2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00033

Trinity: A General Purpose FHE Accelerator

Xianglong Deng[1, 2], Shengyu Fan[1, 2], Zhicheng Hu[3], Zhuoyu Tian[1, 2], Zihao Yang[1, 2], Jiangrui Yu[4, 5],
Dingyuan Cao[6], Dan Meng[1], Rui Hou[1], Meng Li[4, 5], Qian Lou[7] and Mingzhe Zhang[1]
[1]*Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China*
[3]*University of Electronic Science and Technology of China, Chengdu, China*
[4]*Institute for Artificial Intelligence, Peking University, Beijing, China*
[5]*School of Integrated Circuits, Peking University, Beijing, China*
[6]*University of Illinois, Urbana-Champaign, IL, USA*
[7]*University of Central Florida, Orlando, FL, USA*
{dengxianglong, fanshengyu, tianzhuoyu, yangzihao, mengdan, hourui}@iie.ac.cn
zhichenghu@std.uestc.edu.cn, meng.li@pku.edu.cn, jiangrui.yu@stu.pku.edu.cn
dc29@illinois.edu, qian.lou@ucf.edu, mingzhe-zhang@outlook.com

***Abstract*—Fully Homomorphic Encryption (FHE) is crucial for privacy-preserving computing, which allows direct computation on encrypted data. While various FHE schemes have been proposed, none of them efficiently support both arithmetic FHE and logic FHE simultaneously. To address this issue, researchers explore the combination of different FHE schemes within a single application and propose algorithms for the conversion between them. Unfortunately, all prior ASIC-based FHE accelerators are designed to support a single FHE scheme, and none of them supports the acceleration for FHE scheme conversion. This necessitates FHE acceleration systems to integrate multiple accelerators for different schemes, leading to increased system complexity and hindering performance enhancement.**

In this paper, we present the first multi-modal FHE accelerator based on a unified architecture, which efficiently supports CKKS, TFHE, and their conversion scheme within a single accelerator. To achieve this goal, we first analyze the theoretical foundations of the aforementioned schemes and highlight their composition from a finite number of arithmetic kernels. Then, we investigate the challenges for efficiently supporting these kernels within a unified architecture, which include 1) concurrent support for NTT and FFT, 2) maintaining high hardware utilization across various polynomial lengths, and 3) ensuring consistent performance across diverse arithmetic kernels. To tackle these challenges, we propose a novel FHE accelerator named Trinity, which incorporates algorithm optimizations, hardware component reuse, and dynamic workload scheduling to enhance the acceleration of CKKS, TFHE, and their conversion scheme. By adaptive select the proper allocation of components for NTT and MAC, Trinity maintains high utilization across NTTs with various polynomial lengths and imbalanced arithmetic workloads. The experiment results show that, for the pure CKKS and TFHE workloads, the performance of our Trinity outperforms the state-of-the-art accelerator for CKKS (SHARP) and TFHE (Morphling) by 1.49× and 4.23×, respectively. Moreover, Trinity achieves 919.3× performance improvement for the FHE-conversion scheme over the CPU-based implementation. Notably, despite the performance improvement, the hardware overhead of Trinity is only 85% of the summed circuit areas of SHARP and Morphling.

***Index Terms*—CKKS, TFHE, Scheme Conversion, ASIC, FHE.**

I. Introduction

Fully Homomorphic Encryption (FHE) is a novel encryption method that enables direct computations on encrypted data. This capability allows users to perform secure computation on private data in untrusted environments, making FHE a valuable technique for privacy-preserving computing.

Several FHE schemes have been proposed, which can be categorized into two types: the arithmetic FHE (i.e. BGV [8], BFV [7], and CKKS [12]) and logic FHE (i.e. FHEW [15] and TFHE [13]). Arithmetic FHE allows SIMD-style arithmetic operations by packing multiple plaintexts into a ciphertext. In contrast, logic FHE encrypts a single bit into a ciphertext and allows logic operations such as comparison. The real-world applications contain both arithmetic and logic operations, however, none of the FHE schemes can support both of them. Therefore, researchers have started to study the construction of hybrid-scheme applications based on different types of FHE schemes and have proposed several Scheme Conversion algorithms [4], [10], [26] between different types of FHE schemes.

Hybrid-scheme applications pose challenges to the FHE accelerator. Several FHE accelerators [3], [22], [23], [33], [34] have been proposed, but they only provide efficient support for one type of FHE scheme. For example, SHARP [22] can efficiently support the 36-bit version of the CKKS scheme, which outperforms CPU by 22939× with reasonable hardware overheads. Morphling improves the TFHE workloads over CPU by 2343×. Furthermore, to the best of our knowledge, there is no FHE accelerators that can support the Scheme Conversion algorithm. Therefore, a general FHE system should be implemented with heterogeneous accelerators, which increases the system complexity and probably decrease the overall performance.

