A Unified Vector Processing Unit for Fully Homomorphic Encryption

Jiangbin Dong[1,5*] Xinhua Chen[2,4] Mingyu Gao[3,4,5†]
Xi'an Jiaotong University[1] Fudan University[2] Tsinghua University[3]
Shanghai Qi Zhi Institute[4] Institute for Interdisciplinary Information Core Technology, Xi'an[5]
bill412@stu.xjtu.edu.cn xhchen21@m.fudan.edu.cn gaomy@tsinghua.edu.cn

***Abstract*—Fully homomorphic encryption (FHE) algorithms enable privacy-preserving computing directly on encrypted data without leaking sensitive contents, while their excessive computational overheads could be alleviated by specialized hardware accelerators. The vector architecture has been prominently used for FHE accelerators to match the underlying polynomial data structures. While most FHE operations can be efficiently supported by vector processing units, the number theoretic transform (NTT) and automorphism operators involve complex and irregular data permutations among vector elements, and thus are handled with separate dedicated hardware units in existing FHE accelerators. In this paper, we present an efficient inter-lane network design and the corresponding dataflow control scheme, in order to realize NTT and automorphism operations among the multiple lanes of a vector unit. An arbitrarily large operator is first decomposed to fit in the fixed width of the vector unit, and the required data permutation and transposition are conducted on the specialized inter-lane network. Compared to previous designs, our solution reduces the hardware resources needed, with up to $9.4\times$ area and $6.0\times$ power savings for only the inter-lane network, and up to $1.2\times$ area and $1.1\times$ power savings for the whole vector unit.**

***Index Terms*—fully homomorphic encryption, hardware accelerator, vector processing unit, network**

I. Introduction

Data privacy has become a critical concern nowadays, driven by two important trends: the increasingly precious value of user data in the era of artificial intelligence, and the great interest of outsourcing data computations to public cloud computing platforms that are potentially vulnerable. Homomorphic encryption is a family of cryptographic algorithms that allow a user to encrypt her sensitive data before sending to the cloud. The untrusted cloud platform only sees and operates on the encrypted ciphertext. The user then receives the processed ciphertext and decrypts into a plaintext output, which is equivalent to performing the corresponding computations on the plaintext input. This enables computation outsourcing without compromising data privacy. Fully homomorphic encryption (FHE) supports infinite amounts of both addition and multiplication on ciphertexts through a technique called bootstrapping [1]. Although several FHE schemes have been developed [2]–[5], they have not yet been widely deployed in real-world applications, mostly due to the excessive computational cost of homomorphic ciphertext processing, e.g., over $1000\times$, compared to the corresponding plaintext computations.

To overcome the high computational overheads, various domain-specific accelerators for FHE have been proposed [6]–[12]. Most of them follow the vector architecture, where the processing unit has multiple lanes that share similar control and compute on a data vector in parallel. This design choice is motivated by the underlying data structures of modern FHE algorithms, which are mainly based on polynomials of very high degrees up to several thousands. A polynomial, when represented by either its coefficients or a set of evaluation values, naturally forms a long vector. Many basic FHE operations can then be viewed as regular vector operations and efficiently mapped onto such vector architectures. However, two crucial exceptions exist, which are number theoretic transform (NTT) and automorphism. Both involve complex and irregular permutations among the vector elements, unfriendly to the vector processing style. Consequently, previous FHE accelerators typically use dedicated hardware units, such as complex data permutation networks like full crossbars, and/or large on-chip SRAM buffers, to support these two operations.

In this paper, we propose a unified vector processing unit that is able to support all the FHE operations, including the challenging NTT and automorphism. The unit contains multiple lanes, each consisting of modular adder/multiplier and register resources. It naturally supports element-wise operations, as well as cross-lane reduction for matrix/tensor multiplications. Each pair of adjacent lanes can also be combined to realize the butterfly operations in the NTT and inverse NTT kernels.

The key component is *a novel inter-lane network* that efficiently realizes the data permutations required in the irregular NTT and automorphism operations, with very low area and power cost. More specifically, our inter-lane network uses the constant-geometry network [13], [14] to support NTT, and uses a multi-stage shift network to realize data transposition and automorphism. A key contribution of our work is we realize that *an arbitrary automorphism can be decomposed into a series of simple shifts* and thus can be handled by our inter-lane network. In addition, we support mapping NTT and automorphism operations of arbitrarily large sizes to the vector processing unit through decomposing them into smaller ones that match the hardware size, i.e., the number of lanes [9], [15]. While data dimension transpositions would be needed during such decomposition, we again realize them on the same shift network without extra dedicated units, demonstrating the benefits of unified hardware design.

* Work done during the internship at Institute for Interdisciplinary Information Core Technology, Xi'an.
† Corresponding author.

