *i.e.*, $T_{\text{ADDR_RD}} = T_{\text{ADDR_WR}} = \frac{N}{2d}$. In summary, the total delay for the hypercube NTT with d processors is:

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

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

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 $\log N \cdot \frac{N}{d}$ 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 $\log N \cdot (\frac{N}{2d} + 27)$ 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.

3 System Integration on Xilinx MPSOC platform

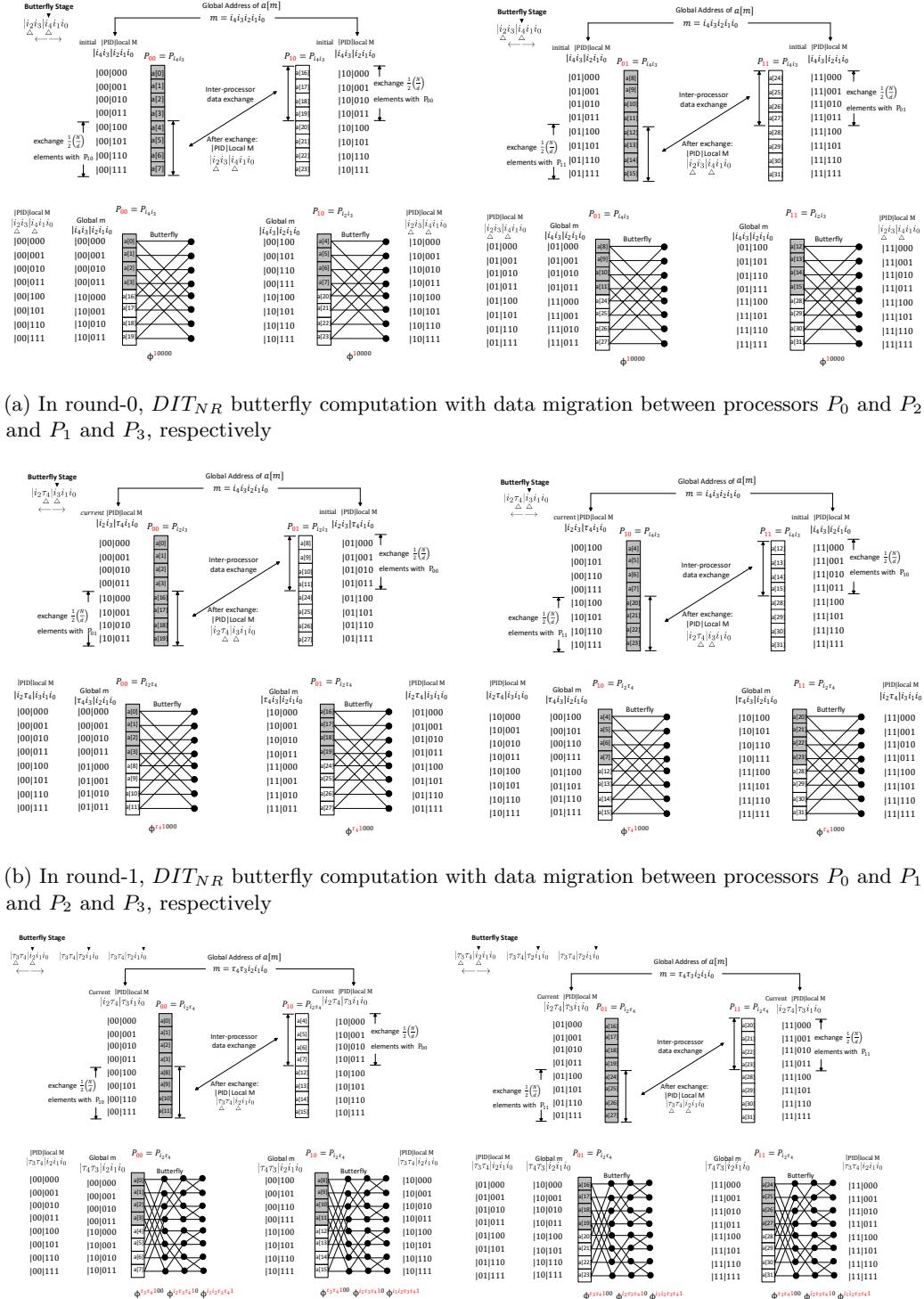
We also conduct FPGA experiment where the NTT module is implemented in PL logic and the FHEW software (written in C++) runs on ARM PS logic. The target hardware platform is Xilinx Versal VMK180 development board. Each NTT node processor (together with its internal block memory) is implemented as a AXI master device. NTT node processors communicate with each other for memory exchange through MUX interface. The content in internal block memory of the node processor is updated by the PS DDR memory through AXI DMA interface. It is worth noting that the block memory update is frequent: whenever

