Morphling: A Throughput-Maximized TFHE-based Accelerator using Transform-domain Reuse

Prasetiyo, Adiwena Putra and Joo-Young Kim
KAIST
Daejeon, Republic of Korea
{pras, adiwena.research, jooyoung1203}@kaist.ac.kr

2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00028

***Abstract*—Fully Homomorphic Encryption (FHE) has become an increasingly important aspect in modern computing, particularly in preserving privacy in cloud computing by enabling computation directly on encrypted data. Despite its potential, FHE generally poses major computational challenges, including huge computational and memory requirements. The bootstrapping operation, which is essential particularly in Torus-FHE (TFHE) scheme, involves intensive computations characterized by an enormous number of polynomial multiplications. For instance, performing a single bootstrapping at the 128-bit security level requires more than 10,000 polynomial multiplications. Our in-depth analysis reveals that domain-transform operations, i.e., Fast Fourier Transform (FFT), contribute up to 88% of these operations, which is the bottleneck of the TFHE system. To address these challenges, we propose Morphling, an accelerator architecture that combines the 2D systolic array and strategic use of transform-domain reuse in order to reduce the overhead of domain-transform in TFHE. This novel approach effectively reduces the number of required domain-transform operations by up to 83.3%, allowing more computational cores in a given die area. In addition, we optimize its microarchitecture design for end-to-end TFHE operation, such as merge-split pipelined-FFT for efficient domain-transform operation, double-pointer method for high-throughput polynomial rotation, and specialized buffer design. Furthermore, we introduce custom instructions for tiling, batching, and scheduling of multiple ciphertext operations. This facilitates software-hardware co-optimization, effectively mapping high-level applications such as XG-Boost classifier, Neural-Network, and VGG-9. As a result, Morphling, with four 2D systolic arrays and four vector units with domain-transform reuse, takes 74.79 mm^2 die area and 53.00 W power consumption in 28nm process. It achieves a throughput of up to 147,615 bootstrappings per second, demonstrating improvements of 3440× over the CPU, 143× over the GPU, and 14.7× over the state-of-the-art TFHE accelerator. It can run various deep learning models with sub-second latency.**

I. Introduction

Privacy-preserving technology has become an increasingly important aspect in current information technology, in which private data are constantly being shared, processed, and stored online. Fully Homomorphic Encryption (FHE) is one of the recent advanced privacy-preserving technologies that enable computations to be performed directly on the encrypted data (ciphertext). One can use this technology to perform the online computation on their sensitive data without disclosing their actual data, thus ensuring the privacy and security of their information. To address privacy concerns, FHE offers a promising solution for secure cloud computing. For example, it can be applied to various interesting applications such as private machine learning [1]–[4], secure genome analysis [5], [6], and secure database application [7], [8].

Despite its great capability, all existing FHE schemes generally pose some major computational challenges when used for real-life applications. First, the computation complexity and the memory requirement are considerably huge in order to perform operations on the ciphertext [9]. For example, when performing operations for a simple Neural Network (NN) with 10 layers using one of the FHE schemes, the number of computations may exceed 11 Giga-Operations (GOps), which is more than 100,000× larger than the computation on the un-encrypted data (plaintext). This significant computational difference is due to the algorithmic characteristics of FHE, which is based on the Learning With Errors (LWE) problem [10]. A single plaintext typically needs to be encrypted into a N-length vector or a N-degree polynomial. As a result, computations need to be performed in vector or polynomial form rather than scalar data, leading to an increase not only in the number of operations but also in the memory footprint.

The second computational challenge is due to the noise feature in LWE-based FHE schemes. To securely hide the data, the noise is added during encryption. Unfortunately, the noise increases every time we perform computations on the ciphertext. At a certain point, the noise level in the ciphertext exceeds the threshold, leading to incorrect computations. In this case, we need to lower the noise level by performing a procedure called bootstrapping. However, the bootstrapping is an expensive operation that typically involves many polynomial operations such as polynomial multiplication, addition, and rotation. Nevertheless, bootstrapping is an essential operation that is crucial in FHE, as its absence would restrict the supported applications.

There are several LWE-based FHE schemes such as BGV [11], BFV [12], CKKS [13], and Torus-FHE (TFHE) [14]. These schemes all rely on the same underlying concept of the LWE problem, but each has its own encryption, decryption, and operation methods. When considering the size of the cryptographic parameters, they can be categorized into two groups: the large-parameter FHE, including CKKS, BGV, and BFV, and the small-parameter FHE, consisting of TFHE (as indicated in Table I). The parameters N, P, and Q, respectively, denote the degree of polynomial, the plaintext space, and the ciphertext space. In this work, we use the TFHE scheme for the following reasons. First, from a utility perspective, TFHE sup-

2378-203X/24/$31.00 ©2024 IEEE
DOI 10.1109/HPCA57654.2024.00028

TABLE I
TYPICAL CIPHERTEXT PARAMETERS FROM DIFFERENT FHE SCHEMES.

Scheme	$\log_2 \lvert P \rvert$	$\log_2 \lvert Q \rvert$	$\log_2 N$
TFHE	$1-8$	32/64	$8-12$
CKKS/BGV/BFV	up to 32	up to 1024	$10-16$

ports a wider variety of operations (e.g., logical and relational operators) compared to other schemes due to its programmable bootstrapping feature. Second, the TFHE construction employs small polynomial coefficients (32-bit or 64-bit), whereas other schemes rely on large polynomial coefficients, such as 1024-bit coefficients. Typically, the Residue Number System (RNS) [15]–[17] is required to decompose large coefficients into smaller ones and then perform the computation on them, making the computation more complex. Conversely, TFHE allows for direct operations on small coefficient ciphertext without the need for RNS operation. To keep the ciphertext parameter small, the TFHE scheme encrypts large-precision plaintext into multiple ciphertexts [18]. From a hardware perspective, the operation can be seen as the computation of multiple small-parameter ciphertexts rather than a single large-parameter ciphertext. This approach enables independent computations on each ciphertext, improving parallelism, data reuse, and simplifying the computation flow.

Recently, there has been significant progress in accelerating FHE computation. While several CKKS-based accelerators [19]–[22] and BGV-based accelerator [23] have been developed, they are characterized by large-parameter ciphertexts and a substantial need for on-chip memory and computational resources due to their large working set. There are some effort to reduce the working set in CKKS schemes such as [24]–[27]. However, these still demand high on-chip memory or otherwise limit the choice of security parameters. Additionally, TFHE-based accelerators have been explored in [28]–[33]. The limitation of some TFHE-based approaches is focus only for a small precision set of TFHE parameters, which can limit the range of supported applications.

In the TFHE scheme, the huge number of polynomial multiplications significantly challenges the overall computation. For example, executing bootstrapping at a 128-bit security level requires more than 10,000 polynomial multiplications. While the polynomial multiplication can be accelerated using transform-domain methods such as Fast Fourier Transform (FFT) or Number Theoretic Transform (NTT), the computational cost of the domain-transform operations, essential for converting data to and from the transform domain, remains considerably high. Our analysis reveals that these operations contribute to the majority of total operations, accounting for up to 88% of the total bootstrapping operations in TFHE. Therefore, besides accelerating the domain-transform operation itself, reducing the number of domain-transform operations is crucial to accelerate the overall computation of TFHE.

Our primary goal in this work is to increase the computational efficiency of the TFHE scheme. We aim to tackle the computational and memory growth, as well as the heavy computational demands in the bootstrapping. To this end, we introduce an accelerator design called Morphling. It is based on the key idea of accelerating multiple polynomial multiplications through transform-domain reuse and is equipped with a specialized computation unit tailored to reduce the number of domain-transform operations. We first identify the compute-intensive and memory-intensive tasks in TFHE, enabling us to allocate efficient computation units accordingly. Our analysis reveals that blind rotation (BR) is the most computationally intensive in TFHE bootstrapping. It involves a series of external product operations, which involve the multiplication between a vector of polynomials and a matrix of polynomials, taking up to 97% of the overall operations. On the contrary, tasks such as modulus switching (MS), key switching (KS), and sample extraction (SE) have a small fraction of operation and tend to be memory-intensive operations. Based on this observation, Morphling has two separate computation units: the external product unit is dedicated to handling blind rotation, specifically external product operations, while the vector processing unit is designed to handle other operations.

In this work, we propose a novel approach that uses a two-dimensional (2D) systolic array of vector processing elements (VPEs) combined with data reuse in the transform domain. This approach can effectively reduce the number of required domain-transform operations by up to 83.3% when performing bootstrapping at a 128-bit security level. One thing to note is that the effectiveness of the reduction depends on the extent to which polynomial operations can be shared and performed concurrently, as well as the utilization of the 2D-systolic VPE array. In addition, we use merge-split pipelined-based FFT to further improve the efficiency of the transform unit. This method not only provides high-throughput FFT computation, but also enables the transformation of two polynomials using just one FFT unit, thereby further reducing the overall number of domain-transform operations. Furthermore, we provide a specialized buffer design and utilize the double-pointer method to improve the rotation of the polynomial during blind rotation, providing a high-throughput rotation. We also provide custom instructions for our computation unit, enabling software-hardware co-optimization, such as tiling, batching, and scheduling. This effectively maps the computation onto the accelerator and improves memory bandwidth utilization.