978-3-9826741-0-0/DATE25/© 2025 EDAA

We compare our design with previous methods that use complex networks and/or dedicated SRAM buffers to realize NTT and automorphism. When ported to the same vector processing unit architecture with identical number of 64 lanes, our inter-lane network design reduces the area cost by $1.6\times$ to $9.4\times$, and the power consumption by $2.9\times$ to $6.0\times$, for supporting these two irregular operations. The full vector processing unit, while dominated by the modular arithmetic logic and the register files, can still exhibit up to $1.2\times$ area and $1.1\times$ power savings. At the same time, our unified inter-lane network does not significantly compromise performance. The computation logic is 75% to 85% utilized compared to the ideal case when running various NTT operations, and always fully utilized for automorphism.

In summary, our paper makes the following contributions.

- We discover a novel decomposition for automorphism, which transforms an automorphism of arbitrary length into only a set of shift operations.
- We propose a novel inter-lane network design consisting of constant-geometry NTT connections and multi-stage shifts, which can realize all the necessary permutations for NTT and automorphism of arbitrary length.
- We design a unified vector processing unit, using the above inter-lane network and multiple lanes of modular arithmetic logic, to support all the types of operations in modern FHE algorithm schemes.
- We compare our vector processing unit with previous approaches, showing up to $9.4\times$ area and $6.0\times$ power savings for only the inter-lane network, and up to $1.2\times$ area and $1.1\times$ power savings for the whole vector unit.

II. Background

We first give a brief overview of FHE, particularly the CKKS scheme. Then we introduce number theoretic transform and automorphism, which need special optimizations in vector architectures. Finally we discuss previous FHE accelerator designs and how they implement the above two operations.

A. Fully Homomorphic Encryption (FHE)

Several FHE schemes have been proposed thus far. In this paper we mainly discuss the most efficient CKKS scheme [2], while other schemes like BGV [3], BFV [4], [5] can also be similarly supported given their similar computation patterns.

In CKKS, each ciphertext can encrypt $N/2$ plaintext numbers that are processed in the SIMD manner [16], [17]. A ciphertext is represented as two polynomials with N coefficients in a large modular field $\mathbb{R}_Q$. Using the residual number system (RNS), each coefficient can be decomposed into L numbers, each in a smaller field $\mathbb{R}_{q_i}$ with narrower bitwidth for more efficient computations [18]–[20]. Thus each ciphertext is effectively a tensor of shape $2\times N\times L$. Homomorphic addition (HAdd) adds two ciphertext tensors in an element-wise way. Homomorphic multiplication (HMult) involves polynomial multiplications that can be accelerated using *number theoretic transforms* (NTTs), and then goes through a process of *keyswitch* that contains element-wise operations, matrix/tensor multiplications, as well as NTTs. Homomorphic rotation (HRot), which shifts the plaintext numbers in a ciphertext, involves a special operation called *automorphism*, after which a keyswitch pass is also needed. After a certain number of homomorphic operations are applied, the ciphertext would gather too much noise that makes decryption fail. At this time, a complicated process called *bootstrapping* is necessary to reset the ciphertext noise. We omit the details of bootstrapping here, but emphasize that it involves the same basic operations including HAdd, HMult, and HRot. For more details, please refer to [21]–[23].

As a summary, homomorphic computations, including bootstrapping, mainly involve element-wise operations, matrix/tensor multiplications, NTTs, and automorphisms. Most of these operations can be efficiently supported with vector-style processing. This observation has motivated previous FHE accelerators to use vector architectures [6]–[9], [11], [12]. However, NTT and automorphism involve irregular data permutations within each polynomial, and cannot be directly handled in the vector manner. We focus on these two operations in this paper.

B. Number Theoretic Transform (NTT)

Similar to fast Fourier transforms on complex data, NTTs perform on integer field data, and transform polynomials between the coefficient domain and the value domain in $O(N\log N)$ complexity. NTTs are widely used in cryptography to accelerate polynomial multiplications, by first performing NTTs to convert the input polynomials into the value domain, then doing element-wise multiplications, and finally converting the result back to coefficients through an inverse NTT (iNTT). The computation of (i)NTT on N elements has $\log N$ stages, where the ith stage does *butterfly* operations on each pair of elements with a specific stride of distance $N/2^i$. Pre-computed *twiddle factors* are also needed in these butterfly operations.

Mapping NTTs onto vector architectures has several difficulties. First, when N is large and the elements do not all fit in the local buffer, fetching the strided input elements exhibits irregular data access patterns with little locality, resulting in excessive expensive accesses to the off-chip memory and/or across the global chip interconnect. Second, even with small N values, organizing the elements in the strided patterns to feed the multiple processing units needs special data permutations between the vector processing lanes.

