2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 979-8-3503-7243-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/FCCM60383.2024.00010

Bandwidth Efficient Homomorphic Encrypted Discrete Fourier Transform Acceleration on FPGA

Zhihan Xu*, Yang Yang*, Rajgopal Kannan† and Viktor K. Prasanna*
* Department of Electrical and Computer Engineering, University of Southern California
† DEVCOM Army Research Office
Email: {zhihanxu, yyang172, prasanna}@usc.edu, rajgopal.kannan.civ@army.mil

***Abstract*—Fully Homomorphic Encryption (FHE) plays an important role in privacy-preserving computation on the cloud. It allows computations on encrypted data without decryption. Bootstrapping is a fundamental operation in FHE, enabling an unlimited number of homomorphic encrypted computations, but at a significant time cost. A major bootstrapping component, the Homomorphic Encrypted Discrete Fourier Transform (HE DFT), is particularly time-consuming and requires the transfer of a large amount of data from external memory.**

In this paper, we propose a bandwidth-efficient FPGA implementation of HE DFT. We design a cost model to evaluate the on-chip memory requirement and the off-chip data transfer overhead for HE DFT. Our analysis shows that prior approaches can lead to significant off-chip data transfers, which process the entire ciphertext between subroutines. To address DRAM transfer overhead, we propose LimbFlow, an optimized dataflow approach for HE DFT that enhances fine-grained data reuse by rearranging the processing order of ciphertext and merging several subroutines. Leveraging the LimbFlow, we develop an FPGA-based accelerator tailored for HE DFT. We evaluate the accelerator on AMD U280 FPGA across various sets of security parameters. Our accelerator achieves up to 4.90× and 1.98× speedup compared with the State-Of-The-Art (SOTA) GPU and FPGA implementations.

I. Introduction

Homomorphic Encryption (HE) provides a solution to privacy-sensitive applications deployed on the cloud. It allows computations on the encrypted data directly, ensuring that third-party cloud servers cannot access or infer any details from the underlying data. As HE operations progress, the noise term required for security accumulates in the ciphertext [1], [2]. A decryption error occurs if the noise is over a threshold. To mitigate this, bootstrapping is employed to effectively refresh the noise, allowing for an unlimited number of HE operations [3]. HE schemes with this mechanism are referred to as FHE. However, bootstrapping is the most time-consuming operation with a low arithmetic intensity (<1 Op/byte) and more expensive (>100×) in terms of both compute and memory requirements than other primitive HE operations [4]. A significant part of this computational burden arises from HE DFT and HE Inverse-DFT (HE IDFT)[1]. The DFT and IDFT operations are used to decode and encode the original message vector. In comparison, HE DFT and HE IDFT evaluate these transformations directly on the ciphertext with substantially increased complexity, severely affecting the bootstrapping efficiency in FHE [5].

The complex dataflow involving a variety of HE subroutines and extensive DRAM data transfers makes HE DFT operation a significant bottleneck in FHE. Benefiting from the configurable on-chip SRAM memory hierarchy and compute cores, FPGAs can be tailored to reduce data movement significantly. Prominent FPGA vendors also incorporate High Bandwidth Memory (HBM) to address the limitations of DRAM bandwidth [6], [7]. The latest FPGAs can provide up to 820 GB/s bandwidth and 32 GB storage capacities, which makes them highly suitable for boosting performance in memory-intensive tasks such as HE DFT [8].

The HE DFT operation can be expressed as the product of a $N \times N$ plaintext matrix and an encrypted ciphertext vector [9]. Given the large dimension of the inputs (e.g., 2^{16}) and the substantial coefficient size (e.g., thousand bits) for each input element, this type of operation is very costly in FHE [10]. As discussed in [11], it can be efficiently done by handling the matrix in the diagonal order and utilizing the SIMD computation. The $M \times V$ product can be achieved by multiplying each matrix diagonal with the circularly shifted ciphertext vector and accumulating the partial sums together. In [12], [13], the computation of the HE DFT is performed through the homomorphic evaluation of the Cooley-Tukey algorithm. This approach decomposes the DFT matrix into several sparse block diagonal matrices, a process known as matrix factorization. However, HE DFT still requires the transfer of substantial amounts of data from off-chip memory, including gigabytes of evaluation keys for ciphertext rotation and megabytes of plaintext matrices and ciphertexts, which create significant bandwidth demands.

Efforts to enhance bandwidth efficiency in HE DFT have produced various accelerators targeting bootstrapping performance. Previous SOTA works [14], [15], [16], have attempted to improve efficiency by amortizing specific operations across multiple rotations, a technique known as hoisting [17]. Despite these advancements, a critical gap in these designs is the lack of a detailed analysis of the bandwidth requirements specific to HE DFT. In addition, current dataflows in these designs still operate at the ciphertext level, which is relatively coarse and suffers from significant off-chip data transfer. Addressing this challenge is crucial to improving the overall bootstrapping efficiency and practicality of FHE systems.

[1]We use HE DFT to denote both HE DFT and HE IDFT in this paper.

2576-2621/24/$31.00 ©2024 IEEE
DOI 10.1109/FCCM60383.2024.00010

To overcome these challenges, we propose a bandwidth-efficient approach for implementing HE DFT on FPGAs. Firstly, we introduce a detailed cost model to analyze the memory requirements of the HE DFT. Guided by the modeling results, we propose a new HE DFT dataflow method named LimbFlow. LimbFlow rearranges the processing order and processes ciphertext in a fine-grained manner to achieve greater data reuse. We develop a novel FPGA-based accelerator coupled with the LimbFlow, specifically targeting the HE DFT operation for bootstrapping in FHE. Our design significantly improves the HE DFT efficiency with reduced bandwidth requirements. The key contributions of this work are as follows.

- We design a general cost model to quantitatively analyze the off-chip memory access and on-chip memory requirements for HE DFT. Given the input HE parameters, the model can calculate the total amount of off-chip data transfers and determine the required SRAM to facilitate ciphertext reuse across subroutines.
- We propose a novel dataflow named LimbFlow for HE DFT, which reorganizes the polynomial processing order and merges several subroutines. LimbFlow is designed to achieve fine-grained ciphertext reuse at the limb level, reducing off-chip data movement.
- We develop a bandwidth-efficient accelerator specifically tailored for HE DFT based on LimbFlow. Experimental results show that the accelerator achieves up to 4.90× and 1.98× speedup compared with the SOTA GPU and FPGA implementations.

II. Background

In this section, we briefly review the CKKS FHE scheme [10], explain the bootstrapping process, and analyze the HE DFT operation. In Table I, we summarize the parameters for CKKS used in this paper.

A. The CKKS FHE scheme

The CKKS scheme supports the encryption of complex fixed-point vectors. In CKKS, a message $\mathbf{m}$ is a vector in $\mathbb{C}^{N/2}$. The message vector is packed and encoded into a plaintext polynomial $\mathrm{P}_{\mathbf{m}}$ with degree $N-1$, and then encrypted into a ciphertext $[\![\mathbf{m}]\!] = (\mathbf{a}, \mathbf{b})$ in $\mathcal{R}_Q^2$, where $\mathbf{a}$ and $\mathbf{b}$ are a pair of elements in $\mathcal{R}_Q = \mathbb{Z}_Q[X]/(X^N+1)$. The ciphertext polynomial coefficients are integers modulo Q, and the degree is a power-of-two integer N, typically ranging from 2^{10} to 2^{16} [18]. The primitive operations of CKKS are shown in Table II. We refer [16] to the readers for more details.

The coefficient modulus Q typically spans several thousand bits to ensure the required security level. To allow efficient operations on such a large number, Residual Number System (RNS) [19], [20] is employed. The modulus Q can be represented as the product of the coprime moduli $Q = \prod_{i=0}^{L} q_i$, where each q_i is less than a machine word. Therefore, RNS enables operations over values in $\mathbb{Z}_Q$ without native support for multi-precision arithmetic. The scalar $x \in \mathbb{Z}_Q$ is represented with $L+1$ integers modulo each of q_i leveraging the isomorphism between $\mathbb{Z}_Q$ and the product group $\mathbb{Z}_{q_0} \otimes \cdots \otimes \mathbb{Z}_{q_L}$. We refer to a *limb* as one ciphertext polynomial corresponding to one modulus. A ciphertext at level ℓ in Full-RNS CKKS [19] has $2 \cdot N \cdot (\ell+1)$ coefficients and $2 \cdot (\ell+1)$ limbs.