To evaluate our design, we built a cycle-accurate simulator and RTL model, synthesizing it using TSMC 28 nm technology. We performed application benchmarks using various applications such as XG-Boost classifier, Deep Neural-Network, and VGG-9. Morphling achieves a speed-up of up to $3440\times$ against CPU, $143\times$ against GPU, and up to $14.7\times$ against state-of-the-art accelerator in the bootstraping operation.

In this paper, we present the following contributions:

- We propose a novel approach that combines a two-dimensional systolic array of vector processing elements architecture with the transform-domain reuse, resulting in a significant reduction in the number of required domain-transform operations. Additionally, we provide an

analysis of reuse opportunities and suitable dataflow for our architecture.

- We perform an in-depth analysis of computation types in the TFHE scheme and categorize them into two groups: compute-intensive task (i.e., blind rotation) handled by the external product unit, and memory-intensive tasks (i.e., key-switching, modulus-switching, and sample extraction) performed by the programmable vector processing unit.
- We optimize the microarchitecture design by incorporating various techniques, including merge-split pipelined-Fast Fourier Transform (FFT) architecture for efficient domain-transform operation, the double-pointer method for high-throughput polynomial rotation, and the specialized buffer design.
- We provide our computation unit with custom instructions that enable tiling, batching, and scheduling techniques. This facilitates software-hardware co-optimization, effectively mapping the application onto our accelerator and improving the utilization of memory bandwidth.

II. Background

A. Ciphertext in TFHE

In the TFHE scheme, a message can be various types of data, including boolean, integer, or fixed-point (real) numbers. Before encryption, the scheme performs the encoding of the message, mapping it into the plaintext space. The scheme then encrypts the plaintext into ciphertext, which can be presented in the form of a vector, a polynomial, or a set of polynomials. The ciphertext in TFHE consists of elements called torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, set of real numbers in $[0, 1)$ [34]. In the implementation, these torus elements are actually represented as discretized-torus $\mathbb{T}_q$, a fixed-point representation of the torus $\in \{0, \frac{1}{q} \ldots, \frac{q-1}{q}\}$, where q is the modulus coefficient of ciphertext [34], [35]. There are several types of ciphertext used in TFHE including LWE ciphertext, General-LWE (GLWE) ciphertext, and General-GSW (GGSW) ciphertext. TFHE parameters and notation used in the paper are summarized in Table II.

LWE ciphertext primarily encrypts the scalar type of plaintext or message. If we define n as the dimension of the LWE ciphertext, p as the modulus of the plaintext, and the secret key $\mathbf{s} = (s_1, s_2, \ldots, s_n) \in \mathbb{B}^n = \{0, 1\}^n$, then the LWE ciphertext of the plaintext $m \in \mathbb{T}_p$ is defined by $\boldsymbol{c} = \text{LWE}_{\boldsymbol{s}}(m) = (a_1, a_2, \ldots, a_n, b) \in \mathbb{T}_q^{n+1}$. In this case, $a_1, a_2, \ldots, a_n$ are LWE masks, n random integers sampled from the integer modulo q (i.e., $\mathbb{Z}_q$) that represent the discretized-torus, and $b = \sum_{i=1}^{n} a_i s_i + m + e$, where e is a small noise term. In the implementation, we can represent the LWE ciphertext as $(n + 1)-$ tuple of scalar elements.

GLWE ciphertext is used to encrypt the polynomial type of plaintext or message. We denote the ring polynomial $\mathbb{T}_{(q,N)}[X] := \mathbb{T}_q[X]/(x^N + 1)$ as the polynomial with the modulus coefficient of q and the modulus polynomial of $(x^N + 1)$. If we define k as the dimension of the GLWE ciphertext and the secret key of the polynomials $\boldsymbol{S} = (S_1(x), \ldots, S_k(x)) \in \mathbb{B}_N[X]^k$, the GLWE ciphertext of the plaintext $m \in \mathbb{T}_{(p,N)}[X]$ is defined by $\boldsymbol{C} = \text{GLWE}_{\boldsymbol{S}}(m) = (A_1(x), \ldots, A_k(x), B(x)) \in \mathbb{T}_{(q,N)}[X]^{k+1}$. In this case, $A_1(x), \ldots, A_k(x)$ are GLWE masks and $B(x) = \sum_{i=1}^{k} A_i(x) \cdot S_i(x) + M(x) + E(x)$, where $E(x)$ is the small noise term. In the implementation, we can represent the GLWE ciphertext as $(k + 1)-$ vector of polynomials.

GGSW ciphertext. Let l define the level, the decomposition base β, and the secret key $\boldsymbol{S} = (S_1(x), \ldots, S_k(x)) \in \mathbb{B}_N[X]^k$. The GGSW encryption of plaintext $m \in \mathbb{T}_{(p,N)}[X]$ is the extension of the GLWE ciphertext. The GGSW ciphertext is defined by $\boldsymbol{C} = \text{GGSW}_{\boldsymbol{S}}(m) \in \mathbb{T}_{q,N}[X]^{(k+1)l \times (k+1)}$. In the implementation, we can represent the GLWE ciphertext as $(k + 1) \times (k + 1)l-$ matrix of polynomials.

Several types of ciphertext are used in the TFHE computation. First, the message is encrypted using LWE ciphertext. The Bootstrapping Key (BSK) is derived from the GGSW encryption of the secret key, which is used to encrypt the message. This BSK is then used to reset the noise within the LWE ciphertext. Each secret key in $(s_1, s_2, \ldots, s_n)$, used for message encryption, is encrypted into a GGSW ciphertext. Therefore, the BSK consists of n GGSW ciphertexts: $\text{BSK}_1, \text{BSK}_2, \ldots, \text{BSK}n$ $(\in \mathbb{T}_{q,N}[X]^{(k+1)l_b \times (k+1)})$, where l_b

TABLE II
TFHE parameters and Notations

Symbol	Description
N	Size of polynomial
n	Dimension of LWE ciphertext
k	Dimension of GLWE ciphertext
q	Modulus coefficient of ciphertext
β	Decomposition base
l_b	Bootstrapping key level
l_k	Key-switching key level
BSK_i	Bootstrapping key at iteration-$i \in \mathbb{T}_{q,N}[X]^{(k+1)l_b \times (k+1)}$
ACC_i	Accumulation ciphertext at iteration-$i \in \mathbb{T}_{q,N}[X]^{(k+1)}$
$\text{KSK}_{(i,j)}$	KSK for LWE mask-i and level-$j \in \mathbb{T}_q^{(n+1)}$

Algorithm 1: Programmable Bootstrapping in TFHE

Input: LWE ciphertext $c = (a_1, \ldots, a_n, b) \in \mathbb{T}_q^{n+1}$
Require: BSK, KSK, TP
Output: LWE ciphertext $c'' \in \mathbb{T}_q^{n+1}$

// Mod-switching
$\tilde{c} = (\tilde{a}_1, \ldots, \tilde{a}_n, \tilde{b}) \leftarrow MS(c)$
// Blind rotation
$\text{ACC}_0 \leftarrow \text{TP}$
for $i = 1$ ***to*** n **do**
  // External product
  $\text{ACC}_i \leftarrow \text{BSK}_i \boxdot (X^{\tilde{a}_i} \cdot \text{ACC}_{i-1} - \text{ACC}_{i-1}) + \text{ACC}_{i-1}$
// Sample Extraction
$c' = (a_1', \ldots, a_{kN}', b') \leftarrow SE(\text{ACC}_n)$
// Key-switching
$c'' = (0, \ldots, b') - \sum_{i=1}^{kN} \sum_{j=1}^{l_k} (a_i')_j \cdot \text{KSK}_{(i,j)}$
return c''

is the bsk-level. The Key-Switching Key (KSK) is used in the TFHE scheme to homomorphically change one secret key into another. It consists of $kN \times l_k$ LWE ciphertexts, where $\text{KSK}_{(i,j)} \in \mathbb{T}_q^{n+1}$ and l_k is the ksk-level. Additionally, the Test Polynomial (TP) is a GLWE ciphertext that stores all function values of any function $f(m)$. Lastly, the accumulation ciphertext (ACC) is another GLWE ciphertext that is specifically used to store the polynomial accumulation results during the blind rotation. We refer to ACC_{i-1} as the ACC input and ACC_i as the ACC output during the iteration.

B. Computation in TFHE

The bootstrapping plays a critical role in TFHE, because it not only resets the noise level but also can perform most types of operation in TFHE at the same time. The bootstrapping is basically a homomorphic calculation designed to reset the internal noise level of the ciphertext. In the TFHE scheme, it also evaluates the function $f(m)$ over the torus, which is also known as programmable bootstrapping [14]. It mainly consists of four consecutive steps, MS $\rightarrow$ BR ($n \times$external product) $\rightarrow$ SE $\rightarrow$ KS, as shown in Algorithm 1.

The first step involves modulus switching (line 1), which is used to rescale and round the coefficients of the input ciphertext to a smaller modulo value ($2N$). This operation can be performed as follows: $\tilde{a_i} = \lfloor 2Na_i \rceil_{2N}$ and $\tilde{b} = \lfloor 2Nb \rceil_{2N}$. From a hardware perspective, this operation can be seen as scalar multiplication and rounding applied to all the LWE masks of the input ciphertext.