NTT decomposition. We use the widely used NTT decomposition method [15], [24]–[26] to address the first challenge above, which recursively decomposes a large NTT into multiple smaller NTTs that each matches the hardware size. Assume the hardware has the computing and buffering resources to process a length-m NTT each time. We can decompose an arbitrary NTT of length N into $\lceil\log N/\log m\rceil$ dimensions, which are processed one-after-one. For each dimension, we independently process all the individual length-m NTTs. When switching dimensions, we need to transpose the data to continuously collect the elements in each NTT of the next dimension, and also do a pass of element-wise twiddle factor multiplications on all the elements. For example, with 2D decomposition of $N = R\times C$, we first process the R instances of row NTTs, each of length C. Then we do element-wise twiddle multiplications on all the N elements and transpose the matrix into column-

major. Finally we compute the C instances of column NTTs, each of length R.

C. Automorphism

Automorphism is a permutation on N elements, which moves the input elements following the mapping $\sigma_{\Phi,r}(i)$ in Eq. (1), i.e., the element of index i goes to the new position at index $\sigma_{\Phi,r}(i)$. Φ is a number co-prime to the input length N, typically chosen as $\Phi = 5$.

$$\sigma_{\Phi,r} : \ i \ \mapsto \ i \times \Phi^r \bmod N, \qquad i = 0, 1, \ldots, N-1 \quad (1)$$

Automorphism followed with keyswitch is used to implement HRot in FHE. Specifically, applying $\sigma_{\Phi,r}(i)$ on the ciphertext polynomials would rotate the plaintext numbers in the ciphertext with a distance of r, i.e., from $(z_0, z_1, \ldots, z_{N/2-1})$ to $(z_r, \ldots, z_{N/2-1}, z_0, \ldots, z_{r-1})$.

Similar to NTT, automorphism also exhibits irregular data permutations among the polynomial elements. Note that although the encrypted plaintext numbers are rotated in a cyclic way, the ciphertext element movement has little locality. For example, with $N = 64$ and $r = 2$, the original elements at $0, 1, 2, 3, 4, \ldots$ should be moved to $0, 5, 25, 61, 49, \ldots$. Consequently, we need efficient optimizations to address the random off-chip accesses and the complex on-chip permutations.

Automorphism decomposition. Previous work has noticed that an automorphism can also be decomposed into small permutations [7], [9], [10]. This allows us to overcome the random off-chip access inefficiency similarly to NTTs above. For example, we can view the N elements as a row-major matrix with R rows and C columns where $N = R \times C$. Then the elements in the same column would remain in the same column after automorphism [9], [11]. This is because, for an original element i whose column index $c = i \bmod C$, after automorphism its new column $c' = \sigma_{\Phi,r}(i) \bmod C = i \times \Phi^r \bmod C = c \times \Phi^r \bmod C$ only depends on c. This property suggests that an automorphism can be processed column by column, where each column is permuted independently, and then stored into the new column position. The new row and column indices are given as below [11].

$$r' = r \times \Phi^r \bmod R + \left\lfloor \frac{c \times \Phi^r}{C} \right\rfloor \bmod R \quad (2)$$

$$c' = c \times \Phi^r \bmod C \quad (3)$$

Here Eq. (3) is effectively a smaller automorphism on the C columns. The permutation in Eq. (2) contains two parts. The first term is also a smaller automorphism on the R elements in this column. The second term is independent on r and thus a constant in each specific column, representing a cyclic shift.

D. Related Work

Various domain-specific accelerators for FHE [7]–[10], [12] followed the vector-style architecture that includes a set of vector processing units to process the many elements in the ciphertext tensors in parallel. However, to support the vector-unfriendly operations of NTT and automorphism that irregularly permute elements in one vector (i.e, a polynomial), previous designs mostly used dedicated units and complicated networks, as summarized in Table I. We compare our optimized unified network design with them quantitatively in Section V.

TABLE I
COMPARISON OF RELATED DESIGNS.

Design	Transpose in NTT	Automorphism
F1 [7]	Quadrant-swap buffers	Cyclic shift + transpose
CraterLake [8]	Fixed network	Fixed network
BTS [10]	Crossbars	Crossbars
ARK [9]	Dedicated unit	Dedicated network
SHARP [12]	Quadrant-swap buffers	Dedicated network
Ours	Unified constant-geometry + shift network	

For NTT, all designs applied 2D or 3D NTT decomposition and then focused on the small NTTs that fit in hardware. The small NTTs were processed using dedicated NTT units with customized networks to realize the butterfly access patterns. For the transpose between dimensions, F1 [7] and SHARP [12] used SRAM buffers for hierarchical quadrant swaps; CraterLake [8] used a fixed permutation network dedicated to transpose; BTS [10] did implicit transpose when transferring data along its global horizontal and vertical crossbars, storing data elements to the new addresses in the destinations; ARK [9] also adopted a dedicated transpose unit.