TABLE I
CKKS Parameters and Descriptions

Param	Description
$\mathbf{m}$	Message vector in real or complex numbers
$\mathrm{P}_{\mathbf{m}}$	Plaintext polynomial encoding $\mathbf{m}$
$[\![\mathbf{m}]\!]$, $[\![\mathbf{m}]\!]_s$	Ciphertext encrypting $\mathbf{m}$ under key s
N	Degree. # of coefficients in a ciphertext polynomial
Q	Ciphertext initial modulus $= \prod_{i=0}^{L} q_i$
P	Auxiliary modulus for `Keyswitch` $= \prod_{i=0}^{k-1} p_i$
L	Maximum level of ciphertext
ℓ	Current level of ciphertext
k	# of moduli in P
$\mathcal{B}$	$\{p_0, \cdots p_{k-1}\}$ a set of prime moduli of P
$\mathcal{C}$	$\{q_0, \cdots q_L\}$ a set of prime moduli of Q
dnum	RNS decomposition number. Max # of digits
α	Fixed number. # of limbs in one digit $= (L+1)/\mathsf{dnum}$
β	# of digits in ciphertext $= \lceil \ell+1 \rceil / \alpha$

TABLE II
CKKS primitive operations

Operation	Output	`Keyswitch`	`Rescale`	`NTT`
`PAdd`$([\![\mathbf{m}]\!], \mathrm{P}_{\mathbf{m}'})$	$[\![\mathbf{m} + \mathrm{P}_{\mathbf{m}'}]\!]$	N	N	N
`PMult`$([\![\mathbf{m}]\!], \mathrm{P}_{\mathbf{m}'})$	$[\![\mathbf{m} \cdot \mathrm{P}_{\mathbf{m}'}]\!]$	N	Y	Y
`HAdd`$([\![\mathbf{m}]\!], [\![\mathbf{m}']\!])$	$[\![\mathbf{m} + \mathbf{m}']\!]$	N	N	N
`HMult`$([\![\mathbf{m}]\!], [\![\mathbf{m}']\!])$	$[\![\mathbf{m} \cdot \mathbf{m}']\!]$	Y	Y	Y
`Rotate`$([\![\mathbf{m}]\!], d)$	$[\![\mathbf{m} \ll d]\!]$	Y	N	Y
`Conjugate`$([\![\mathbf{m}]\!])$	$[\![\overline{\mathbf{m}}]\!]$	Y	N	Y

The complexity of polynomial multiplication is $O(N^2)$. To accelerate this time-consuming operation in FHE, Number Theoretic Transform (NTT) is used to reduce the complexity to $O(N \log N)$ [21]. During `HMult` and `Rotate`, an intermediate ciphertext $[\![\mathbf{m}]\!]_{s'}$ is generated. To switch it back to $[\![\mathbf{m}]\!]_s$, the `Keyswitch` operation is performed, which requires an evaluation key **evk**. The `Keyswitch` operation involves multiplication between key limbs and ciphertext limbs, thereby invoking the `NTT` operation. The original message is multiplied by a scaling factor Δ during encoding. After multiplying a ciphertext by an operand with a scaling factor Δ, the scale of the result becomes Δ^2. To reduce the scaling factor, `Rescale` is performed by discarding the last prime modulus q_ℓ and multiplying $q_\ell^{-1} \bmod q_i$ $(i = 0, \ldots, \ell-1)$ with the remaining limbs [19]. This is an approximation of dividing by Δ.

B. Bootstrapping

As more `Rescale` operations are performed, the multiplicative level of the ciphertext will reduce to 0, with only one limb q_0 left. No more multiplications can be allowed. To compute indefinitely on a ciphertext, a complex operation called bootstrapping is required to raise the modulus while preserving the correct structure of the ciphertext. Bootstrapping is the main bottleneck for FHE, and we briefly introduce the steps below. Due to limited space, we refer [9], [22] to readers for details.

- ModUp: Raises the ciphertext modulus from q_0 to Q. This yields a ciphertext with an increase in noise. The following three steps aim to homomorphically evaluate modular reduction operation modulo q_0 on the ciphertext to remove the related noise term.
- HE IDFT: Evaluates the decoding process (IDFT) homomorphically so that we can perform the slot-wise evaluation of approximated modular reduction.
- Sine Evaluation: Approximated modular reduction modulo q_0 utilizing a sine function with period q_0.
- HE DFT: Functions as the inverse operation of HE IDFT, producing a ciphertext that encrypts the correct message.

C. HE DFT

The homomorphic linear transformations, specifically HE DFT and HE IDFT, are the most time-consuming components of bootstrapping, accounting for over 60% of the total time in recent studies [9], [16], [23]. Meanwhile, bootstrapping itself represents the most costly operation in the RNS-CKKS scheme [24]. Here, we describe the baseline HE DFT [9] in Algo. 1, a multiplication process between a plaintext matrix and a ciphertext vector. The plaintext DFT matrix is factorized or decomposed into fftiter sparse diagonal submatrices, each containing r diagonals. This transforms the operation into the Hadamard product of plaintext matrix diagonals and the ciphertext vector, reducing the complexity and enabling SIMD computation. To align with different diagonals in the plaintext matrix, the ciphertext has to perform Rotate r times for each submatrix or fftiter.

Algorithm 1 Baseline Algorithm for HE DFT

Input: Ciphertext: $[\![\mathbf{m}]\!] = (\mathbf{a}, \mathbf{b})$, Rotation: $\{d_s^i, \mathbf{evk}_s^i\}$, Precomputed plaintext diagonals: $\boldsymbol{M}_{s,i}$

1: **for** $s = 1, \cdots,$ fftiter **do**
2: $\quad \mathbf{a}_j := \texttt{Decomp}(\mathbf{a})$ ▷ β digits
3: $\quad \hat{\mathbf{a}}_j := \texttt{ModUp}(\mathbf{a}_j)$ for $1 \le j \le \beta$ ▷ $Q^{(j)} \to PQ_\ell$
4: $\quad (\mathbf{a}_y, \mathbf{b}_y) \leftarrow (0, 0)$
5: $\quad$ **for** i from 1 to r **do**
6: $\quad\quad \hat{\mathbf{a}}_{\mathsf{rot}}^j := \texttt{Automorph}(\hat{\mathbf{a}}_j, d_s^i)$ for $1 \le j \le \beta$
7: $\quad\quad (\hat{\mathbf{u}}, \hat{\mathbf{v}}) := \texttt{KeyMultSum}(\hat{\mathbf{a}}_{\mathsf{rot}}^j, \mathbf{evk}_s^i)$
8: $\quad\quad (\mathbf{u}, \mathbf{v}) := (\texttt{ModDown}(\hat{\mathbf{u}}), \texttt{ModDown}(\hat{\mathbf{v}}))$ ▷ Q_ℓ
9: $\quad\quad \mathbf{b}_{\mathsf{rot}} := \texttt{Automorph}(\mathbf{b}, d_s^i)$
10: $\quad\quad (\mathbf{a}_y, \mathbf{b}_y) += \boldsymbol{M}_{s,i} \odot (\mathbf{u}, \mathbf{v} + \mathbf{b}_{\mathsf{rot}})$ ▷ DiagMultSum
11: $\quad$ **end for**
12: $\quad (\mathbf{a}_y, \mathbf{b}_y) := (\texttt{Rescale}(\mathbf{a}_y), \texttt{Rescale}(\mathbf{b}_y))$ ▷ $Q_{\ell-1}$
13: $\quad (\mathbf{a}, \mathbf{b}) \leftarrow (\mathbf{a}_y, \mathbf{b}_y)$
14: **end for**
15: **return** $[\![\mathbf{m}]\!] = (\mathbf{a}, \mathbf{b})$

Each Rotate requires Decomp and ModUp steps, and HE DFT operation requires multiple rotations of the ciphertext vector to align with the diagonals of each submatrice. The baseline algorithm is a widely adopted method by amortizing Decomp and ModUp among multiple rotations, a.k.a., hoisting [9] optimization, lifting both operations out of the rotation loop (line 5 in Algo. 1). The number of Fast Fourier Transform (FFT) stages (fftiter) equals $\log_{rdx} N$, where rdx denotes the radix, which is a power of 2. The number of rotations r in each iteration equals $2 \cdot rdx - 1$, the number of diagonals in each submatrix. The parameter fftiter affects both the bandwidth requirements and the raised multiplicative levels of ciphertext after bootstrapping. We will explore the trade-off between these factors in Section III.