The blind rotation (line 3-4) involves n sequential external product (line 4) operations, whereas each external product consists of a series of operations: rotation $\rightarrow$ decomposition $\rightarrow$ polynomial multiplication. The rotation operation of the ACC input, denoted as $X^{\tilde{a}_i}\text{ACC}_{i-1}$, involves the rotation of each polynomial in ACC_{i-1}. The decomposition operation is used to limit the noise growth during ciphertext operations, similar to what is used in GGSW ciphertext. The decomposition operation of the polynomial $A_1(x)$ is defined as follows: $\text{Decomp}^{\beta,l}(A_1(x)) = (A_{1,1}(x), \cdots, A_{1,l}(x))$ such that $A_1(x) = A_{1,1}(x)\frac{q}{\beta^1} + A_{1,2}(x)\frac{q}{\beta^2} + \cdots + A_{1,l}(x)\frac{q}{\beta^l}$. In TFHE, the parameters q and the decomposition base β are typically chosen as powers of two. From a computational perspective, this can be viewed as the decomposition of each coefficient of the polynomial, involving operations of bit-slicing and rounding. Next, consider the value of Λ_{i-1}, which is derived from the ACC input and represents the difference between the rotated ACC input and the original. We define $\Lambda_{i-1} = X^{\tilde{a}}\text{ACC}_{i-1} - \text{ACC}_{i-1} = (A_1^{(i-1)}(x), \cdots, A_{k+1}^{(i-1)}(x))$. Then, $\text{Decomp}^{\beta,l_b}(\Lambda_{i-1})$ is expressed in (1). The matrix of polynomials BSK_i is defined in (2). Lastly, the external product between BSK_i and Λ_{i-1} is defined by $\text{BSK}_i \boxdot \Lambda_{i-1} = \text{Decomp}^{\beta,l_b}(\Lambda_{i-1}) \times \text{BSK}_i$, which is equivalent to the multiplication between the vector of polynomials in (1) and the matrix of polynomials in (2). This calculation involves $(k+1) \times (k+1) \times l_b$ polynomial multiplications in total.

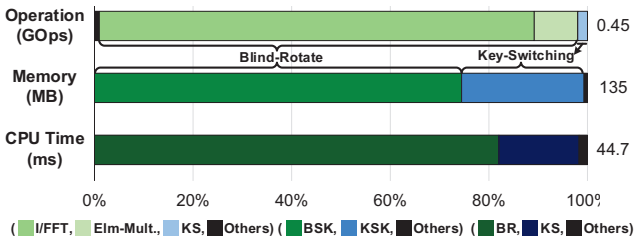

Fig. 1. Operation breakdown of bootstrapping with 128-bit security parameter ($N = 1024, n = 481, k = 2, l_b = 4, l_k = 9$).

$$(\underbrace{A_{1,1}^{(i-1)}(x), \cdots, A_{1,l_b}^{(i-1)}(x)}_{A_1^{(i-1)}(x)}, \cdots, \underbrace{A_{k+1,1}^{(i-1)}(x), \cdots, A_{k+1,l_b}^{(i-1)}(x)}_{A_{k+1}^{(i-1)}(x)}) \quad (1)$$

$$\begin{pmatrix} B_{1,1}^{(i)}(x) & B_{2,1}^{(i)}(x) & \cdots & B_{(k+1),1}^{(i)}(x) \\ B_{1,2}^{(i)}(x) & B_{2,2}^{(i)}(x) & \cdots & B_{(k+1),2}^{(i)}(x) \\ \vdots & \vdots & \ddots & \vdots \\ B_{1,(k+1)l_b}^{(i)}(x) & B_{2,(k+1)l_b}^{(i)}(x) & \cdots & B_{(k+1),(k+1)l_b}^{(i)}(x) \end{pmatrix} \quad (2)$$

Sample extraction (line 5) extracts the LWE ciphertext, c', from the final GLWE ciphertext, ACC_n. This operation involves only memory access and data-regrouping, with essentially no computation needed. Although the noise of the ciphertext has been reset at this point, the resulting ciphertext actually corresponds to a different secret key. The key-switching (line 6) is used to revert the encryption key back to the original. It involves scalar multiplication between the LWE-masks of ciphertext c' and $\text{KSK}_{(i,j)}$, producing the final result of the bootstrapping of c''.

III. Motivation

The computations in current FHE schemes, particularly TFHE, are primarily characterized by a significant number of polynomial multiplications. For instance, performing a bootstrapping in TFHE might involve more than $10,000$ polynomial multiplications. The number of domain-transform operations required to convert both the input and output polynomials would be double this amount, assuming that the BSK is already pre-computed in the transform-domain. To examine which computation is dominant, we first break down the computation workload in TFHE in terms of the number of operations, memory requirements, and execution time when running on a CPU. We analyze the TFHE algorithm using Concrete library [36], a widely used CPU implementation of the TFHE scheme.

Figure 1 shows the breakdown of the bootstrapping computatation with a 128-bit security parameter ($\lambda = 128$). From this graph, several key points can be observed. First, the domain-transform operations, i.e., Inverse FFT or FFT (I/FFT), contribute a large proportion of the overall operation within the blind rotation, accounting for 88% of the total operations. In contrast, key-switching and other operations (decomposition, mod-switching, and sample extraction) represent only a small

fraction of the operations, 1.9% and 1%, respectively. It should be noted that, in this context, an 'operation' is defined as a single multiplication. Second, the memory requirements during blind rotation come primarily from the BSK, taking up 101.4 MB. Likewise, for key-switching, the memory requirement is mainly due to the KSK, taking up 33.8 MB. Third, the execution time on the CPU is primarily dominated by blind rotation, taking 37.7 ms, and key-switching, which takes 6.4 ms. In the blind rotation, the majority of the execution time can be attributed to the time to execute the I/FFT operation. Despite constituting only a small fraction of the overall operation, the key-switching execution time is not negligible, as its execution time is mainly dominated by the process of fetching the KSK from external memory. In fact, when we calculate the computation intensity (Operations/Byte), blind rotation yields a higher value compared to key-switching and other operations. This observation suggests that blind rotation is more computationally intensive dominated by I/FFT operation, whereas key-switching and other operations are more memory intensive. Therefore, it is crucial to reduce the overall domain-transform operations to speed up blind rotation and ultimately the bootstrapping process. In addition, accelerating memory access for the KSK could contribute to improving performance in key switching operation.

To accelerate polynomial multiplication, existing works rely on transform domain methods such as FFT- or NTT-based convolution. The effectiveness of this method generally depends on the amount of data available in the transform-domain, which makes the domain-transform unit a critical component of this method. However, the cost of domain-transform operations, which convert data from/to the transform-domain, remains high. The existing accelerator designs [19]–[25], [28] provide efficient implementations of domain-transform operations through various optimization techniques, such as algorithmic, architectural, data path, and memory access optimizations. While focusing on a single polynomial operation is an effective method, the overall domain-transform operations can be further improved by considering the optimization of multiple polynomial operations through reuse.

In our analysis of TFHE computations, we have identified the potential for operation reuse, which can result in a substantial reduction in the number of domain-transform operations. Specifically, during blind rotation involving BSK and ACC, we found opportunities for reutilization, which effectively minimize these operations. The blind rotation consists of multiplication between a vector of polynomials and a matrix of polynomials, which presents a crucial aspect in the bootstraping operation. The number of reuses depends mostly on the parameters k and l_b, which will be discussed in more detail in Section IV-B. The potential of transform-domain reuse during the blind rotation has not been explored in previous TFHE-based accelerator designs [28]–[33], which designed the architecture optimized for $k = 1$. To support higher precision and a wider range of applications, the accelerator needs to support $k > 1$ in the current TFHE scheme, with a typical value of $k = 1, 2,$ or 3 [36].

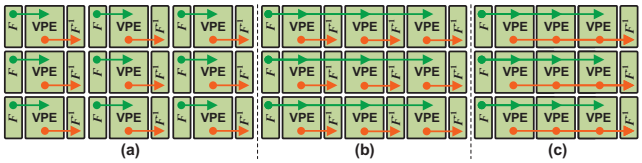

Fig. 2. Illustration of transform-domain reuse on 3×3 VPE Array: (a). **No** transform-domain **Reuse**, (b). **Input** transform-domain **Reuse**, (c). **Input and Output** transform-domain **Reuse**.

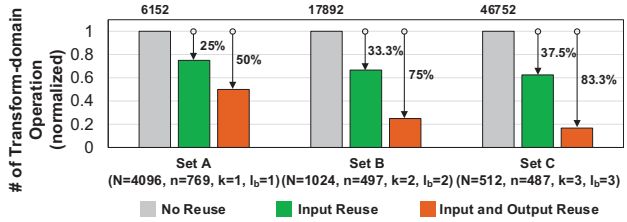

Fig. 3. Reduction in the number of domain-transform operations during bootstrapping from different transform-domain reuse types.

When it comes to data reuse, 2D systolic arrays are known to be an effective architecture to maximize data reuse in applications involving matrix-vector multiplication [37]. The matrix-vector multiplication and the external product exhibit computational similarity, with the main difference being that in the latter, a single scalar element is replaced by a polynomial. To explore this potential, we perform a preliminary study by mapping the blind rotation onto a 2D VPE array. Figure 2 illustrates the mapping of this operation to a 3×3 VPE array for three different architectural types. In this setup, one VPE corresponds to one polynomial multiplication. The ACC input (green line) and the ACC output (orange line) flow through the VPE rows as shown in the figure, while the pre-computed BSK flows through the VPE columns. As the pre-computed BSK does not need domain-transform operations, these details are omitted from the figure for simplicity. In the No-Reuse type, both the input transform (F) and the output transform (F^{-1}) are needed on each VPE. Conversely, in the Input-Reuse type, the input transform is shared across the same VPE rows, but each VPE still requires the output transform. Finally, the Input and Output-Reuse type shares both the input and output transforms across the same VPE rows, reducing domain-transform overhead significantly.