For automorphism, F1 [7] used cyclic shifts in conjunction with the aforementioned transposition unit. CraterLake [8] instead used a fixed network to simplify control complexity. ARK [9] and SHARP [12] implemented a specialized automorphism unit that contains a complex multi-stage permutation network. BTS [10] again used its global crossbars to implicitly conduct automorphism through specialized addressing schemes.

III. Hardware Architecture

In this paper, we propose a unified vector processing unit (VPU) architecture that supports all the FHE operations. The key to supporting the vector-unfriendly NTT and automorphism operations in the VPU is the design of an inter-lane network. This section describes the hardware of the VPU. Section IV discusses how to map NTT and automorphism to the VPU.

Fig. 1(a) shows the overall hardware architecture of the FHE accelerator, which contains multiple VPUs connected through a network-on-chip (NoC). On-Chip SRAM is used to cache data for maximum reuse. This high-level architecture follows the common structure of recent FHE accelerators [7]–[10], [12]. Our novel design lies inside the VPU, as shown in Fig. 1(b), which has m computing lanes (by default $m = 64$), interconnected with an inter-lane network. We next discuss the lane and the inter-lane network in details.

A. Computing Lane

As illustrated in Fig. 1(c), each computing lane in the VPU has identical structure, including a modular multiplier, a modular adder/subtractor, and a register file with two read ports and one write port. This naturally allows the multiple lanes in a VPU to conduct element-wise modular addition/subtraction/multiplication with the data in the register

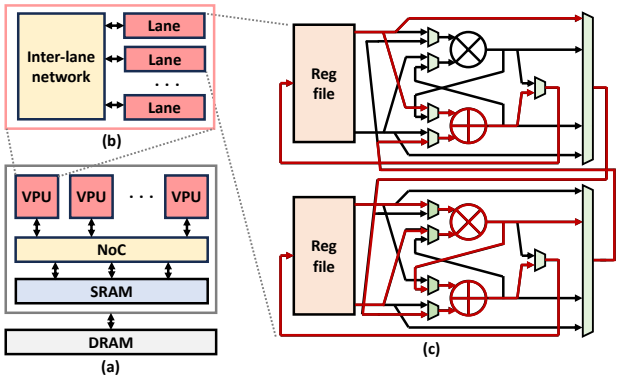

Fig. 1. The overall architecture (a), the vector processing unit (b), and the lane structure (c). The inter-lane network is shown in Fig. 2. The datapath of a DIT butterfly operation is highlighted in (c).

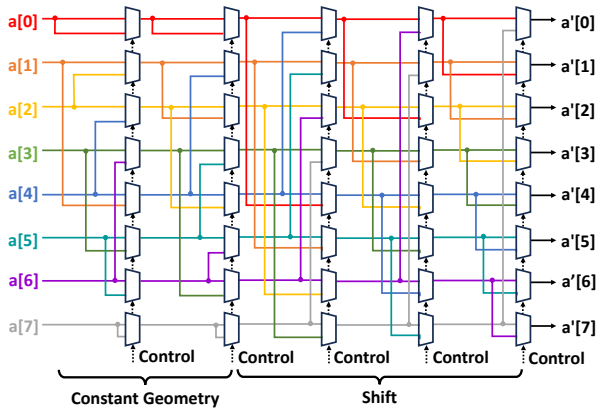

Fig. 2. The inter-lane network, which consists of two stages of DIT/DIF constant-geometry, and several stages of shift. Showing an example of $m = 8$.

files. For matrix/tensor multiplications, besides element-wise operations, cross-lane reduction is also needed, which can be trivially done using the shift functionality of the inter-lane network as described in Section III-B. We use Barrett reduction for the modular arithmetics [6], [9]–[12], [27], as it could better support the base conversion in the FHE keyswitch than Montgomery multipliers.

Each pair of lanes also has connections that allow them to directly obtain each other's data, in order to facilitate the butterfly operation in NTT that involves both lanes' data. Note that to support NTT and iNTT without the need of bit reverse, both decimal-in-time (DIT) and decimal-in-frequency (DIF) butterfly patterns are needed. Fig. 1(c) highlights the activated datapath for DIT; DIF can be similarly executed. Besides the two data elements, the twiddle factor is also read from the register file in one of the two lanes.