RNS decomposition (Decomp) splits the ciphertext polynomial into β digits, each of which has α limbs except for the last digit ($\le \alpha$). We define the digit as one component of the split ciphertext. Due to the limited space, we refer [19] to readers for RNS basis conversion (BConv). It is used in modulus raising and reduction, which converts the residue of a polynomial into a new basis that is coprime to the original basis. We present ModUp and ModDown in Algo. 2 and Algo. 3, which correspond to raising and reducing the ciphertext level, respectively.

Algorithm 2 $\texttt{ModUp}_{\mathcal{C}\to\mathcal{B}\cup\mathcal{C}}([\mathbf{a}]_\mathcal{C})$

1: $[\tilde{\mathbf{a}}]_\mathcal{C} = (a^{(0)}, a^{(1)}, \ldots, a^{(\ell)})$
2: $[\mathbf{a}]_\mathcal{C} \leftarrow \texttt{INTT}([\mathbf{a}]_\mathcal{C})$ ▷ *coefficient domain*
3: $[\mathbf{a}]_\mathcal{B} \leftarrow \texttt{BConv}_{\mathcal{C}\to\mathcal{B}}([\mathbf{a}]_\mathcal{C})$
4: $[\tilde{\mathbf{a}}]_\mathcal{B} \leftarrow \texttt{NTT}([\mathbf{a}]_\mathcal{B})$ ▷ *evaluation domain*
5: **return** $\left(\tilde{a}^{(0)}, \ldots, \tilde{a}^{(k+\ell)}\right)$

Algorithm 3 $\texttt{ModDown}_{\mathcal{B}\cup\mathcal{C}\to\mathcal{C}}([\tilde{\mathbf{a}}]_{\mathcal{B}\cup\mathcal{C}})$

1: $[\tilde{\mathbf{a}}]_\mathcal{B} \leftarrow \texttt{INTT}([\tilde{\mathbf{a}}]_\mathcal{B})$ ▷ *coefficient domain*
2: $[\tilde{\mathbf{b}}]_\mathcal{C} \leftarrow \texttt{BConv}_{\mathcal{B}\to\mathcal{C}}([\tilde{\mathbf{a}}]_\mathcal{B})$
3: $[\tilde{\mathbf{b}}]_\mathcal{C} \leftarrow \texttt{NTT}([\tilde{\mathbf{b}}]_\mathcal{C})$ ▷ *evaluation domain*
4: **for** $0 \le j < \ell + 1$ **do**
5: $\quad a^{(j)} = P^{-1} \cdot \left(\tilde{a}^{(k+j)} - \tilde{b}^{(j)}\right) \pmod{q_j}$
6: **end for**
7: **return** $\left(a^{(0)}, \ldots, a^{(\ell)}\right)$

The Automorph operation is the key of Rotate operation, which permutes the polynomial given the rotation index d. The original slot in a ciphertext with index i, is mapped to a correspondingly rotated slot with index j via Eq. 1. The underlying message vector is circularly shifted by d.

$$j = \frac{5^d - 1}{2} + 5i \pmod{N} \tag{1}$$

As discussed, the Keyswitch operation is necessary for the Rotate operation. In the context of HE DFT, we refer to it as KeyMultSum.

III. Analytical Modeling

In this section, we design a cost model for the breakdown analysis of the on-chip memory requirement and off-chip data movement for HE DFT. We denote $\mathcal{B}_{coeff}$ as the size of a polynomial coefficient in bytes. The model takes HE parameters as inputs and generates the total DRAM transfer size and the required SRAM size to reuse all ciphertexts

TABLE III
SRAM REQUIREMENT ANALYSIS FOR BASELINE HE DFT

Subroutine (X)	**Input Size** (X_{in})	**Output Size** (X_{out})	**Required SRAM Size** (X_{SRAM})
`Decomp`	$(\ell+1)\cdot\mathcal{B}_{limb}$	$(\ell+1)\cdot\mathcal{B}_{limb}$	$2\cdot(\ell+1)\cdot\mathcal{B}_{limb}$
`ModUp`	$(\ell+1)\cdot\mathcal{B}_{limb}+\mathcal{B}_{tf}$	$\beta\cdot(\ell+k+1)\cdot\mathcal{B}_{limb}$	$(\beta\cdot(\ell+k+1)+(\ell+1))\cdot\mathcal{B}_{limb}$
`Automorph`	$(\beta\cdot(\ell+k+1)+(\ell+1))\cdot\mathcal{B}_{limb}$	$(\beta\cdot(\ell+k+1)+(\ell+1))\cdot\mathcal{B}_{limb}$	$2\cdot(\beta\cdot(\ell+k+1)+(\ell+1))\cdot\mathcal{B}_{limb}$
`KeyMultSum`	$\mathcal{B}_{\mathbf{evk}}+\beta\cdot(\ell+k+1)\cdot\mathcal{B}_{limb}$	$2\cdot(\ell+k+1)\cdot\mathcal{B}_{limb}$	$((\beta+2)\cdot(\ell+k+1)+2\cdot(\ell+1))\cdot\mathcal{B}_{limb}$
`ModDown`	$2\cdot(\ell+k+1)\cdot\mathcal{B}_{limb}+\mathcal{B}_{tf}$	$2\cdot(\ell+1)\cdot\mathcal{B}_{limb}$	$(\beta\cdot(\ell+k+1)+4\cdot(\ell+1))\cdot\mathcal{B}_{limb}$
`DiagMultSum`	$\mathcal{B}_{diag}+5\cdot(\ell+1)\cdot\mathcal{B}_{limb}$	$2\cdot(\ell+1)\cdot\mathcal{B}_{limb}$	$(\beta\cdot(\ell+k+1)+3\cdot(\ell+1))\cdot\mathcal{B}_{limb}$
`Rescale`	$2\cdot(\ell+1)\cdot\mathcal{B}_{limb}$	$2\cdot\ell\cdot\mathcal{B}_{limb}$	$2\cdot\ell\cdot\mathcal{B}_{limb}$

between subroutines. A ciphertext contains two elements **a** and **b**, each composed of $\ell+1$ degree-(N-1) polynomials. The initial ciphertext has $L+1$ limbs for each element. The size of one limb in bytes is:

$$\mathcal{B}_{limb} = N\cdot\mathcal{B}_{coeff} \tag{2}$$

A. SRAM Requirement Analysis

This subsection evaluates the amount of SRAM needed to eliminate the DRAM transfers of ciphertext in the baseline HE DFT. In the baseline dataflow, each subroutine can only be performed after the one before completed due to dependency. Therefore, if the on-chip memory size is less than one ciphertext size, more data transfer will occur from the DRAM, which is significantly bandwidth-inefficient. The input and output of each subroutine and the required SRAM size to store ciphertext on-chip are summarized in Table III. We explain how we derive the equations as follows.

1) Decomposition: The `Decomp` operation reads one element **a** of the ciphertext and outputs the decomposed version of the element into β digits with α limbs per digit. It does not change the ciphertext size.

2) ModUp: The `ModUp` operation raises the modulus of each digit to PQ_ℓ, so the number of limbs of each digit increases from α to $\ell+k+1$. The size of twiddle factors for `NTT&INTT` corresponds to the limb count, given as:

$$\mathcal{B}_{tf} = (\ell+k+1)\cdot\mathcal{B}_{limb} \tag{3}$$

3) Automorph: The `Automorph` operation permutes each digit of the element **a** and the other element **b**. It is necessary to store both input and output, as the input is required and reused in multiple rotation iterations. Although several precomputed powers-of-5, as indicated in Eq. 1, are needed, we exclude them from the cost model since their size is marginal.

4) KeyMultSum: The `KeyMultSum` requires to read one rotation key **evk** and all digits. Each key is a $2\times\beta$ matrix of polynomials, with $(\ell+k+1)$ degree-(N-1) polynomials for each matrix element. Therefore, one key size is given as:

$$\mathcal{B}_{\mathbf{evk}} = 2\cdot\beta\cdot(\ell+k+1)\cdot\mathcal{B}_{limb} \tag{4}$$

After multiplying each digit with the key components, the results from different digits are accumulated together. This process reduces the number of digits of **a** from β to 2.

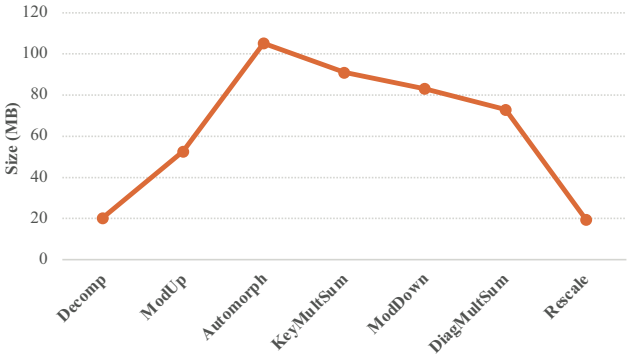

Fig. 1. The required SRAM size to store all intermediate ciphertext during each subroutine in HE DFT baseline dataflow (Set-A1 in Table V).