an NTT computation is required, the block memory content must be refreshed. Typically, a single FHEW bootstrapping operation involves thousands of NTT computations.

Performance

4 Conclusions

References



(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

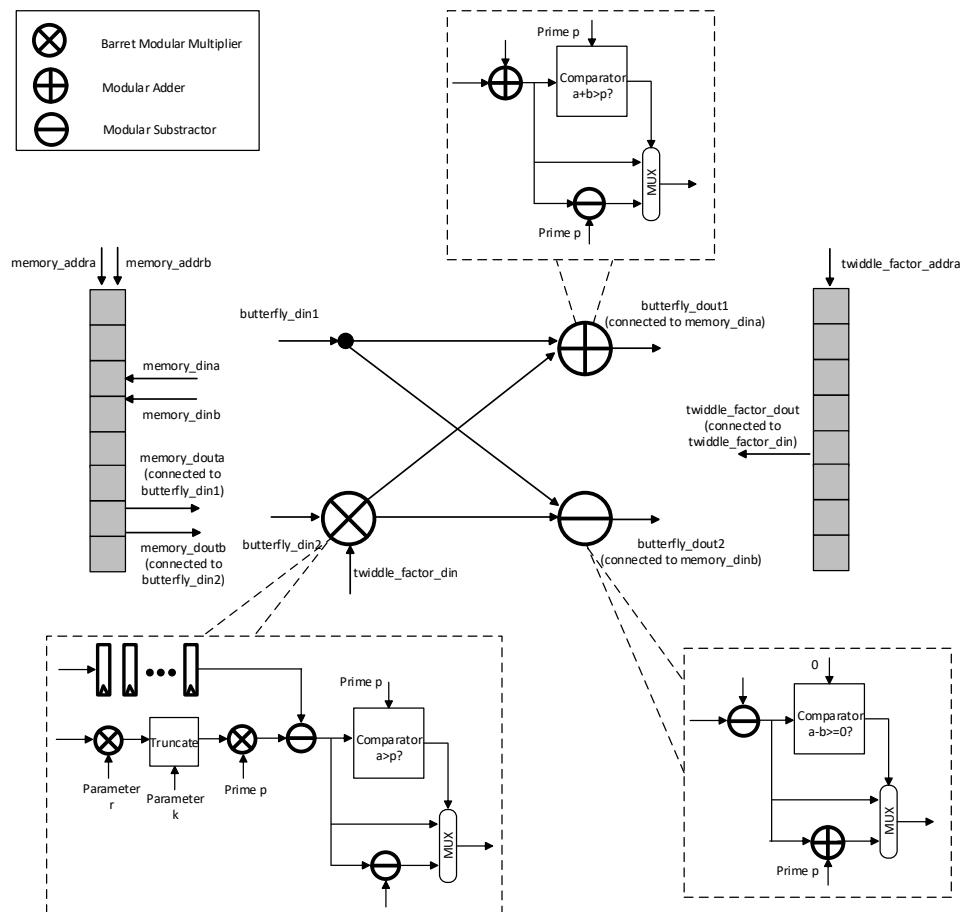


Figure 5: Internal structure of butterfly processor

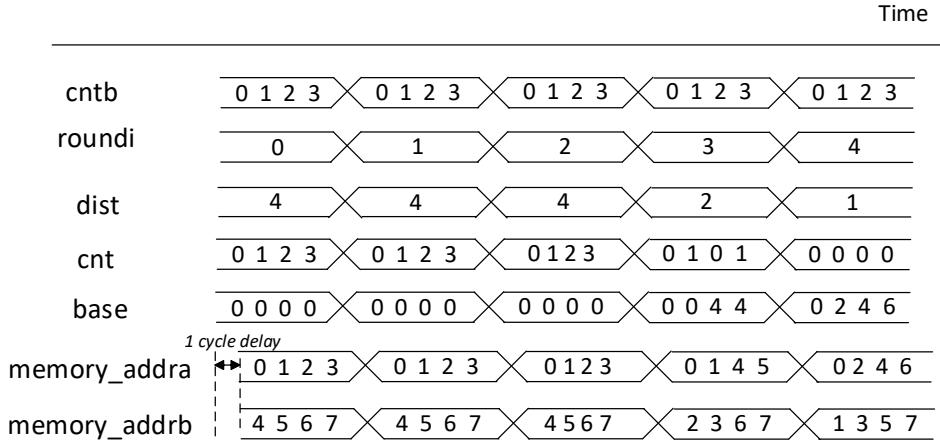