B. Inter-Lane Network

The inter-lane network is illustrated in Fig. 2. It consists of two NTT constant-geometry (CG) stages [13], [14] for DIT and DIF, respectively, plus a shift network with $\log m$ stages each shifting a distance of $m/2, m/4, \ldots, 1$. When $m = 4$, the two CG stages are identical and merged into one. Each stage has m 2-to-1 MUX'es, selecting between the local lane's and another specific lane's elements. With typical numbers of lanes like $m = 32, 64$, there are only 7 to 8 stages, whose critical path is well within the desired clock cycle like 1 GHz.

Note that the MUX'es in the shift stages use separate control signals, so they can be independently controlled as long as there is no conflict. For example, in the first stage in Fig. 2, $a[0]$ and $a[4]$ are co-controlled as their outputs conflict, but they are independent from $a[1]$ and $a[5]$. Overall, the stages have $m/2, m/4, \ldots, 1$ independent signals, in total $m - 1$ bits.

The inter-lane network is used to realize the length-m NTT permutation, the required transpose in NTT decomposition, and the automorphism. These complement the already supported element-wise operations and matrix/tensor multiplications in the computing lanes, to cover all the FHE operations. Specifically, when performing the decomposed small NTT of length m, only the corresponding DIT or DIF CG stage is activated, while the rest stages all use local direct connections. The CG stage pairs the two elements required in each butterfly operation to adjacent lanes, using a single uniform network to realize the strides in different NTT stages [13], [14]. For transpose and automorphism, our key contribution is to decompose them into a series of shift operations that can be realized using the shift stages. The details will be described in Section IV.

IV. Operation Mapping

This section details our approaches to map NTT and automorphism onto our VPU and realize their permutations on the unified inter-lane network. Similar to previous designs, we apply decomposition to make arbitrarily large operations fit in the size of a VPU. For simplicity, we focus on how to execute the operation on a single VPU. It is easy to extend the mapping to multiple VPUs for parallel execution.

A. NTT Mapping

We decompose a length-N NTT into multiple dimensions each up to length-m, and process the small NTTs of each dimension on the VPS by mapping the m elements to the m lanes and using the CG network for the butterfly operations (Section III-B). If the last dimension size is smaller than m, e.g., $m/2$, the CG network also can be divided into multiple independent groups to allow multiple smaller NTTs to execute in parallel. The element-wise twiddle multiplications between dimensions are done using the element-wise mode of the VPU. Thus the remaining issue is to realize dimension transpose, which can use the inter-lane network.

We use an example in Fig. 3 to illustrate how this can be done. Assume the NTT is decomposed into three dimensions of x, y, z, where x and y have length of 4 (two bits each) matching the number of lanes m, but z is shorter with only length 2 (one bit). Initially dimension x is across lanes, while $z|y$ are within one lane in its register file. We now want to transpose y to the dimension across lanes, i.e., $z|y \times x \rightarrow x|z \times y$ as in Fig. 3(a). Such a transpose can be done by first transforming each column (e.g., $[0, 1, 2, 3]$) to *a diagonal pattern* (as in the middle figure), and then from this diagonal to a row. Specifically, in the first step, we shift down each column by y, so the new lane ID of

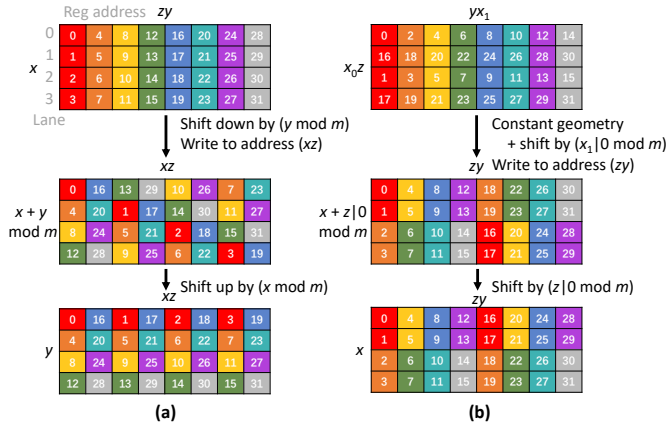

Fig. 3. Dimension transpose in NTT on the inter-lane network. The NTT is decomposed into three dimensions of $x_1|x_0$, $y_1|y_0$, z. (a) shows the transpose from $z|y_1|y_0 \times x_1|x_0$ to $x_1|x_0|z \times y_1|y_0$. (b) shows the transpose from $y_1|y_0|x_1 \times x_0|z$ to $z|y_1|y_0 \times x_1|x_0$.

each element becomes $x + y \bmod m$. For example, the second column $[4, 5, 6, 7]$ has $y = 01_2$, so they are shifted to lanes $1, 2, 3, 0$. We also write them to the register addresses of $x|z$, i.e., the target columns $0, 2, 4, 6$. In the second step, we simply shift up each column by x to make the new lane ID as y. A similar process can transpose $x|z \times y \rightarrow y|x_1 \times x_0|z$. Here because z is shorter than the VPU width, a partial dimension x_0 is also distributed across the lanes to fully utilize the VPU.