5) ModDown: The `ModDown` operation reduces the modulus of ciphertext back to Q_ℓ. This process involves loading the precomputed P^{-1} (mod q_j) for all $q_j\in\mathcal{C}_\ell$, as illustrated in Algo. 3. However, this is also excluded from the cost model as its size is negligible.

6) DiagMultSum: The `DiagMultSum` is a similar operation as `KeyMultSum`. It requires reading one matrix diagonal with size:

$$\mathcal{B}_{diag} = (\ell+1)\cdot\mathcal{B}_{limb} \tag{5}$$

7) Rescale: The `Rescale` operation is equivalent to `ModDown` operation but reduces only one limb in ciphertext, resulting in a modulus of $Q_{\ell-1}$.

8) Exemplar Study: According to the modeling results of baseline, the required SRAM size for Set-A1 in Table V to store intermediate ciphertext during each subroutine is shown in Fig. 1. This does not include auxiliary data such as the rotation key, plaintext matrix diagonals, and twiddle factors. The sizes of one rotation key $\mathcal{B}_{\mathbf{evk}}$ and one diagonal $\mathcal{B}_{diag}$ at the maximum level L are 84.9 MB and 1.3 MB, respectively.

B. Trade-off Analysis of fftiter

The number of evaluation keys relates to how the plaintext matrix gets factorized. The number of FFT stages fftiter corresponds to the number of factorized diagonal-sparse matrices. As introduced in Sec. II-C, the fftiter equals $\log_{rdx} N$, and number of rotations r required in each FFT stage is $2\cdot rdx-1$. If fftiter $=1$, the dense $N\times N$ DFT matrix has $2N-1$

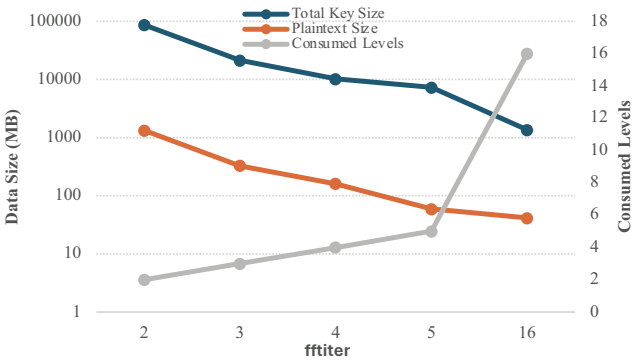

Fig. 2. Trade-off of fftiter on data size and consumed multiplicative levels with Set-A1.

diagonals, which requires $N-1$ rotations. Therefore, the total number of rotations r_{total} is given as:

$$r_{total} = \begin{cases} N-1, & \text{if fftiter} = 1 \\ \text{fftiter} \cdot (2 \cdot rdx - 1), & \text{otherwise} \end{cases} \quad (6)$$

Here, we assume that the radix rdx for each submatrix is the same. However, for a prime polynomial degree (e.g. $N = 2^{17}$), rdx cannot be the same for each submatrix, which could be $(2^6, 2^6, 2^5)$ if fftiter is 3.

Each rotation requires one rotation key. Hence, the total size of rotation keys that needs to be fetched from DRAM is:

$$\mathcal{B}_{\textbf{evks}} = \sum_{s=1}^{\text{fftiter}} \sum_{i=1}^{r} \textbf{evk}_s^i < r_{total} \cdot \mathcal{B}_{\textbf{evk}} \quad (7)$$

The number of diagonals in each submatrix is $2 \cdot rdx - 1$. The total size of the plaintext matrix is:

$$\mathcal{B}_{PtMat} = \sum_{s=1}^{\text{fftiter}} \sum_{i=1}^{r} \boldsymbol{M}_s^i < \text{fftiter} \cdot (2 \cdot rdx - 1) \cdot \mathcal{B}_{diag} \quad (8)$$

Taking the right-hand side of $<$ in Eq. 7 & 8 is useful for analyzing and understanding HE DFT operation. The right-hand side is larger because $\mathcal{B}_{\textbf{evk}}$ and $\mathcal{B}_{diag}$ are derived at the maximum level L. It does not consider the size shrinking because the number of limbs reduces by one after each fftiter due to `Rescaling`.

Fig. 2 shows the trade-off associated with fftiter, balancing between the total sizes of the key and plaintext matrix diagonals against the consumed levels. Increasing fftiter can decrease both the total key size and the number of required rotations, simplifying the complexity and reducing main memory traffic. However, the smallest $\mathcal{B}_{\textbf{evks}}$ is still 1.4 GB when fftiter takes its largest number 16, necessitating the loading of rotation keys from global memory. Moreover, a larger fftiter leads to the consumption of more multiplicative levels, reducing the raised level after bootstrapping. Consequently, it is impractical to select a very large fftiter.

TABLE IV
OFF-CHIP DATA MOVEMENT FOR BASELINE HE DFT

Subroutine (X)	External Memory Traffic (X_{DRAM})
`Decomp`	$\texttt{Decomp}_{in}$
`ModUp`	$\mathcal{B}_{tf} + \texttt{ModUp}_{out}$
`Automorph`	$\texttt{Automorph}_{in} + \texttt{Automorph}_{out}$
`KeyMultSum`	$\texttt{KeyMultSum}_{in} + \texttt{KeyMultSum}_{out}$
`ModDown`	$\texttt{ModDown}_{in} + \texttt{ModDown}_{out} - \mathcal{B}_{tf}$
`DiagMultSum`	$\texttt{DiagMultSum}_{in} + \texttt{DiagMultSum}_{out}$
`Rescale`	$\texttt{Rescale}_{in} + \texttt{Rescale}_{out}$

C. Off-chip Data Movement Analysis

The SOTA FPGA-based FHE accelerators [14], [25] with the acceleration card AMD U280 are equipped with 41 MB internal SRAM. This capacity is insufficient for the reuse of intermediate ciphertexts among subroutines, as illustrated in Fig. 1. In this subsection, we analyze the global memory access of baseline under the 41 MB constraint with Set-A1. The main memory traffic for each subroutine of the baseline dataflow is summarized in Table IV.

The `ModUp` operation increases the size of each digit from 3.4 MB (α limbs) to 14.2 MB ($L+k+1$ limbs). However, storing 3 (β) raised digits on-chip is unfeasible, necessitating their transfer back to DRAM. The `Automorph` operation requires reading the entire ciphertext. It is not feasible to store the original ciphertext and its permuted version on-chip. Therefore, the off-chip data transfers for `Automorph` must include both input and output. For the `KeyMultSum` operation, it is impossible to store one rotation key (84.9 MB) on-chip. This operation is limb-based, which means that the output polynomials are computed limb by limb, $2{\cdot}(L{+}k{+}1)$ in total. The `ModDown` operation reads two elements (28.4 MB) in modulo PQ_ℓ. Twiddle factors (14.2 MB) from `ModUp` can be reused. The outputs are two elements ($\mathbf{u}$,$\mathbf{v}$) in modulo Q_ℓ (20.3 MB), which have to be written back to DRAM. For the `DiagMultSum` operation, the intermediate result of $\mathbf{u} + \mathbf{b}_{rot}$ can be stored on-chip. Overall, within the baseline dataflow, all subroutines, except for `Decomp` and `ModUp`, require both reading from and writing to DRAM.

Therefore, the total off-chip data movement for baseline dataflow is given as:

$$\begin{aligned} \texttt{Total}_{DRAM} = \text{fftiter} \cdot (&\texttt{Decomp}_{DRAM} + \texttt{ModUp}_{DRAM} + \\ r \cdot (&\texttt{Automorph}_{DRAM} + \texttt{KeyMultSum}_{DRAM} + \\ &\texttt{ModDown}_{DRAM} + \texttt{DiagMultSum}_{DRAM}) + \\ &\texttt{Rescale}_{DRAM}) \end{aligned} \quad (9)$$

IV. PROPOSED DATAFLOW – LIMBFLOW

As discussed above, the intermediate results of most subroutines in the baseline dataflow cannot be stored on-chip due to the limited SRAM size on the FPGA. Therefore, the baseline involves significant off-chip data movement. In this section, we introduce an optimized fine-grained dataflow - LimbFlow, which significantly reduces off-chip data transfer and saves SRAM size compared with the baseline dataflow. Fig. 3 shows an overview of LimbFlow.