Figure 3 shows the reduction in the number of domain-transform operations when the bootstrapping is mapped to the 4×4 VPE array in different reuse scenarios. From this figure, we can see that bootstrapping could require up to 46752 domain-transform operations. The input transform-domain reuse can reduce the number of domain-transform operations by 25% when the parameters $(k, l_b) = (1, 1)$, and up to 37.5% when the parameters $(k, l_b) = (3, 3)$. Additionally, the input and output transform-domain reuse can potentially further reduce the number of domain-transform operations by up to 83.3% in parameter $(k, l_b) = (3, 3)$. We can observe that as the parameter (k, l_b) increases, the amount of reduction also increases, as it provides more

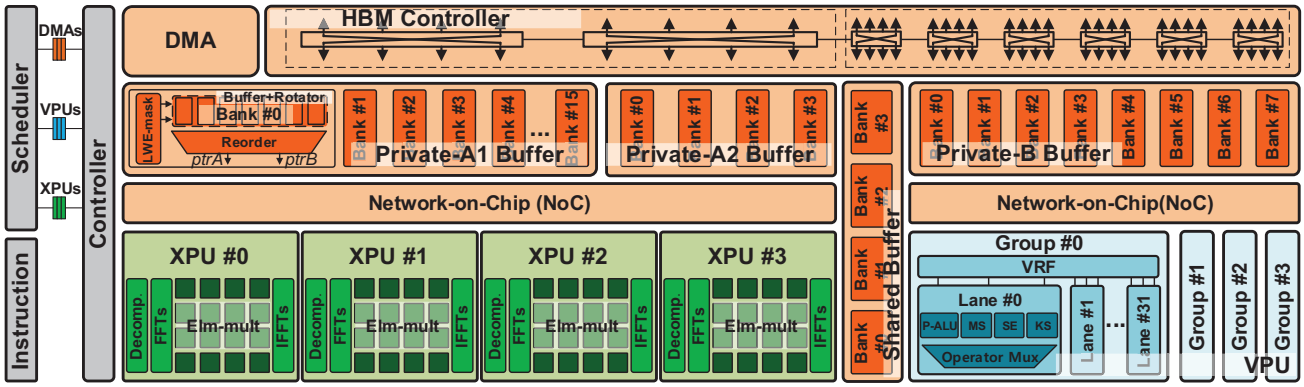

Fig. 4. Overall architecture of Morphling, illustrating its key components and design ideas.

opportunities for reuse. The analysis of this reduction also provides insight into the quantity of reuse that is present during the bootstrapping, which is significantly high. This result motivates us to design an accelerator using a 2D systolic array-type architecture. By combining this with the input and output transform-domain reuse, we aim to achieve a significant reduction in domain-transform operations. As a result, we can accommodate more computational cores with the same resources, potentially leading to a significant improvement in the overall system throughput.

IV. Morphling Architecture

A. Architecture Overview

Figure 4 shows the overall architecture of Morphling. It mainly consists of four external product units (XPUs), one vector processing unit (VPU), specialized buffers (Private-A1, Private-A2, Shared and Private-B buffers), the Network-on-Chip (NoC), the HBM controller, the DMA, and the Scheduler. Morphling adopts a streaming architecture, providing a complete computation pipeline to accommodate all computations inherent in TFHE. The key design features of Morphling are distinguished by a combination of a novel method and optimized processing units, as detailed below. Firstly, the 2D-systolic VPE array is combined with transform-domain reuse within the XPU to minimize the number of domain-transform operations during blind-rotation, as discussed in more detail in Sections IV-B and V-A. Furthermore, Morphling includes the programmable VPU capable of supporting various operations such as polynomial operations (P-ALU), MS, SE, and KS. Morphling incorporates four types of specialized buffers, one of which is integrated with the rotator to enable high-throughput polynomial rotation, as further explained in Section V-C. Lastly, the software-hardware scheduler and three types of instructions for the XPU, VPU, and DMA are provided to facilitate hardware-software co-optimization strategies, discussed in Section V-E.

B. 2D Systolic VPE Array with Transform-domain Reuse

Reuse opportunity during blind rotation. First let us take look on the opportunity reuse that we can exploit. The external product involves three different ciphertexts: the ACC input, the BSK, and the ACC output. Let us further examine the reuse opportunities that exist during this operation, focusing on the discussion from (1) and (2) in Section II-B. To observe the reuse opportunity of the ACC input, we consider the decomposition result in (1), which yields $(k+1)\times l_b$ polynomials. Each polynomial is used for all dot-product computations between this vector of polynomials and all column vectors of the polynomials in BSK_i, allowing each polynomial to be reused up to $k+1$ times. Meanwhile, each polynomial in BSK_i is used exactly once during the external product, which does not give the opportunity for reuse. Even for the different iteration, the different BSK is used, therefore, there is no opportunity reuse for BSK within one ciphertext operation. On the other hand, for each dot-product between vector of polynomial in (1) and every column vector of polynomial in (2), the partial sum of the ACC output is reused $(k+1)\times l_b$ times. This reuse is particularly beneficial when leveraging the linear property of IFFT, as it allows us to perform the accumulation entirely in the transform-domain, hence reducing the number of IFFT operations needed during computation.

VPE array dataflow. To capitalize on these reuse opportunities, we employ a 2D systolic VPE array architecture. Given our design goal of reducing the number of domain-transform operations, we focus on optimizing the reduction from the ACC input and the ACC output transforms. The VPE array is configured to internally compute the transform-domain data by performing domain-transform operations at the input and output of the array. Additionally, each processing element takes a vector element corresponding to the transform-domain data. In VPE array of our design, we have three possible dataflows to determine which data remain inside the VPE: the ACC output stationary, the ACC input stationary, and the BSK stationary. We choose the ACC output stationary dataflow for the following reasons. The ACC output offers more opportunities for reuse than the others, ensuring minimal

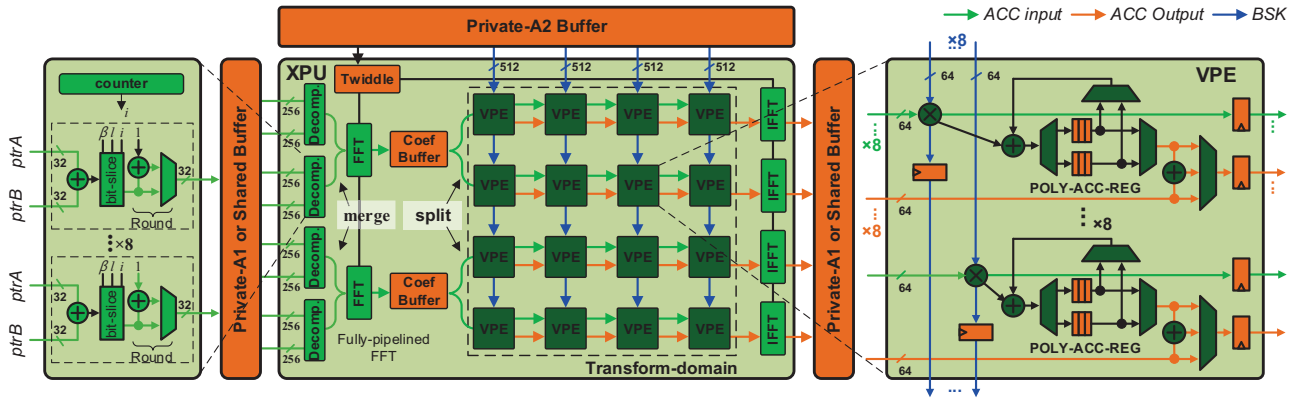

Fig. 5. External Product Unit, consisting of decomposition units, VPE array, and I/FFTs.

data movement. The ACC input stationary and the BSK stationary dataflows would require the partial sum of the ACC output to be stored in Private-A1 buffer. However, given our method to utilize transform-domain reuse, we have to store the transform-domain data instead of polynomial data. This choice doubles the memory requirement for the Private-A1 buffer, thus making these dataflows less preferable. In addition, utilizing the BSK stationary involves more ciphertext, which puts additional pressure on the external memory bandwidth.

C. Improving memory bandwidth utilization for BSK and KSK

Improving memory bandwidth utilization for BSK. In addition to enhancing the computational unit in XPU, we can increase the efficiency of the computation by optimizing the memory access of BSK. Although it does not directly impact the number of domain-transform operation reductions, as BSK is pre-computed, its optimization can enhance external bandwidth utilization. As discussed in Section IV-B, there is no opportunity for BSK reuse within the same ciphertext computation. The only way to reuse BSK is through a different ciphertext computation. In Morphling's architecture, the BSK can be reused in three different ways. First, BSK can be reused within the same XPU through the same VPE column, the number of reuses here being limited by the number of VPE in a column (4 times). Second, BSK can be reused across all XPUs, with the number of possible reuses constrained by the number of XPUs (4 times). Third, BSK can be reused for consecutive streams of input ciphertext, where the number of reuses is limited by the number of ACC ciphertexts that can be stored in the private-A1 buffer (up to 4 times). So in total, we can reuse the BSK for up to 64 different ciphertexts. This strategy reduces the external bandwidth pressure for BSK memory access and allows us to allocate more bandwidth for KSK memory access.