However, when we transpose this layout back to the original one, i.e., $y|x_1 \times x_0|z \rightarrow z|y \times x$, we face a different scenario as in Fig. 3(b). In the first column $[0, 16, 1, 17]$, the elements need irregular shift-down distances of $0, 1, 3, 0$. Only using the shift network cannot realize this. Fortunately, we can use the CG network to first reorganize the data, e.g., changing the column to $[0, 1, 16, 17]$, and then shift and write them to proper register addresses as in the middle figure. Then a simple shift could reach the final layout. Note that the shorter z compares to m, the more CG passes would be needed, up to $\log m - 1$ times.

B. Automorphism

We also decompose a large length-N automorphism to $R \times C$ following Section II-C, where $R = m$ is the number of lanes so each column is processed in parallel on all the lanes of the VPU. According to Eq. (3), each column is still in the same column, so after processed it is written to the new column position as a whole. The remaining problem is to realize Eq. (2) on the inter-lane network. While the shift part can be directly supported, the automorphism is challenging.

The key insight is that the first term in Eq. (2) vanishes when we have $\Phi^r \bmod R = 1$. Considering Φ is co-prime with N and N is typically a power of 2, this condition can be satisfied if we have $R = 2$ here. That is, if we recursively decompose the length-R automorphism until length 2, then all the automorphisms would vanish and only the shift operations at these recursive levels remain.

Specifically, for a length-m automorphism, we decompose it into $R' \times C'$ with $C' = 2$. This eliminates the automorphism in Eq. (3), and only leaves one automorphism and one shift in Eq. (2) for each of the two length-$m/2$ sub-columns. The automorphism is recursively decomposed, while the shift can be directly implemented using our inter-lane network. For example, with 8 lanes in Fig. 2, the two sub-columns $[0, 2, 4, 6]$ and $[1, 3, 5, 7]$ can be shifted using the first and second stages. Assume the desired outputs are $[4, 6, 0, 2]$ and $[7, 1, 3, 5]$, i.e., shifting by distances of 2 and 3 (or global distances of 4 and 6), respectively. The first stage shifts both groups, and the second stage only shifts the second group. This is possible with separate control signals for each MUX. The other smaller sub-columns can be similarly handled.

Note that although the recursive decomposition results in multiple shifts, these shifts can be all merged by updating their control signals, i.e., two shifts of distance 2 become one shift of distance 4. By doing so, we could guarantee that *for any length-m automorphism, data only need to go through the inter-lane network once*. However, since these shifts are highly irregular, the shift control signals are also too complicated to be generated at runtime. We thus pre-generate them for all possible automorphisms and store them in on-chip SRAM. With m lanes, there are at most $\frac{m}{2}$ distinct automorphisms with any Φ co-prime with 2, each requiring $m - 1$ bits of control, in total $\frac{m}{2}(m - 1)$ bits. For example, with $m = 64$, we need about 2 kbits, a small area cost.

Now consider the full length-N automorphism. When decomposed into $N = R \times C$ with $R = m$, we still need to combine the length-m automorphism with a length-m shift as in Eq. (2). Their control signals are merged at runtime using some extra simple logic gates. Consequently, *for any automorphism, data only go through the inter-lane network once*.

We open source the control signal generation details at https://github.com/tsinghua-ideal/automorphism-decomposition.

V. Evaluation

A. Methodology

We implement our VPU design, including the lanes and the inter-lane network, in RTL using Verilog. We use $m = 64$ lanes by default, with 64-bit datapaths. We synthesize the design using the ASAP 7 nm library [28], [29]. The design successfully meets the 1 GHz frequency target. The SRAM and register files are modeled with FN-CACTI [30], and scaled to 7 nm following the parameters from [31], [32]. We compare our design with previous FHE accelerators, including F1 [7], BTS [10], ARK [9], and SHARP [12]. BTS, ARK, and SHARP are under 7 nm; F1 uses 14/12 nm, and we scale it to 7 nm. We exclude CraterLake [8] as it uses fixed networks for specific parameters that are not general. However, all these designs use drastically different processing units that are difficult to normalize. As the main goal of this work is to showcase the benefits of our inter-lane network design, to conduct a fair comparison, we port previous designs' approaches to realizing NTT and automorphism into our multi-lane VPU, and replace our inter-lane network. We then report the area and power comparisons for only this data permutation component as well as the full VPU.

TABLE II
AREA AND POWER COMPARISON BETWEEN VARIOUS DESIGNS, ALL WITH 64 LANES.