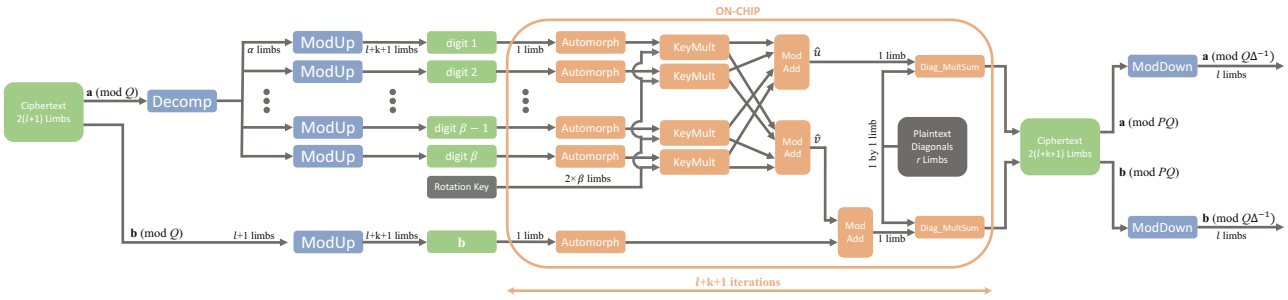

Fig. 3. Bandwidth Efficient LimbFlow. On-chip $\beta+1$ ciphertext limbs with the same modulus are reused r times for r rotations. Each iteration generates one limb for **a** and one limb for **b**. A total of $\ell+k+1$ iterations are required to generate all output ciphertext limbs.

A. Comparison with the Baseline Dataflow

1) Automorph: After `ModUp`, the ciphertext at its maximum level with the largest size is stored back in global memory. In the baseline, each digit with $\ell+k+1$ limbs is read, permutated, and written back to DRAM during `Automorph`. However, in LimbFlow, we read one limb from each digit (e.g., the first limb) since these limbs with the same modulus are used for the following computations. Therefore, LimbFlow achieves a finer-grained limb-level data reuse and removes the write-back operation of `Automorph`. The $\beta+1$ ciphertext limbs on-chip are reused r times, as we need r rotations for `DiagMultSum`.

2) KeyMultSum: In LimbFlow, the inputs for this operation include $\beta+1$ ciphertext limbs (reused from `Automorph`) and $2\cdot\beta$ key limbs for computation. This approach significantly reduces the off-chip data movement compared with the baseline. Moreover, two limbs are produced as outputs and stored on-chip; these can be reused for `DiagMultSum`. In the baseline scenario, the results from `KeyMultSum` must be written back to DRAM.

3) DiagMultSum: In the baseline, the plaintext diagonals are fetched sequentially, one by one, with a size of $\mathcal{B}_{diag}$ for each due to multiple rotations. Each diagonal comprises $\ell+1$ limbs. In contrast, LimbFlow reorders the processing sequence by fetching the limb with the same modulus from different diagonals (a total of r diagonals). Specifically, only one limb with the same modulus is fetched from each diagonal (e.g., the first limb) since their results can be immediately accumulated to produce a single output limb.

In the baseline, the output of each fftiter can be obtained only after completing all r rotations, resulting in a significant amount of off-chip data transfer. In contrast, LimbFlow processes the output ciphertext limb by limb. Consequently, it requires a total of $\ell+k+1$ iterations, with the $\beta+1$ ciphertext limbs in each iteration being reused across r rotations.

4) ModDown: In LimbFlow, `ModDown` is hoisted out of the rotation loop, which reduces its execution from $2\cdot\text{fftiter}\cdot r$ times to only $2\cdot\text{fftiter}$ times. This change amortizes the computational cost across multiple rotations. It also facilitates the merging of `ModDown` and `Rescale`, transitioning from reducing by modulus P to reducing by modulus $P\cdot\Delta$ [4], [26]. This optimization significantly reduces computation and memory access, as `ModDown` involves computation-intensive `NTT/INTT` operations.

5) Summary: Overall, LimbFlow significantly reduces off-chip data transfer among `Automorph`, `KeyMultSum`, and `DiagMultSum`. It is achieved by reorganizing the processing order and enabling finer-grained, limb-level data reuse, as the accumulation within `KeyMultSum` and `DiagMultSum` is performed on the limb with the same modulus. The outer loop in the baseline is r, reflecting the number of rotations the ciphertext undergoes. However, r serves as the inner loop to facilitate limb reuse in LimbFlow. Additionally, LimbFlow's limb-based processing significantly reduces SRAM requirements through data reuse and the strategic hoisting of `ModDown`.

B. SRAM Requirement Analysis

After the `ModUp` operation, the ciphertext with increased size must be stored back in the main memory on FPGA, as explained in Sec. III-C. However, as we merge the three subroutines `Automorph`, `KeyMultSum`, `DiagMultSum` with fine-grained limb reuse, we only need to store $\beta+1$ ciphertext limbs, $2\cdot\beta$ key limbs, and r diagonal plaintext limbs on-chip. Taking into account the twiddle factor reuse, LimbFlow (Set-A1) only requires 32 MB to eliminate the off-chip ciphertext movement among these three subroutines.

Although our analysis is conducted on a platform with limited SRAM, LimbFlow is scalable to larger designs equipped with more SRAM resources, such as ASICs. By designing a larger ciphertext buffer to accommodate all intermediate data, we can seamlessly integrate it into LimbFlow, further leveraging its advantages in reducing computation cycles. However, if the platform has a much smaller SRAM capacity, LimbFlow can prioritize storing the ciphertext limbs and loading the diagonal limbs in multiple travels. Essentially, LimbFlow represents a general dataflow technique that is applicable across a wide range of hardware platforms with various SRAM availability.

C. Off-chip Data Movement Analysis

In this subsection, we provide a detailed breakdown of the main memory traffic for LimbFlow compared to the baseline.

1) Automorph: In the baseline, the ciphertext is rotated r times, with each rotation requiring the entire ciphertext to be read from and then the permuted ciphertext to be written back to main memory. However, in LimbFlow, we load the ciphertext and perform permutation limb-wise. The permutated limbs are transferred to `KeyMultSum` operation without writing back to the DRAM. The $\beta + 1$ limbs are still on-chip for the next `Automorph`, which achieves data reuse r times across rotations. Therefore, the total DRAM transfer is reduced from $\mathsf{fftiter} \cdot r \cdot \mathtt{Automorph}_{DRAM}$ to $\mathsf{fftiter} \cdot (\mathtt{Automorph}_{DRAM}/2)$.

2) KeyMultSum: As mentioned, `KeyMultSum` takes the permutated β ciphertext limbs as input, and it only needs to load the corresponding $2 \cdot \beta$ key limbs for computation. However, we still need to read all the evaluation keys from DRAM. The loading of keys takes $r \cdot (\ell + k + 1)$ iterations in total. The result of each iteration is transferred to `DiagMultSum` directly without writing back to DRAM. Here, we reduce the off-chip data movement from $\mathsf{fftiter} \cdot r \cdot \mathtt{KeyMultSum}_{DRAM}$ to the total evaluation key size $\mathcal{B}_{evks}$.

3) DiagMultSum: The LimbFlow achieves accumulation from different diagonals immediately after the element-wise multiplication, as the results are derived from the same limb. The plaintext diagonals are loaded in $(\ell + k + 1)$ iterations. In each iteration, r diagonal limbs with the same modulus from r different diagonals are loaded. Under the 41 MB SRAM constraint, the results of `DiagMultSum` have to be written back to the main memory before `ModDown` can proceed. Therefore, the DRAM access is reduced from $\mathsf{fftiter} \cdot r \cdot \mathtt{DiagMultSum}_{DRAM}$ to $\mathcal{B}_{PtMat} + \mathsf{fftiter} \cdot (\mathtt{ModDown}_{in} - \mathcal{B}_{tf})$. Here, we consider the reuse of the twiddle factor.

4) ModDown & Rescaling: The off-chip data movement for `ModDown` is reduced from $\mathsf{fftiter} \cdot r \cdot \mathtt{ModDown}_{DRAM}$ to $\mathsf{fftiter} \cdot \mathtt{ModDown}_{DRAM}$. The `Rescaling` operation is incorporated into `ModDown` benefiting from hoisting `ModDown`.