Improving memory bandwidth utilization for KSK. Our solution involves two key strategies. The first strategy is to prioritize channels from DMAs for KSK read whenever the key-switching operation is required, ensuring that the KSK data transfers are given enough external bandwidth. As discussed previously, the key-switching needs to be performed only after the blind-rotation ($n\times$ external-products) has been completed. Given that the ratio of the computation for the external-product is much larger than this key-switching, this strategy does not affect the XPU's computation. The second strategy focuses on reusing KSK for several LWE computations, which requires only the use of a single KSK memory access. This strategy is feasible given that each group of lanes can independently manage distinct operations, which can be handled by the scheduler. Although the KSK reuse factor is limited to only 64 ciphertexts, this strategy helps to reduce the pressure on the external memory bandwidth for KSK.

V. Microarchitecture

A. External Product Unit

Figure 5 shows the detailed microarchitecture inside the XPU. The XPU features the 4 × 4 VPE array, two FFT units, four IFFT units, and four decomposition units. Data processing within the XPU occurs in vector form, where each data path comprises eight elements. The polynomial data are packed in every eight 32-bit coefficients corresponding to the vector $(p_1, p_2, ..., p_8)$, resulting in a 256-bit data path. Conversely, transform-domain data are packed into a 512-bit data path consisting of eight 64-bit complex elements, each with a 32-bit real part and a 32-bit imaginary part. The XPU receives two types of input: ACC input (*ptrA*) and its rotated counterpart (*ptrB*) from the Private-A1 buffer, as well as pre-computed transform-domain data of BSK from the Private-A2 buffer. The following describes the main components within the XPU.

1) Decomposition Unit: It is primarily responsible for computing the decomposition of each vector element and providing a data stream to pipelined FFT. These elements correspond to the polynomial coefficients, which is derived from the result of addition or subtraction operations between *ptrA* and *ptrB*. The decomposition consists of two steps, bit-slicing to each coefficient depending on the decomposition base β and bsk-level l_b, followed by rounding the result. It decomposes $k+1$ polynomials into $(k+1) \times l_b$ polynomials expressed in (1).

2) VPE array with ACC output stationary data-flow: It consists of the 4×4 array of VPEs, each of which performs element-wise multiplication and addition on the vector elements of the ACC input and BSK. The VPE array receives a data stream consisting of the transform-domain data of the ACC input from the FFT, as well as the pre-computed BSK from the Private-A2 buffer. Within the VPE array, the ACC input flows from the left VPE to the right VPE, while the BSK flows from the top VPE to the bottom VPE, as shown in Figure 5. The partial sum of the ACC output stays within the VPE at POLY-ACC-REG until the final ACC output is computed. After this, it flows from the left VPE to the right VPE to IFFT. Each row of VPEs corresponds to the parallel computation of different LWE ciphertexts, all reusing the same BSK. While each column of VPEs corresponds to the computation that involves different columns of the BSK_i, i.e., $B_1^{(i)}$ to $B_{k+1}^{(i)}$ in (2), reusing the same ACC input.

Each VPE corresponds to the dot product between the vector of polynomials in (1) and the column vector of polynomials as given in (2). To illustrate, consider the computations performed within the first column of VPEs. Initially, the VPE calculates the element-wise multiplication between the transform-domain representations of $A_{1,1}(x)$ and $B_{1,1}(x)$ for each group of eight elements in each cycle, storing all the results in POLY-ACC-REG. In the next round, the VPE calculates the element-wise multiplication between the transform-domain representations of $A_{1,2}(x)$ and $B_{1,2}(x)$, and it accumulates this result with the previous one. This process iterates until the element-wise multiplication between the transform-domain representations of $A_{k+1,l_b}(x)$ and $B_{1,(k+1)\times l_b}(x)$ is achieved. The final accumulation result corresponds to the transform domain data of $A_{1,1}(x) \times B_{1,1}(x) + A_{1,2}(x) \times B_{1,2}(x) + \cdots + A_{k+1,l_b}(x) \times B_{1,(k+1)l_b}(x)$. Finally, the results are transformed back into the polynomial using the IFFT operation. Note that the IFFT unit is reused for the computation of all VPEs in the same row. To implement this strategy, Morphling utilizes two instances of POLY-ACC-REG. One register stores the accumulation result, which is queued for IFFT, while the other register computes the subsequent accumulation simultaneously. Additionally, the addition operation located on the right side inside the VPE is utilized to facilitate the accumulation of polynomials from different VPEs on the same VPE row, thus providing flexible mapping to VPEs and improving VPE utilization.

3) Merge-Split Fully-Pipelined FFT (MS-FFT): We leverage the fully-pipelined FFT, specifically the 8-coefficient parallelism of the multi-delay commutator FFT architecture [38]. This FFT architecture is known to provide the high throughput that is required in Morphling's architecture. It consists of all $\log_2 N$ stages of the butterfly unit and Shuffling-Buffers to shuffle the connection of FFT on-the-fly. Therefore, it can provide the transform-domain data to the VPE array in each cycle. The similar nega-cyclic FFT algorithm from [39] is used, which effectively reduces the N-point FFT calculation using only one N/2-point FFT unit. FFT algorithm normally takes complex number (real, imaginary) as the input. However, when the input is only real-part, there is well-known method to compute two real-input FFTs using single FFT operation, leveraging the properties of the FFT and the conjugate symmetry of the FFT for real signals. Since polynomial coefficient does not have imaginary part, we can use this technique to improve the computation efficiency. First, we merge two polynomials into one polynomial by interleaving them to put the first polynomial into the real-part and the second polynomial into the imaginary-part. Then, it calculates the FFT for this new polynomial. To obtain the final result, we must split the FFT output into two sections. An additional buffer, the Coef buffer, is required to store the half coefficients of the FFT result, along with the addition and shifter. With only minimal hardware overhead, we can effectively double the throughput of FFT.

B. Vector Processing Unit

The VPU is organized into four lane-groups (Group #1 through Group #4), each comprising 32 lanes that correspond to a 32-element of a vector, as shown in Figure 4. Each lane consists of processing elements such as multipliers, adders, and shifters. Their datapath can be programmed to handle different computations such as KS, MS, SE, P-ALU and scalar multiplication. The Vector Register File (VRF) is utilized in each lane-group, storing three vector operands as input vectors for the VPU lanes. To accommodate various types of computation, each group can be programmed individually based on the scheduled computations. This feature provides the VPU with the flexibility to concurrently support different types of operations. To enhance efficiency, the VPU is used to process computations across all XPUs, given that operations apart from blind rotation consume only a minor portion of the computational resources.

C. On-chip Memory and Rotator

Morphling features a two-level memory architecture. The first-level consists of four different types of multibank memory: Private-A1, Private-A2, Private-B, and Shared Buffer as shown in Figure 4. Firstly, the Private-A1 buffer is specifically designed to support rotator operations. It stores the ACC ciphertexts during the blind-rotation performed by XPUs and also stores the LWE-masks required for the rotation. Secondly, the Private-A2 buffer is used to store the transform-domain data from BSK and the twiddle factor. This buffer mainly serves as a double buffer, functioning as a pre-fetcher to hide the latency of memory access for the BSK_i needed in the next iteration. Thirdly, the shared buffer is utilized to store the blind-rotation results from the XPUs, which is used for subsequent computations by the VPU, and vice versa. This approach ensures that computations can continue concurrently on both the XPU and VPU, effectively decoupling their operations. Lastly, the Private-B buffer is exclusively utilized by the VPU to store data such as LWE ciphertext, KSK, vector and scalar operands, as needed by specific applications. The second-level includes the POLY-ACC register within the VPE, the VRF within the VPU, and buffers related to I/IFFT operation (Twiddle, Coef, and Shuffle Buffers).

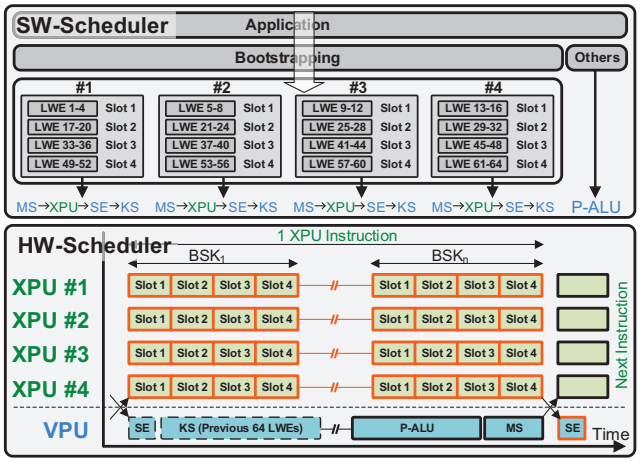

Fig. 6. Illustration of the SW-HW co-optimization strategy.

The rotation operation within the Private-A1 buffer. The rotation, performed during the external product, needs to access both the ACC input (ACC_{i-1}) and its rotations ($X^{\tilde{a}_i} \cdot \text{ACC}_{i-1}$). We could simply perform the rotation using a variable delay or shifter in XPU. However, this method is less effective because the delay varies depending on the LWE mask ($\tilde{a}_i$), which differs for each ciphertext. This causes variable latency in the computation pipeline, which leads to pipeline stalls, consequently degrading the overall system performance. To avoid this problem, we move the rotation inside the Private-A1 buffer as shown in Figure 4. We employ a double-pointer strategy for this purpose: the first pointer accesses the original polynomials from ACC_{i-1}, while the second pointer accesses the rotated polynomials from $X^{\tilde{a}_i}\text{ACC}_{i-1}$. This strategy is feasible because Morphling employs the ACC output stationary dataflow. The ACC is updated and stored in the POLY-ACC Register within the VPE, while the ACC in the Private-A1 buffer remains the same until the next accumulation polynomial is retrieved, which occurs after $(k+1) \times l_b$ polynomial multiplication. The LWE-mask unit generates the addresses of the data across the buffer based on the mask value ($\tilde{a}_i$). Given that the coefficients are packed and tiled into a vector with each element of the vector placed in a fixed and different buffer location, the reorder unit is used to support unaligned vector access. It enables rotation of any value of $\tilde{a}_i$. As a result, this strategy provides high-throughput rotation, maintaining a constant data stream to pipelined-FFT on each cycle and enabling continuous operation of the XPUs, thereby optimizing the overall system performance.