Design	**Network area** (μm^2 | **ratio**)	**VPU area** (μm^2 | **ratio**)	**Network power** (mW | **ratio**)	**VPU power** (mW | **ratio**)
F1 [7]	55616.42 | 9.40×	300306.61 | 1.20×	93.50 | 6.00×	842.12 | 1.10×
BTS [10]	19405.16 | 3.28×	264095.35 | 1.05×	45.13 | 2.90×	793.75 | 1.04×
ARK [9]	9480.50 | 1.60×	254170.69 | 1.01×	46.35 | 2.97×	794.97 | 1.04×
SHARP [12]	44453.51 | 7.52×	289143.70 | 1.15×	44.04 | 2.83×	792.66 | 1.04×
Ours	5913.62 | 1.00×	250603.81 | 1.00×	15.59 | 1.00×	764.21 | 1.00×

TABLE III
THROUGHPUT UTILIZATION OF NTT AND AUTOMORPHISM IN OUR DESIGN.

N	**NTT**	**Automorphism**
2^{10}	74.77%	100%
2^{12}	85.14%	100%
2^{14}	77.63%	100%
2^{16}	79.96%	100%
2^{18}	81.81%	100%
2^{20}	80.80%	100%

TABLE IV
AREA AND POWER RESULTS OF OUR INTER-LANE NETWORK.

Num. of lanes	**Area** (μm^2)	**Power** (mW)
4	208.99	0.59
8	509.45	1.38
16	1180.83	3.13
32	2664.50	7.02
64	5913.62	15.59
128	12975.47	34.28
256	28226.38	75.02

B. Area and Power

We summarize the area and power comparison in Table II. Specifically, F1 [7] uses a dedicated transpose unit, which mainly consists of a set of quadrant-swap SRAM buffers. An automorphism is done through another cyclic shift network together with the transpose unit. Its area and power are dominated by the SRAM buffers, which are $9\times$ and $6\times$ larger than ours. BTS [10] uses full crossbars (scaled to 64-bit links in our comparison), which are expensive and scale poorly. The area is $3.3\times$ larger than ours, and the power is $2.9\times$ higher. ARK [9] has a dedicated automorphism unit with a special network design, and a dedicated NTT unit with customized NTT connections. These two networks, while area-efficient, are separate and thus have unnecessary overheads, resulting in $1.6\times$ larger area and $3\times$ higher power than our unified inter-lane network design. Finally, SHARP [12] inherits its automorphism unit from ARK, but for NTT it adds the SRAM-based transpose unit similar to F1. The latter structure supports larger NTT lengths at the expense of much larger area, up to $7\times$.

When assessing the total area and power of the VPU, which includes both the lanes (the modular arithmetic units and the register files) and the inter-lane network, we see that the lanes dominate the cost. Nevertheless, the more efficient inter-lane network still exhibits $1.01\times$ to $1.20\times$ area reduction, and $1.04\times$ to $1.10\times$ power reduction. We note that this comparison is conservative. In most of the baselines, the modular arithmetic units for NTT and the element-wise processing units are also separated, resulting in further area and power duplication. While in our design, we reuse the same modular adder/multiplier for both element-wise and NTT operations.

C. Performance

Regarding performance, we evaluate NTT and automorphism operations with various lengths N on our VPU, and compare to an ideal case that fully utilizes all the lanes at all time. We report the throughput utilization, i.e., the actual throughput on our VPU vs. the ideal full throughput, in Table III. For NTTs, transposing the data needs to traverse through the inter-lane network for multiple times, and thus results in throughput loss. Overall, we still achieve about 75% to 85% utilization of the lane logic. Note that when N increases over 2^{12} and 2^{18}, which are integer powers of the number of lanes $m = 64$, the throughput utilization drops, due to one more dimension in the decomposition and thus one more round of transposition. On the other hand, our network always achieves full throughput for automorphism. This is because each data element only goes through the inter-lane network once, thanks to the ability to merge all the shifts in a decomposed automorphism.

D. Scalability

Finally, we examine the scalability of our design, by varying the number of lanes m from 4 to 256, and measuring the area and power in Table IV. We see slightly super-linear scalability for area and power. When increasing from 4 lanes to 256 by $64\times$, the area increases by $135\times$ and the power increases by $127\times$, roughly corresponding to $2.27\times$ and $2.24\times$ growth for every doubling of lane counts.

VI. CONCLUSIONS

In this paper, we propose a unified vector processing unit that is able to support all types of FHE operations. The key innovation lies in how to efficiently support the irregular data permutations among the multiple lanes of the vector processing unit, i.e., NTTs and automorphisms, through the use of an inter-lane network consisting of two constant-geometry stages plus a multi-stage shift network. Compared to previous approaches, our unified design saves significant area and power while maintaining high throughput utilization.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion.