Figure 6: Illustrative timing diagram for memory address generation in line with Alg. 9($N = 32, d = 4$)

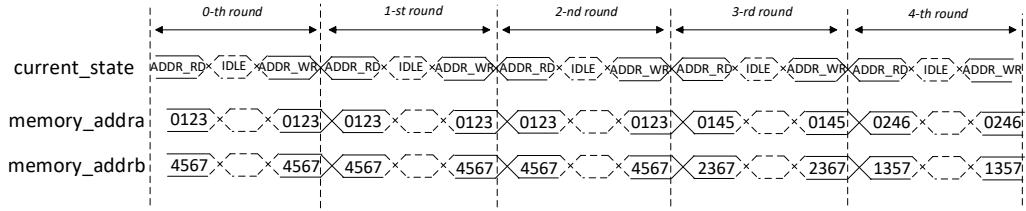


Figure 7: Top-level timing diagram for hypercube NTT in 5 rounds($N = 32, d = 4$)

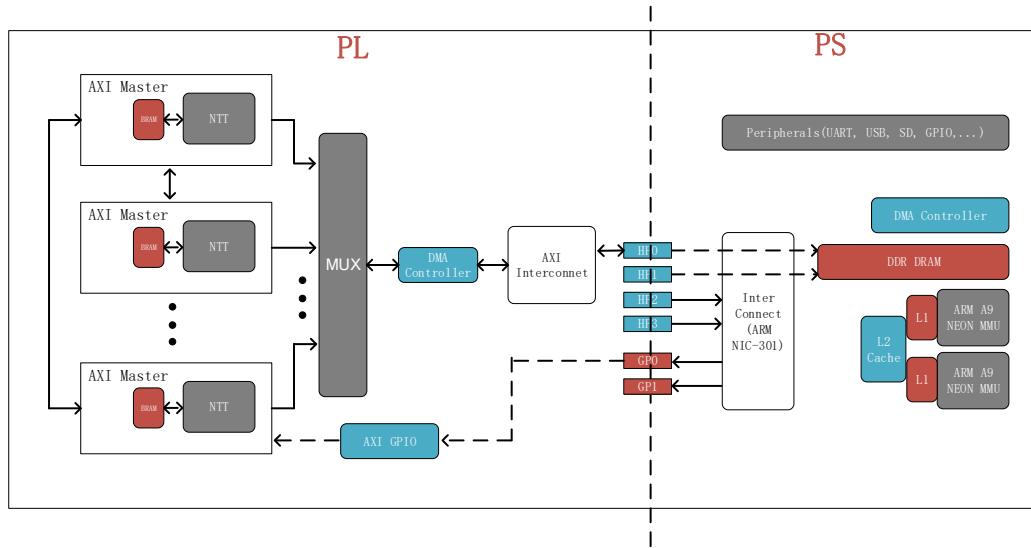


Figure 8: A software-hardware co-design for FHEW where the NTT module is implemented in PL logic and the software runs in PS logic