D. Network-on-Chip

Morphling utilizes the 2D systolic VPE array for primary computations and the specialized buffer design, both of which significantly enhance the NoC implementation. The 2D systolic VPE array contributes to the reduction of bandwidth requirements for the NoC by efficiently moving the data from one VPE to another, minimizing the data movement between Buffers and XPUs. Meanwhile, the specialized buffer design provides a fixed and predictable data-flow during computation. This facilitates the optimization of the NoC to meet specific requirements. For instance, the connection between the Private A1-buffer and the XPUs in Figure 4 is structured as a four-to-four crossbar topology, accommodating the connection of every set of four banks to four XPUs. This configuration is also used for the connections between the XPUs and the Shared buffer, the Shared buffer and the VPU, and the Private-B buffer and the VPU. Additionally, the connection from the Private-A2 buffer to the XPUs adopts a multicast topology, with each bank establishing a multicast connection to four XPUs. Given that the Private-A2 buffer is exclusively used to provide the BSK and twiddle factors, it requires only a one-directional connection. Overall, the NoC is capable of supporting a chip-wide bandwidth of 4.8 TB/s for computation.

E. SW-HW Co-optimized through Schedulling

Figure 6 illustrates the SW-HW co-optimization strategy, which consists of SW-scheduler and HW-scheduler. The SW-scheduler handles computation scheduling at the application level by batching or tiling operations within the application and then generates an instruction stream for the hardware. The HW-scheduler continually monitors the status of all computation units, assigning tasks based on the instruction stream to available resources. In the SW-scheduler part, the operations inside the application are grouped according to whether they require bootstrapping or not. The scheduler then further organizes the bootstrapping tasks, grouping every 64 LWE ciphertexts into four groups. The operations can be parallelized within different groups. For example, in the figure, LWEs 1-to-16 can be computed at the same time through different XPUs, while operations that have dependencies can be assigned to different slots within the same group. Note that if the ciphertexts are encrypted using different secret keys, they should be placed in the different group. The bootstrapping tasks within the same group are computed using a single dependent instruction stream, VPU(MS) $\rightarrow$ XPU $\rightarrow$ VPU(SE) $\rightarrow$ VPU(KS). The HW-scheduler recognizes this instruction dependency, allowing it to schedule the computation accordingly, as shown in the figure. In addition, DMA instructions are also generated to assign data from memory to the correct buffer location. This strategy, enabling tiling and batching as discussed in Section IV-C, can be applied to any application. It effectively maps the application onto the accelerator, thereby improving memory access for both BSK and KSK.

VI. Evaluation

A. Experimental Methodology

To evaluate our design, we implemented the Morphling architecture in RTL and synthesized it to estimate the area and power. We also developed a cycle-accurate simulator to estimate the performance of our design. We use the TFHE parameters listed in Table III, which are chosen from the original implementation of the TFHE schemes [36], [40]. Parameter sets I-IV with $k = 1$ are used to compare the result with

TABLE III
TFHE PARAMETER SET FOR EXPERIMENTS.

Parameter Set	N	n	k	l_b	λ
I	1024	500	1	2	80-bit
II	1024	630	1	3	110-bit
III	2048	592	1	3	128-bit
IV	2048	742	1	1	128-bit
A	4096	769	1	1	128-bit
B	1024	497	2	2	128-bit
C	512	487	3	3	128-bit

previous TFHE-based accelerators, while parameter sets A-C are used for other experiments. We evaluate our design using application workloads that have been implemented in TFHE, including the XG-Boost classifier [41], [42], DeepCNN-20, DeepCNN-50, DeepCNN-100 [42], and VGG-9 [43]. We also compared the experimental results of our design against 64 CPU cores of Intel (R) Xeon (R) Gold 6226R @2.9 GHz, with 216 GB of RAM. More details on the benchmark models are given below.

- The XG-Boost classifier is a tree-based model used for classification. The model consists of 100 estimators, with a maximum tree depth of six, as used in [41], [42].
- The DeepCNN-X model [42], with configurations X=20, 50, 100, takes an $8 \times 8 \times 1$ input size. The first layer is the 3×3 convolution (CONV) layer with filter size of 2, followed by another 3×3 CONV layer with filter size of 92 and stride of 2. The next X layers are 1×1 CONV layers, each with a filter size of 92. The last CONV layer is a 2×2 CONV layer with a filter size of 16, followed by fully connected (FC) layer with 10 neurons.
- The architecture of the VGG-9 model [43] is used for the CIFAR-10 image classification. The model takes an input image size of $32 \times 32 \times 3$. The six CONV layers use a kernel size of 3×3. The filter sizes of the six CONV layers are 64, 64, 128, 128, 256, and 256 filters, respectively. The 2×2 average pooling is applied after the second and fourth CONV layers. The first two FC layers consist of 512 neurons, and the last FC layer consists of 10 neurons for classification.

B. Hardware Modelling

Morphling operates at a clock frequency of 1.2 GHz in TSMC 28 nm technology. The implementation consists of four XPUs with each having 4×4 VPE array, VPU with 4 groups of 32 lanes, 16-bank of Private-A1 buffer with total 4 MB, 4-bank of Private-A2 buffer with total 4 MB, 4-bank of Shared buffer with total 1 MB, and 8-bank of Private-B buffer with total 2 MB, same as shown in Figure 4. Table IV shows the area and power breakdown of Morphling. We utilize one HBM2e stack for external memory bandwidth, with the power and area of HBM2e PHY estimated from [44]. One HBM2e stack consists of 8 channels, and we assume a moderate average bandwidth of 310 GB/s for the simulation. We prioritize 6 HBM channels for VPU and 2 HBM channels for XPU during the computation. Morphling has the overall area of 74.79 mm^2 and the total power consumption of 53.00 W.

TABLE IV
THE AREA AND POWER BREAKDOWN OF MORPHLING.

Component	Area (mm^2)	Power (W)
$4\times$ Decomposition Unit	0.01	$<$0.01
$2\times$ FFT	1.22	0.91
$2\times$ Coef-Buffer	0.06	0.03
Twiddle-Buffer	0.75	0.37
4×4 VPE Array	4.71	3.13
$4\times$ IFFT	2.45	1.82
XPU	9.23	6.23
$4\times$ XPU	36.95	25.11
VPU	0.22	0.13
NoC	0.21	0.17
Private-A1 Buffer (4 MB)	8.31	4.27
Private-A2 Buffer (4 MB)	8.10	3.99
Private-B Buffer (2 MB)	4.05	2.42
Shared Buffer (1 MB)	2.02	0.99
HBM2e PHY	14.90	15.90
Total	74.79	53.00

TABLE V
COMPARISON OF BOOTSTRAPPING LATENCY AND THROUGHPUT ACROSS VARIOUS IMPLEMENTATION PLATFORMS.

	Platform	Area (mm^2)	Power (W)	Parameter Set	Latency (ms)	Throughput (BS/s)
Concrete [40]	CPU	–	–	I	15.65	63
				II	27.26	36
				III	82.19	12
NuFHE [45]	GPU	–	–	I	240.00	2,500
				II	420.00	550
cuda TFHE [46]	GPU	–	–	IV	66.00	1,786
XHEC [31]	FPGA	–	–	I	$\sim$1.15	4,000
				II	$\sim$1.65	2,800
MATCHA [28]	ASIC (16 nm)	36.96	39.98	I	0.20	10,000
Strix [33]	ASIC (28 nm)	141.37	77.14	I	0.16	74,696
				II	0.23	39,600
				III	0.44	21,104
Morphling	ASIC (28 nm)	74.79	53.00	I	0.11	147,615
				II	0.20	78,692
				III	0.38	41,850
				IV	0.16	98,933

C. Result and Discussion

Latency and Throughput. Table V presents the latency and throughput measurements of our design and other implementations on various platforms. Throughput is quantified as the number of bootstrapping operations per second (BS/s). The results indicate that Morphling significantly outperforms other implementations. Specifically, Morphling achieves a speed-up of $2145-3439\times$ over Concrete (CPU-based), $60-144\times$ over NuFHE (Nvidia Titan RTX), $55\times$ over cuda TFHE (Nvidia Tesla V100S), and $28-37\times$ over XHEC (FPGA-based). While it shows modest latency improvements of $1.1-1.8\times$ over recent ASIC implementations (MATCHA and Strix), Morphling achieves significant throughput improvement by $1.98-14.76\times$. The enhancement in performance is attributed to the increased number of bootstrapping cores, which is made possible by the utilization of transform domain reuse, even

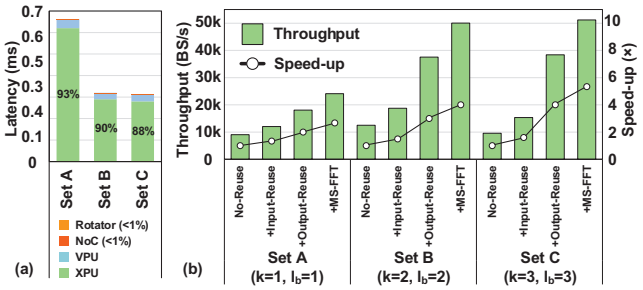

Fig. 7. Computational breakdown of Morphling: (a) Latency across the components, (b) Throughput and speed-up across different types of transform-domain reuse architecture.