The total off-chip data movement of LimbFlow is given as:

$$\mathtt{Total}_{DRAM}^{LF} = \mathsf{fftiter} \cdot (\mathtt{Decomp}_{DRAM} + \mathtt{ModUp}_{DRAM} + \mathtt{ModDown}_{DRAM} + \mathtt{Automorph}_{DRAM}/2 + \mathtt{ModDown}_{in} - \mathcal{B}_{tf}) + \mathcal{B}_{\mathbf{evks}} + \mathcal{B}_{PtMat} \tag{10}$$

V. FPGA Accelerator Design

The architecture of the accelerator implemented on the Alveo U280 FPGA is depicted in Fig. 4. The accelerator mainly consists of a Processing Element (PE) array, on-chip memory, interconnect, HBM, and a control unit. The following subsections will provide detailed descriptions of the design and functionality of these main units within the accelerator.

A. PE Array

The PE array includes dp PE units, achieving dp data parallelism. Each PE consists of one modular multiplier (MM) and one modular adder (MA). The PE can also perform modular subtraction (MS) or multiply-accumulate (MAC) controlled by the control logic. All computations within the PE are fully pipelined, and high-level subroutines are executed through the PE array.

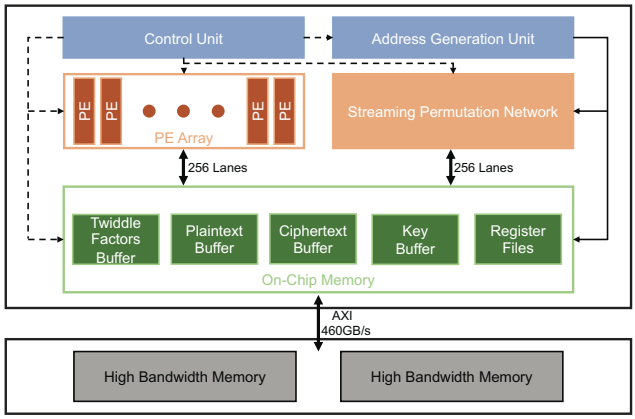

Fig. 4. The overview of the accelerator on Alveo U280

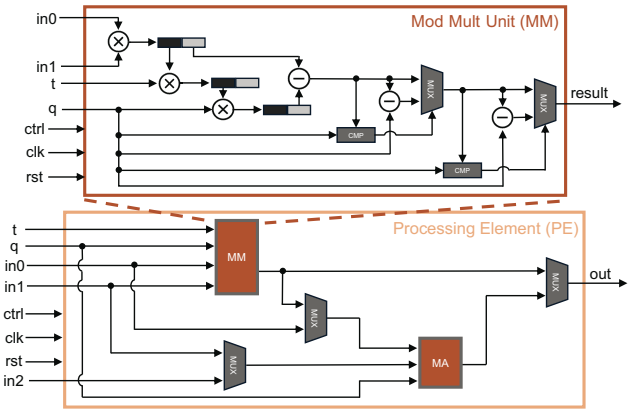

Fig. 5. The Architecture of PE unit and MM unit.

Fig. 5 shows the architecture of the PE unit and the MM unit. The MM unit is built based on the Barret Reduction [27]. The fully pipelined MM unit with shift registers can get one MM result per cycle. The pipelined MM is designed based on [28], which includes one full coefficient width Integer Multiplier (IntMult) and two half coefficient width IntMult.

B. On-chip Memory System

The on-chip memory mainly comprises the buffers for twiddle factor, plaintext diagonal, key, and ciphertext. The buffer size is designed according to the LimbFlow SRAM requirement analysis in Sec. IV. The pre-computed twiddle factors are stored in the twiddle factor buffer and reused in both `ModUp` and `ModDown`. The rotation key limbs and the plaintext diagonal limbs are loaded from HBM to their corresponding buffers, respectively.

These on-chip buffers are built by wisely stacking and combining available on-chip Block RAM (BRAM) and Ultra RAM (URAM). On the Alveo U280, each BRAM block has a data width of 18 bits and a depth of 1024. For example,

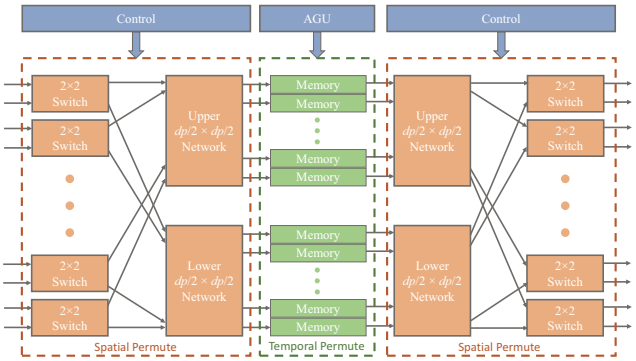

Fig. 6. The Architecture of Streaming Permutation Network (SPN)

by combining three BRAMs, we can store 1K 54-bit coefficients [14]. Stacking BRAM blocks achieves storing more coefficients. Similarly, the URAM block is in the dimension of 72-bit×4096. Three URAM blocks are combined to keep 16K 54-bit coefficients. We also utilize multiple register files (RFs) to store pre-computed values, e.g., powers-of-5 for Address Generation Unit (AGU) for `Automorph` permutation.

C. Interconnect

The interconnect mainly comprises the connection between the PE array and the memory on the chip, and the Streaming Permutation Network (SPN) [29] for the required permutation work in the HE DFT.

The SPN is based on a folded version multi-stage Benes network [30], which can achieve high-throughput arbitrary permutations. The SPN has three subnetworks, including two spatial networks and one temporal network, as shown in Fig. 6. The spatial network has $(dp/2) \cdot \log dp$ numbers of 2×2 switches to compose a dp-to-dp connection recursively. Temporal permutation is realized by issuing reads and writes to dp dual-port memory using addresses generated from AGU. The execution of SPN works as follows. First, N coefficients are streamed into the first spatial network with N/dp cycles, controlled by the routing table within the control logic. Then, the data are written into dp memory blocks based on the addresses from AGU. Lastly, dp coefficients are read out every cycle for the second spatial permutation, which also takes N/dp cycles in total.

D. Operation Mapping

In this subsection, we show how subroutines in HE DFT using LimbFlow are mapped on the accelerator.

1) Basis Conversion: The `BConv` operation is the core operation in `ModUp` and `ModDown`. Each coefficient output of a new limb is obtained by several consecutive MM and MAC operations performed within the PE array. If we are converting the RNS basis from $\mathcal{C}$ to $\mathcal{B}$, dp outputs are obtained after $(L+1) \times K$ MM and MAC operations. The intermediate results are stored in the ciphertext buffer, and the pre-computed moduli are stored in the register files.

TABLE V
PARAMETER SET FOR ANALYSIS AND EXPERIMENT

Param. Set	$\log q$	N	L	k	dnum	fftiter	λ
Set-A1	54	2^{16}	22	9	3	4	128
Set-A2	54	2^{16}	22	9	3	3	128
Set-A3	54	2^{16}	22	9	3	2	128
Set-B	64	2^{17}	39	11	2	4	128
Set-C	32	2^{16}	37	15	3	4	128

2) NTT&INTT: The `NTT` operation is very costly in `ModUp` and `ModDown` to convert the polynomial coefficients between *coefficient domain* and *evaluation domain*. Our datapath utilizes the same data mapping logic for both `NTT` and `INTT` with a unified Cooley-Tukey algorithm [31]. The PE units inside the PE array function as radix-2 ButterFly Units (BFU). Each BFU takes two inputs and generates two outputs by two MAC operations in parallel. One MAC operation performs modular multiplication and addition, and the other handles modular multiplication and subtraction. The fully pipelined architecture of the PE array enables the output of dp coefficients per cycle. Therefore, we need approximately $\log N \cdot N/dp$ cycles for the computation of `NTT` or `INTT`. The twiddle factors are precomputed and stored within the twiddle factor buffer. In terms of the permutation process between `NTT` stages, the ciphertext coefficients are fetched from the buffer to the SPN after each stage. The specific permutation step of each `NTT` stage is set by the routing table within the control logic, and the memory access within the temporal network is set by the AGU, which is also controlled by the control unit.

3) Automorph: The `Automorph` operation is also realized by the SPN. Unlike the fixed permutation pattern employed in `NTT` stages, the AGU initially computes the new mapping indices through bit-shifting and AND operations with the precomputed powers-of-5. Based on the new mapping indices, the coefficients are read from the ciphertext buffer and subsequently permuted within the SPN. This permutation process takes $2 \cdot N/dp$ cycles, with an additional minor delay occurring within the temporal network of the SPN. Upon completion, the permutation outputs are written back to the ciphertext buffer.