REFERENCES

[1] C. Gentry, "Computing Arbitrary Functions of Encrypted Data," *Communications of the ACM (CACM)*, vol. 53, no. 3, p. 97–105, March 2010.

[2] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *23rd International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT'17)*, 2017, pp. 409–437.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, July 2014.

[4] Z. Brakerski, "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP," in *32nd Annual Cryptology Conference (CRYPTO'12)*, Berlin, Heidelberg, 2012, pp. 868–886.

[5] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," Cryptology ePrint Archive, Paper 2012/144, 2012.

[6] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference," in *27th IEEE International Symposium on High-Performance Computer Architecture (HPCA'21)*, 2021, pp. 26–39.

[7] 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 *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*, 2021, p. 238–252.

[8] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data," in *49th Annual International Symposium on Computer Architecture (ISCA'22)*, 2022, p. 173–187.

[9] 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 *55th IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*, 2022, pp. 1237–1254.

[10] 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 *49th Annual International Symposium on Computer Architecture (ISCA'22)*, 2022, p. 711–725.

[11] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical Homomorphic Encryption Accelerator," in *29th IEEE International Symposium on High-Performance Computer Architecture (HPCA'23)*, 2023, pp. 870–881.

[12] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption," in *50th Annual International Symposium on Computer Architecture (ISCA'23)*, 2023.

[13] M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, p. 252–264, April 1968.

[14] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. C. Cheung, D. Pao, and I. Verbauwhede, "High-Speed Polynomial Multiplication Architecture for Ring-LWE and SHE Cryptosystems," *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, vol. 62, no. 1, pp. 157–166, 2015.

[15] C. Wang and M. Gao, "SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition," in *42nd IEEE/ACM International Conference on Computer Aided Design (ICCAD'23)*, 2023, pp. 1–9.

[16] F. Boemer, S. Kim, G. Seifu, F. D.M. de Souza, and V. Gopal, "Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52," in *9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC'21)*, 2021, p. 57–62.

[17] W. Jung, E. Lee, S. Kim, J. Kim, N. Kim, K. Lee, C. Min, J. H. Cheon, and J. H. Ahn, "Accelerating Fully Homomorphic Encryption through Architecture-Centric Analysis and Optimization," *IEEE Access*, vol. 9, pp. 98 772–98 789, 2021.

[18] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes," in *23rd International Conference on Selected Areas in Cryptography (SAC'16)*, 2017, pp. 423–442.

[19] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A Full RNS Variant of Approximate Homomorphic Encryption," in *25th International Conference on Selected Areas in Cryptography (SAC'18)*, 2019, pp. 347–368.

[20] A. Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme," *IEEE Transactions on Emerging Topics in Computing (TETC)*, vol. 9, no. 2, pp. 941–956, 2021.

[21] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for Approximate Homomorphic Encryption," in *37th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'18)*, 2018, pp. 360–384.

[22] K. Han and D. Ki, "Better Bootstrapping for Approximate Homomorphic Encryption," in *The Cryptographers' Track at the RSA Conference 2020 (CT-RSA'20)*, 2020, pp. 364–390.

[23] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-Sparse Keys," in *40th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'21)*, 2021, pp. 587–617.

[24] D. H. Bailey, "FFTs in External of Hierarchical Memory," in *2nd ACM/IEEE Conference on Supercomputing (SC'89)*, 1989, p. 234–242.

[25] E. Chu and A. George, *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC press, 1999.

[26] T.-W. Sze, "Schönhage-Strassen Algorithm with MapReduce for Multiplying Terabit Integers," in *4th International Workshop on Symbolic-Numeric Computation (SNC'11)*, 2012, p. 54–62.

[27] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An Architecture for Computing on Encrypted Data," in *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020, p. 1295–1309.

[28] L. T. Clark, V. Vashishtha, D. M. Harris, S. Dietrich, and Z. Wang, "Design Flows and Collateral for the ASAP7 7nm FinFET Predictive Process Design Kit," in *2017 IEEE International Conference on Microelectronic Systems Education (MSE'17)*, 2017, pp. 1–4.

[29] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm FinFET Predictive Process Design Kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[30] D. P. Ravipati, R. Kedia, V. M. Van Santen, J. Henkel, P. R. Panda, and H. Amrouch, "FN-CACTI: Advanced CACTI for FinFET and NC-FinFET Technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 3, pp. 339–352, 2022.

[31] IEEE, "International Roadmap for Devices and Systems: 2018," Website, 2018, https://irds.ieee.org/editions/2018.

[32] S. Narasimha, B. Jagannathan *et al.*, "A 7nm CMOS Technology Platform for Mobile and High Performance Compute Application," in *63rd IEEE International Electron Devices Meeting (IEDM'17)*, 2017, pp. 29.5.1–29.5.4.