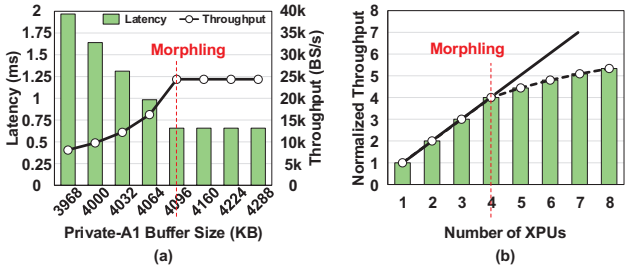

Fig. 8. Architectural analysis of Morphling: (a) Impact of on-chip memory size on latency and throughput, (b) Impact of number of XPUs on throughput.

with fewer I/FFTs. In fact, MATCHA and Strix use 40 and 32 I/FFTs, respectively, each corresponding to 8 bootstrapping cores. MATCHA belongs to the No-Reuse type, while Strix belongs to the Input-Reuse type, and both are optimized for $k = 1$. In constrast, Morphling employs 24 I/FFTs, which correspond to 16 bootstrapping cores, and belongs to Input and Output-Reuse type. Note that these comparisons are based on a small parameter $k = 1$. We expect more pronounced improvements as the value of k increases.

Latency breakdown. Figure 7-a presents the latency breakdown of bootstrapping operation across different components. The results indicate that the XPU dominates the computation, accounting for $88 - 93\%$ of the total latency. As Morphling employs a pipelined architecture, the overall throughput is primarily determined by the XPU. Of particular note, the VPU is a programmable unit. While waiting for results from the XPU, it can be scheduled to execute non-bootstrapping operations using P-ALU instructions, as illustrated in Figure 6.

Transform-domain reuse impact on the performance. Figure 7-b shows the performance and speed-up breakdown results for different types of transform domain reuse architectures. Initially, we implement No-Reuse, Input-Reuse, and Input and Output-Reuse type architectures as discussed in Section III. We set all architecture types to have the same computing resources, and then measure the throughput using the same bootstrapping operation. We set the No-Reuse type as the baseline of the speed-up. As seen in the figure, the input-reuse type provides the speed-up of $1.3 - 1.6\times$ over the No-Reuse type architecture, while the Input and Output-Reuse type further speeds up performance by $2 - 3.9\times$ compared to the No-Reuse type architecture. As the parameters k and l_b increase, the improvement also increases to $2\times$ in set A, $2.9\times$ in set B, and $3.9\times$ in set C. Moreover, the use of the merge-split FFT improves performance by $1.2 - 1.3\times$. Overall, combining all design techniques gives a performance improvement by $2.6-5.3\times$ speed-up. This result suggests that with the same I/FFT resources, we can actually achieve larger performance by taking advantage of transform-domain reuse.

Architectural analysis. Figure 8 analyzes the impact of the on-chip memory size on latency and throughput, as well as the impact of the number of XPUs on throughput. In the experiment portrayed in Figure 8-a, we fix the size of the Private-A2 buffer to hold the minimum amount required for BSK_i, and also set the size of the Private-B and Shared buffers to a fixed value. Then, we vary the size of Private-A1 around 4096KB, as depicted in the figure. The performance of Morphling degrades when the Private-A1 buffer size falls below 4096 KB, and the performance stabilizes for buffer sizes greater than 4096 KB. This value actually corresponds to the minimum size of the Private-A1 Buffer required to hold the minimum ACC ciphertexts for the given external memory bandwidth. Therefore, Morphling sets the Private-A1 buffer size to 4096 KB. In the experiment portrayed in Figure 8-b, we fix the size of the private-A1 buffer 4096 KB and vary the number of XPUs. We observe that performance increases linearly until the number of XPUs reaches 4, after which performance starts to degrade with any further increase in XPU count. This is because when the number of XPUs is larger than four, the performance becomes limited by the given external memory bandwidth. In this case, increasing the number of XPUs is no longer effective in increasing the performance. Therefore, Morphling is set to utilize four XPUs, which is the optimized configuration for the given external memory bandwidth and the on-chip memory size. The number of XPUs can be effectively scaled up by increasing either the external bandwidth or the size of Private-A1 buffer.

TABLE VI
COMPARISON OF MORPHLING EXECUTION TIME WITH CPU ACROSS VARIOUS APPLICATIONS.

	CPU (s)	Morphling (s)	Speed-up
XG-Boost	9.59	0.06	144 ×
DeepCNN-20	33.32	0.34	95×
DeepCNN-50	74.94	0.84	88×
DeepCNN-100	180.09	1.72	104×
VGG-9	94.78	0.67	140×

Application Benchmarks. Table VI shows the result of the execution time of Morphling compared to the CPU when running several applications described in Section VI-A. In the XGBoost classifier, bootstrapping is utilized during comparison operations, while in the DeepCNN and VGG-9 models, bootstrapping is used during the execution of ReLU activation functions. Morphling achieves an execution time of

0.06 s, which is 144× faster than the CPU execution time when running the XGBoost classifier. The execution time of Morphling is 0.34s, 0.84s, and 1.72s for DeepCNN-20, DeepCNN-50, and DeepCNN-100, respectively. This result achieves a speed-up of $88 - 111\times$ compared to the CPU. Although these DeepCNN models have a relatively small input feature size and single 1×1 convolution layer, these DeepCNN models include deep layers (20, 50, 100 layers), each with a filter size of 92, which requires 368 ReLU operations. For the VGG-9 model, Morphling outperforms the CPU execution time, with an execution time of 0.675 s, which is $140\times$ faster. Although the model has only 9 layers, it has a relatively larger input feature size with larger filter sizes (64-to-256) in each layer. Overall, Morphling achieves $88 - 144\times$ speed-up over CPU when running various application workloads.

VII. Related Works

BGV/BFV-based accelerator Early efforts to accelerate FHE were largely centered on the BGV/BFV scheme [11], [12], [47], [48]. Notable work has been conducted with FPGAs, including studies like HEPCloud [49], HEAT [50], and HEAWS [51], which emphasize the Somewhat Homomorphic Encryption (SHE) version of BFV/FV that lacks support for the bootstrapping mechanism. The GPU-based implementations has also been explored as seen in [52]–[55]. Cheetah [56], one of the ASIC accelerators, introduces a framework that incorporates Homomorphic Encryption (HE) and Multi-Party Computation (MPC) for private inference. However, similar to HEAT, it does not facilitate bootstrapping. The first accelerator for the BGV/BFV scheme that comprehensively supports bootstrapping is F1 [23]. While F1 achieves state-of-the-art performance for BGV/BFV scheme acceleration, it is tailored for shallow computation with a small ciphertext size and does not target private deep neural networks.

CKKS-based accelerator. The research on CKKS-based accelerators began with HEAX [19], BTS [21], CraterLake [20], and ARK [22] as well as the GPU implementation [57], focusing on enhancing performance and bootstrapping acceleration. However, this approach resulted in a large on-chip memory requirement (> 500 MB). Subsequent efforts aimed to reduce the on-chip memory requirement (< 50 MB) through efficient scheduling and data reuse in FPGA or GPU [24]–[26]. Nonetheless, their performance slower than earlier ASIC solutions and provided limited security parameter choices. The latest work, SHARP [27], introduced a novel approach by shortening the bit word length to 36 bits, resulting in faster performance than prior ASIC solutions, although still requiring almost 200 MB of on-chip memory.

TFHE-based accelerator. Various platforms have been explored to accelerate the TFHE scheme, including GPU [40], [45], [46], FPGA [29]–[32], and ASIC [28], [33]. Early FPGA attempts lacked transformation techniques for polynomial multiplication, resulting in performance inferior to optimized CPU implementations [32]. Subsequent FPGA research focused on scheduling and data reuse techniques, but TFHE's inherent sequential nature limited FPGA's performance potential [29], [30], [58]. MATCHA, the first ASIC implementation, prioritized bootstrapping key unrolling [59], [60] and optimized FFT circuits for superior performance. However, MATCHA's limitation was its focus on boolean-TFHE with a small polynomial degree ($N = 1024$). Strix addressed this limitation with two-level batching, achieving faster performance than MATCHA. Nonetheless, Strix's design resulted in poor domain transform reuse and required additional FFT units [33].

VIII. Conclusion

In this work, we address computational challenges in Fully Homomorphic Encryption (FHE), specifically the bootstrapping operation in the TFHE scheme. We introduce transform-domain reuse to minimize the domain-transform operations prominent in bootstrapping. By strategically incorporating transform-domain data reuse into our 2D systolic VPE array architecture, we effectively reduce domain-transform operations, enabling higher computational core allocation and system throughput enhancement. Moreover, we optimize at the microarchitecture level with techniques such as merge-split pipelined-FFT, the double-pointer method, and specialized buffer design. We further enhance the efficiency of our design with SW-HW co-optimization techniques such as tiling, batching, and scheduling. Morphling achieves a speedup of $3440\times$, $143\times$, and $14.7\times$ over the CPU, GPU, and state-of-the-art accelerator, respectively, in the bootstrapping operation, and a speedup of $88 - 144\times$ over the CPU in various applications.