4) MultSum: The `KeyMultSum` and `DiagMultSum` operations are performed in the same manner within the PE array by the MAC operation. The limbs of key and plaintext matrix diagonals are loaded from HBM to their corresponding on-chip buffers. The buffer sizes are carefully designed to save the SRAM cost based on the LimbFlow SRAM requirement analysis in Sec. IV.

VI. EVALUATION

A. Experimental Setup

We implement our design on AMD Alveo U280 using Verilog HDL. The FPGA has 1,304K LUTs, 2,607K FFs, 41 MB on-chip SRAM, and 9,024 DSPs. We implement 256/512 PEs ($dp = 256$ or 512) and perform synthesis and place-and-route using Vivado 2023.1. The results are reported after place-and-route. The accelerator utilizes HBM to transfer off-chip data with up to 460 GB/s bandwidth.

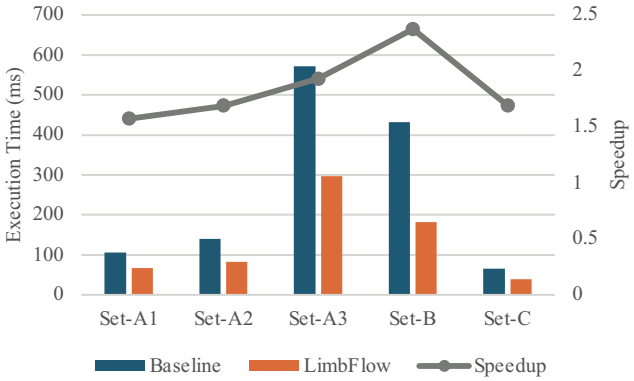

Fig. 7. HE DFT runtime comparison between baseline and LimbFlow.

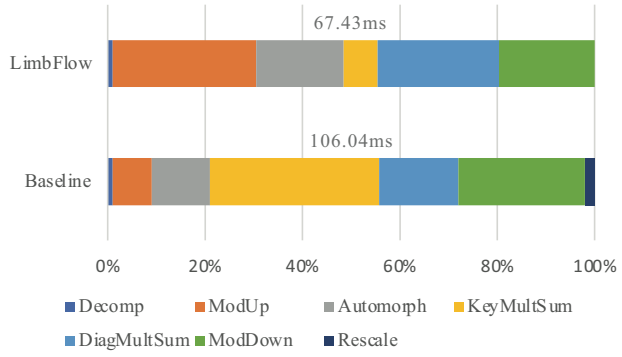

Fig. 8. HE DFT runtime breakdown comparison between baseline and LimbFlow using Set-A1.

We evaluate our design under different parameter settings in Table V. Our parameter set targets 128-bit security. Set-A is utilized to explore the impact of fftiter on the HE DFT. Set-B and Set-C are scaled up and down, respectively. Notably, only the design under Set-C is capable of achieving $dp = 512$.

B. Results and Analysis

1) Comparison with Baseline Dataflow:

Runtime & Speedup: The performance comparison is shown in Fig. 7. We vary fftiter across three different parameter sets within Set-A. As we decrease fftiter, the execution time increases for both dataflows. The reason is that smaller fftiter correspond to larger total evaluation key size and plaintext diagonal size, which leads to more global memory access. Moreover, smaller fftiter increases the number of diagonals, where r_{total} becomes dominated by radix rdx in Eq. 6. For instance, rdx is 2^8 under Set-A3. In terms of the speedup, a smaller fftiter leads to a larger speedup, as LimbFlow benefits from reducing off-chip data movement. Comparing the results from Set-A1 with Set-B, we observe that LimbFlow achieves a greater speedup at a higher degree and maximum level.

Off-Chip Data Movement: Table VI shows the global memory access for both baseline and LimbFlow under different parameter sets. Specifically, the results are derived from our cost model with the constraints of a limited SRAM size (41 MB). On average, LimbFlow can reduce global memory access 4.18×. The off-chip data traffic reduction improves as we decrease fftiter from Set-A1 to Set-A3.

TABLE VI
OFF-CHIP DATA MOVEMENT SIZE (IN GB)

	Baseline	LimbFlow	Traffic Reduction
Set-A1	57.05	14.22	4.01×
Set-A2	74.51	17.72	4.20×
Set-A3	472.23	112.00	4.22×
Set-B	156.72	34.11	4.59×
Set-C	47.97	12.28	3.91×

Runtime Breakdown: We compare the runtime breakdown between the baseline and LimbFlow in Fig. 8. Firstly, in the baseline, `KeyMultSum` consumes the most time due to significant off-chip data movement. However, in LimbFlow, `KeyMultSum` uses only a minimal portion of time, as both inputs and outputs of this operation are stored in on-chip memory. Secondly, our fine-grained ciphertext limb reuse across `Automorph`, `KeyMultSum`, and `DiagMultSum` reduce their execution time notably. The `ModUp` operation occupies a larger portion in LimbFlow, as it still requires writing the raised ciphertext back to HBM. Lastly, by hoisting `ModDown` outside the rotation loop and merging `Rescale` into `ModDown`, the consumed time proportion reduces to only roughly 20% in LimbFlow.

2) Resource Utilization: The resource utilization for LimbFlow is detailed in Table VII. The design using Set-B demands more BRAM/URAM due to the larger size and higher number of coefficients. Additionally, compared to Set-A, the 64-bit coefficients in Set-B require more multiplication stages for each MM unit, leading to increased DSP consumption. The design under Set-C also utilizes more DSP resources than Set-A due to the increased number of PE units from 256 to 512.

TABLE VII
LIMBFLOW RESOURCE UTILIZATION

Resource	Available	Utilized		
		Set-A	Set-B	Set-C
LUTs	1,304K	924,437	1,004K	743,832
FFs	2,607K	1,872K	2,001K	1,694K
DSP	9,024	5,120	8,192	7,168
BRAM	4,032	3,214	3,468	2,722
URAM	962	703	780	576

3) Comparison with SOTA works: Table VIII shows the performance comparison with the SOTA CPU/GPU works. LimbFlow achieves more than 100× speedup compared with the SOTA CPU implementation [9], [32]. Compared to the SOTA GPU design [16], LimbFlow can achieve a speedup of 4.9×. However, if LimbFlow operates at the same frequency as the GPU design, the speedup increases to more than 10×.

Table IX shows the performance comparison with the SOTA FPGA and ASIC implementations. It is important to note that not all previous studies disclose the runtime of HE DFT or its proportion within the bootstrapping process. Therefore,

TABLE VIII
PERFORMANCE COMPARISON WITH SOTA CPU/GPU WORKS

Work	Freq (GHz)	$(\log q, N)$	HE DFT (ms)	Speedup
Lattigo [9] (CPU)	3.5	$(64, 2^{14})$	8,400	**124.57**×
100x [16] (GPU)	1.2	$(54, 2^{16})$	205.81	**3.05**×
100x [16] (GPU)	1.2	$(54, 2^{17})$	330.40	**4.90**×
LimbFlow (FPGA)	0.3	$(54, 2^{16})$	**67.43**	-

TABLE IX
PERFORMANCE COMPARISON WITH SOTA ASIC/FPGA WORKS

Work	Freq (GHz)	$(\log q, N)$	HE DFT (ms)	Speedup
F1 [33] (ASIC)	1.0	$(32, 2^{14})$	34.98	0.90×
BTS [15] (ASIC)	1.2	$(50, 2^{17})$	30.25	0.78×
Poseidon [25] (FPGA)	0.45	$(32, 2^{16})$	76.37	**1.98**×
LimbFlow (FPGA)	0.3	$(32, 2^{16})$	**38.52**	-

we estimate the HE DFT runtime to be 60% of the total bootstrapping time here. This estimate is conservative since the HE DFT portion reported by existing works is consistently above 60%, including CPU [9], GPU [16], and ASIC [23].

ASIC-based acceleration works [33], [15] are still more efficient than LimbFlow, as they use a substantial number of PEs and hundreds of megabytes of on-chip SRAM. It is important to note that LimbFlow, as a general dataflow technique, can also be implemented on ASICs to achieve further speedup with more resources. For example, with 512 MB of SRAM, LimbFlow can also eliminate DRAM transfers for operations such as `ModUp` and `ModDown`. Furthermore, LimbFlow can still see reduced computing cycles through strategic reordering and merging of subroutines.

For the FPGA implementations, our primary focus is on Poseidon [25] which maximizes the computing efficiency of operator cores. We achieve a 1.98× speedup, as Poseidon employs a baseline-like algorithm for the HE DFT operation, which still suffers from the main memory traffic. It should be noted that we select Set-C to ensure a fair comparison. By utilizing a 32-bit coefficient size, we can achieve data parallelism of 512, an improvement over the 256 parallelism achieved with Set-A and Set-B, as depicted in Fig. 4.