Acknowledgment

This work was supported in part by the Ministry of Science and ICT (MSIT), South Korea, under the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant No. 2022-0-01037 and grant No. 2022-0-01036 and in part by Samsung Electronics. We thank the anonymous reviewers for their valuable comments and feedback. Prasetiyo would like to thank the Hyundai Motor Chung Mong-Koo (CMK) Global Scholarship for their scholarship support.

References

[1] D. Kim and C. Guyot, "Optimized privacy-preserving cnn inference with fully homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, 2023.

[2] J.-W. Lee, H. Kang, Y. Lee, W. Choi, J. Eom, M. Deryabin, E. Lee, J. Lee, D. Yoo, Y.-S. Kim, and J.-S. No, "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network," *IEEE Access*, vol. 10, pp. 30 039–30 054, 2022.

[3] S. Meftah, B. H. M. Tan, C. F. Mun, K. M. M. Aung, B. Veeravalli, and V. Chandrasekhar, "Doren: toward efficient deep convolutional neural networks with fully homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3740–3752, 2021.

[4] J. Park, D. S. Kim, and H. Lim, "Privacy-preserving reinforcement learning using homomorphic encryption in cloud computing infrastructures," *IEEE Access*, vol. 8, pp. 203 564–203 579, 2020.

[5] M. Kim and K. Lauter, "Private genome analysis through homomorphic encryption," in *BMC medical informatics and decision making*, vol. 15, no. 5. BioMed Central, 2015, pp. 1–12.

[6] J. Zhou, B. Lei, H. Lang, E. Panaousis, K. Liang, and J. Xiang, "Secure genotype imputation using homomorphic encryption," *Journal of Information Security and Applications*, vol. 72, p. 103386, 2023.

[7] S. Zhang, S. Ray, R. Lu, Y. Guan, Y. Zheng, and J. Shao, "Efficient and privacy-preserving spatial keyword similarity query over encrypted data," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[8] O. Timothy Tawose, J. Dai, L. Yang, and D. Zhao, "Toward efficient homomorphic encryption for outsourced databases through parallel caching," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–23, 2023.

[9] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi, "Does fully homomorphic encryption need compute acceleration?" *arXiv preprint arXiv:2112.06396*, 2021.

[10] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer, 2013, pp. 75–92.

[11] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[12] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[13] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.

[14] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer, 2016, pp. 3–33.

[15] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the aes circuit," in *Annual Cryptology Conference*. Springer, 2012, pp. 850–867.

[16] S. Halevi, Y. Polyakov, and V. Shoup, "An improved rns variant of the bfv homomorphic encryption scheme," in *Topics in Cryptology–CT-RSA 2019: The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4–8, 2019, Proceedings*. Springer, 2019, pp. 83–105.

[17] 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.

[18] L. Bergerat, A. Boudi, Q. Bourgerie, I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Parameter optimization and larger precision for (t) fhe," *Journal of Cryptology*, vol. 36, no. 3, p. 28, 2023.

[19] 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.

[20] 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 *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[21] 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*, 2022, pp. 711–725.

[22] 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)*. IEEE, 2022, pp. 1237–1254.

[23] 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*, 2021, pp. 238–252.

[24] 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)*. IEEE, 2023, pp. 870–881.

[25] 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)*. IEEE, 2023, pp. 882–895.

[26] S. Fan, Z. Wang, W. Xu, R. Hou, D. Meng, and M. Zhang, "Tensorfhe: Achieving practical computation on encrypted data using gpgpu," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2023, pp. 922–934. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA56546.2023.10071017

[27] 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 *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[28] L. Jiang, Q. Lou, and N. Joshi, "Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 235–240.

[29] T. Ye, R. Kannan, and V. K. Prasanna, "Fpga acceleration of fully homomorphic encryption over the torus," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2022, pp. 1–7.

[30] M. Van Beirendonck, J.-P. D'Anvers, and I. Verbauwhede, "Fpt: a fixed-point accelerator for torus fully homomorphic encryption," *Cryptology ePrint Archive*, 2022.

[31] K. Nam, H. Oh, H. Moon, and Y. Paek, "Accelerating n-bit operations over tfhe on commodity cpu-fpga," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3508352.3549413

[32] S. Gener, P. Newton, D. Tan, S. Richelson, G. Lemieux, and P. Brisk, "An fpga-based programmable vector engine for fast fully homomorphic encryption over the torus," in *SPSL: Secure and Private Systems for Machine Learning (ISCA Workshop)*, 2021.

[33] A. Putra, Prasetiyo, Y. Chen, J. Kim, and J.-Y. Kim, "Strix: An end-to-end streaming architecture with two-level ciphertext batching for fully homomorphic encryption with programmable bootstrapping," *arXiv preprint arXiv:2305.11423*, 2023.

[34] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[35] M. Joye, "Guide to fully homomorphic encryption over the [discretized] torus," *Cryptology ePrint Archive*, 2021.

[36] Zama, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data," 2022, https://github.com/zama-ai/tfhe-rs.

[37] R. Xu, S. Ma, Y. Guo, and D. Li, "A survey of design and optimization for systolic array based dnn accelerators," *ACM Computing Surveys*, 2023.

[38] M. Garrido, "A survey on pipelined fft hardware architectures," *Journal of Signal Processing Systems*, vol. 94, no. 11, pp. 1345–1364, 2022.

[39] J. Klemsa, "Fast and error-free negacyclic integer convolution using extended fourier transform," in *International Symposium on Cyber Security Cryptography and Machine Learning*. Springer, 2021, pp. 282–300.

[40] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap, "Concrete: Concrete operates on ciphertexts rapidly by extending tfhe," in *WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2020.

[41] J. Frery, A. Stoian, R. Bredehoft, L. Montero, C. Kherfallah, B. Chevallier-Mames, and A. Meyre, "Privacy-preserving tree-based inference with fully homomorphic encryption," *arXiv preprint arXiv:2303.01254*, 2023.

[42] A. Meyre, B. Chevallier-Mames, J. Frery, A. Stoian, R. Bredehoft, L. Montero, and C. Kherfallah, "Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists," 2022, https://github.com/zama-ai/concrete-ml.

[43] A. Stoian, J. Frery, R. Bredehoft, L. Montero, C. Kherfallah, and B. Chevallier-Mames, "Deep neural networks for encrypted inference with tfhe," in *International Symposium on Cyber Security, Cryptology, and Machine Learning*. Springer, 2023, pp. 493–500.

[44] JEDEC, "High bandwidth memory (hbm) dram," JEDEC, Tech. Rep. JESD235D, 2021.

[45] nucypher. (2020) Nufhe, a gpu-powered torus fhe implementation. [Online]. Available: https://github.com/nucypher/nufhe

[46] S. Narisada, H. Okada, K. Fukushima, S. Kiyomoto, and T. Nishide, "Gpu acceleration of high-precision homomorphic computation utilizing redundant representation," *Cryptology ePrint Archive*, 2023.
[47] A. C. Mert, E. Öztürk, and E. Savaş, "Design and implementation of encryption/decryption architectures for bfv homomorphic encryption scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 353–362, 2019.
[48] Y. Su, B. Yang, C. Yang, and L. Tian, "Fpga-based hardware accelerator for leveled ring-lwe fully homomorphic encryption," *IEEE Access*, vol. 8, pp. 168 008–168 025, 2020.
[49] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "Hepcloud: An fpga-based multicore processor for fv somewhat homomorphic function evaluation," *IEEE Transactions on Computers*, vol. 67, no. 11, pp. 1637–1650, 2018.
[50] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 387–398.
[51] F. Turan, S. S. Roy, and I. Verbauwhede, "Heaws: An accelerator for homomorphic encryption on the amazon aws fpga," *IEEE Transactions on Computers*, vol. 69, no. 8, pp. 1185–1196, 2020.
[52] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance fv somewhat homomorphic encryption on gpus: An implementation using cuda," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 70–95, 2018.
[53] 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*, vol. 9, no. 2, pp. 941–956, 2019.
[54] E. R. Türkoğlu, A. Ş. Özcan, C. Ayduman, A. C. Mert, E. Öztürk, and E. Savaş, "An accelerated gpu library for homomorphic encryption operations of bfv scheme," in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2022, pp. 1155–1159.
[55] Ö. Özerk, C. Elgezen, A. C. Mert, E. Öztürk, and E. Savaş, "Efficient number theoretic transform implementation on gpu for homomorphic encryption," *The Journal of Supercomputing*, vol. 78, no. 2, pp. 2840–2872, 2022.
[56] 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 *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 26–39.
[57] 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*, pp. 114–148, 2021.
[58] K. Nam, H. Oh, H. Moon, and Y. Paek, "Accelerating n-bit operations over tfhe on commodity cpu-fpga," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.
[59] T. Zhou, X. Yang, L. Liu, W. Zhang, and N. Li, "Faster bootstrapping with multiple addends," *IEEE Access*, vol. 6, pp. 49 868–49 876, 2018.
[60] M. Joye and P. Paillier, "Blind rotation in fully homomorphic encryption with extended keys," in *Cyber Security, Cryptology, and Machine Learning: 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30–July 1, 2022, Proceedings*. Springer, 2022, pp. 1–18.