VII. RELATED WORK

CPU/GPU: Several FHE libraries like HEAAN [10], [34], HELib [35], [36], Lattigo [32], [9] developed supports bootstrapping. However, the performance is limited due to the large number of DRAM accesses of the HE DFT operation on the CPU. For example, it takes roughly eight minutes to bootstrap 128 slots within a ciphertext of degree 2^{16} using the HEAAN library. The pioneering GPU-based study [16] supporting bootstrapping introduced a kernel-fusion technique to minimize off-chip data access. However, certain kernels remained unfused due to constraints in on-chip storage capacity. GPU is not specifically tailored for FHE and lacks the implementation of key operators within its computing cores. Moreover, GPUs have higher power consumption than FPGA/ASICs.

FPGAs: Most FPGA-based FHE accelerators [37], [38], [39], [40] only support small parameter sets or certain operations without supporting bootstrapping. Recent works supporting bootstrapping achieve higher performance and efficiency in comparison with CPU/GPU-based acceleration. FAB [14] leveraged the hoisting method, as proposed by Halevi and Shoup [17], to amortize certain operations across multiple rotations in HE DFT. The bootstrapping algorithm of Poseidon [25] is based on [9]. However, all previous FPGA-based accelerators do not consider fine-grained ciphertext limb reuse in bootstrapping and are constrained by the limited on-chip SRAM. We are the first to propose an accelerator specifically tailored for the bottleneck of bootstrapping - HE DFT, reducing the SRAM requirement and off-chip data movement.

ASICs: F1 [33], which is distinctively designed for parameter sets with a low polynomial degree, is limited to supporting single-slot bootstrapping operations that exhibit low throughput. BTS [15] applied the matrix factorization approach for HE DFT based on [22], yet it did not incorporate bandwidth-aware optimizations on bootstrapping. ARK [23] developed a strategy generating matrix plaintext limbs on-the-fly during HE DFT to reduce main memory access. Although ASIC-based accelerators achieve high performance, they require substantial hardware resources, including high operating frequency (e.g., 2 GHz), a large number of PEs (e.g., 2048), and hundreds of megabytes of on-chip SRAMs (e.g., 512 MB).

VIII. CONCLUSION

Bootstrapping represents the most costly operation in FHE, with the HE DFT operation being the most time-consuming aspect due to the extensive amount of global memory access. In this paper, we design an analytical cost model to assess off-chip data movement for the HE DFT operation. Furthermore, we propose a bandwidth-efficient HE DFT dataflow named LimbFlow, which enables fine-grained data reuse and decreases the dependency on SRAM. LimbFlow significantly reduces off-chip memory access compared to baseline dataflow. This efficiency makes it suitable for practical FPGA platforms that have limited on-chip memory, without sacrificing its versatility. As a general dataflow technique, LimbFlow can also be seamlessly adapted to other hardware platforms. Additionally, we develop an FPGA accelerator tailored for HE DFT based on LimbFlow, demonstrating 4.90× and 1.98× speedup against SOTA GPU and FPGA implementations.

IX. ACKNOWLEDGEMENT

This work is supported by the U.S. National Science Foundation (NSF) under grants CCF-1919289 and OAC-2311870. Equipment and support by AMD AECG are greatly appreciated.
Distribution Statement A: Approved for public release. Distribution is unlimited.

REFERENCES

[1] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[2] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer, 2010, pp. 1–23.

[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[4] L. de Castro and V. Vaikuntanathan, "Does fully homomorphic encryption need compute acceleration?" *arXiv preprint arXiv:2112.06396*, 2021.

[5] J. H. Cheon, K. Han, and M. Hhan, "Faster homomorphic discrete fourier transforms and improved fhe bootstrapping," Cryptology ePrint Archive, Paper 2018/1073, 2018. [Online]. Available: https://eprint.iacr.org/2018/1073

[6] Xilinx, "Xilinx UltraScale+ HBM FPGAs," 2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale- plus-hbm.html

[7] Intel, "Intel® oneAPI Math Kernel Library," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/docs/onemkl/get-started- guide/2023-0/overview.html

[8] AMD, "AMD ACAP versal HBM-series," 2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/acap/versal-hbm.html#productAdvantages

[9] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 587–617.

[10] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology – ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409–437.

[11] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology - CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Springer Berlin Heidelberg, 2014, pp. 554–571.

[12] J. H. Cheon, K. Han, and M. Hhan, "Faster homomorphic discrete fourier transforms and improved fhe bootstrapping," Cryptology ePrint Archive, Paper 2018/1073, 2018. [Online]. Available: https://eprint.iacr.org/2018/1073

[13] H. Chen, I. Chillotti, and Y. Song, "Improved bootstrapping for approximate homomorphic encryption," in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 34–54.

[14] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Fab: An fpga-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 882–895.

[15] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "Bts: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 711–725. [Online]. Available: https://doi.org/10.1145/3470496.3527415

[16] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 4, p. 114–148, Aug. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9062

[17] S. Halevi and V. Shoup, "Faster homomorphic linear transformations in helib," in *Advances in Cryptology – CRYPTO 2018*, H. Shacham and A. Boldyreva, Eds. Cham: Springer International Publishing, 2018, pp. 93–120.

[18] T. Ye, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Performance modeling and fpga acceleration of homomorphic encrypted convolution," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 115–121.

[19] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full rns variant of approximate homomorphic encryption," in *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer, 2019, pp. 347–368.

[20] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A full rns variant of fv like somewhat homomorphic encryption schemes," in *International Conference on Selected Areas in Cryptography*. Springer, 2016, pp. 423–442.

[21] Z. Liang and Y. Zhao, "Number theoretic transform and its applications in lattice-based cryptosystems: A survey," *arXiv preprint arXiv:2211.13546*, 2022.

[22] K. Han and D. Ki, "Better bootstrapping for approximate homomorphic encryption," in *Cryptographers' Track at the RSA Conference*. Springer, 2020, pp. 364–390.

[23] J. Kim, G. Lee, S. Kim, G. Sohn, M. Rhu, J. Kim, and J. H. Ahn, "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1237–1254.

[24] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, "DaCapo: Automatic bootstrapping management for efficient fully homomorphic encryption," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024.

[25] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 870–881.

[26] R. Agrawal, L. De Castro, C. Juvekar, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "Mad: Memory-aware design techniques for accelerating fully homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 685–697.

[27] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[28] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "Fpga-based accelerators of fully pipelined modular multipliers for homomorphic encryption," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019, pp. 1–8.

[29] R. Chen and V. K. Prasanna, "Automatic generation of high throughput energy efficient streaming architectures for arbitrary fixed permutations," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[30] V. E. Beneš, "Optimal rearrangeable multistage connecting networks," *Bell system technical journal*, vol. 43, no. 4, pp. 1641–1656, 1964.

[31] Norton and Silberger, "Parallelization and performance analysis of the cooley–tukey fft algorithm for shared-memory architectures," *IEEE Transactions on Computers*, vol. 100, no. 5, pp. 581–591, 1987.

[32] C. V. Mouchet, J.-P. Bossuat, J. R. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," in *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, no. CONF, 2020, pp. 64–70.

[33] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 238–252.

[34] K. Han, M. Hhan, and J. H. Cheon, "Improved homomorphic discrete fourier transforms and fhe bootstrapping," *IEEE Access*, vol. 7, pp. 57 361–57 370, 2019.

[35] S. Halevi and V. Shoup, "Algorithms in helib," in *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I 34*. Springer, 2014, pp. 554–571.

[36] ——, "Bootstrapping for helib," *Journal of Cryptology*, vol. 34, no. 1, p. 7, 2021.

[37] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "Heax: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.

[38] Y. Yang, S. R. Kuppannagari, R. Kannan, and V. K. Prasanna, "Nttgen: a framework for generating low latency ntt implementations on fpga," in

Proceedings of the 19th ACM International Conference on Computing Frontiers, 2022, pp. 30–39.

[39] Y. Yang, W. Long, R. Kannan, and V. K. Prasanna, "Fpga acceleration of rotation in homomorphic encryption using dynamic data layout," in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2023, pp. 174–181.

[40] M. Han, Y. Zhu, Q. Lou, Z. Zhou, S. Guo, and L. Ju, "coxhe: A software-hardware co-design framework for fpga acceleration of homomorphic computation," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1353–1358.