In this paper, we make a comprehensive investigation for the CKKS, TFHE, and the conversion schemes, and then present several key observations: 1) both schemes and the conversion algorithm consist of a finite set of kernels; 2) the Fast Fourier

Xianglong Deng and Shengyu Fan contributed equally to this work. The corresponding author is Mingzhe Zhang (mingzhe-zhang@outlook.com).

979-8-3503-5057-9/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00033

Transform (FFT) used in TFHE can be approximately replaced by the Number Theoretic Transform (NTT); 3) the key to uniform hardware design is maintaining high utilization across various workloads. Based on the above observations, we propose an accelerator named *Trinity*, which efficiently supports the above schemes within the unified architecture as follows: 1) We substitute FFT with NTT in TFHE by selecting an appropriate modulus q; 2) We design a configurable component that dynamically supports NTT across different polynomial lengths; 3) We implement a component reuse strategy that employs some MAC units for NTT computations, ensuring workload balance across various workloads.

We evaluate our proposed *Trinity* within both CKKS and TFHE applications. The experimental results show that, within CKKS applications, *Trinity* achieves an average speedup over SHARP [22] by 1.49×. For TFHE applications, *Trinity* outperforms Morphling [31] by 4.23×. When considering the conversion between CKKS and TFHE, *Trinity* outperforms CPU-based implementation by 919.3×. When considering the hardware overhead, the area of *Trinity* is smaller than the total area of SHARP and Morphling by 15%.

The contributions of this work are as follows:

- For the first time, we make detailed investigation for the opportunities and challenges of efficiently support various FHE schemes and their conversion algorithms within the unified architecture. These experiences bridge the cryptography and computer architecture, which will probably inspire the future works.
- We explore to provide high-performance support for various FHE kernels (i.e., NTT with different polynomial lengths, and NTT and MAC workload with changing proportion) within the heterogeneous architecture and dynamic schedule.
- We experimentally prove that, it is possible to architect high-performance multi-modal accelerator based on reasonable hardware overhead.

II. Background

In this section, we first describe the data types, kernels, and operations utilized in both the CKKS and TFHE cryptographic schemes. Furthermore, we introduce the algorithm for the Scheme Conversion between CKKS and TFHE. To facilitate comprehension, we summarize all notations, parameters, and operations discussed in this paper in Table I. Additionally, we summarize the hierarchical operation reconstruction model of CKKS in Table II.

A. CKKS scheme

CKKS is an arithmetic FHE, which enables SIMD-style arithmetic operations by packing multiple plaintexts into a ciphertext. The homomorphic operation of CKKS can be decomposed into the following kernels:

- **NTT.** NTT accelerates polynomial multiplication, by transforming a polynomial from the coefficient representation to the evaluation representation.

TABLE I: Notations, parameters and operations.

Notation	Description
m	Plaintext vector.
[[**m**]]	RLWE ciphertext of **m**. [[**m**]]= (a(X), b(X)).
m	Scalar plaintext.
$[[m]]$	LWE ciphertext of scalar message m. $[[m]]$= (**a**, b).
$P(X)$	Polynomial.
$P_{\mathbf{m}}$	Plaintext polynomial corresponding to message **m**.
n	The number of slots in a ciphertext.
N	The polynomial size.
Δ	The scale factor.
R_Q	The polynomial ring, $R_Q = Z_Q/(X^N + 1)$.
Q	The polynomial modulus $Q = \prod_{i=0}^{L} q_i$.
P	The special prime $P = \prod_{i=0}^{\alpha} p_i$.
q_i	The small RNS moduli composing Q.
p_i	The small RNS moduli composing P.
L	The maximum level of polynomial.
l	The current level of polynomial.
R_{q_l}	Polynomial at level l.
dnum	The decomposition number.
α	The number of RNS moduli in a digit. $\alpha = \lceil \frac{L+1}{dnum} \rceil$.
β	$\beta = \lceil \frac{l+1}{dnum} \rceil$.
evk	The evaluation key.
$\mathrm{n}_{\mathrm{lwe}}$	The dimension of LWE ciphertext.
k	The dimension of GLWE ciphertext.
q	The modulus of coefficients.
l_b	The decomposition level of bootstrapping key.
l_k	The decomposition level of TFHE keyswitching key.
ACC_i	The accumulation ciphertext in i-th iteration.
ksk	The keyswitch key in TFHE.
bsk	The bootstrapping key in TFHE.
$\mathrm{n}_{\mathrm{slot}}$	The number of valid slots in the RLWE ciphertext.
Rotate	Rotate([[**m**]]) $\rightarrow (a(X) \cdot X^r, b(X) \cdot X^r)$.
SampleExtract	SampleExtract([[**m**]], i) $\rightarrow$ [[**m**[i]]]
Decompose	Decompose($P(X)$, l) $\rightarrow$ ($\tilde{P}(X)_0$,...,$\tilde{P}(X)_{l-1}$).

TABLE II: Hierarchical reconstruction model of CKKS.

Operation	Description	Composing Kernels
HMult	Multiply two ciphertexts	NTT, BConv, IP, ModMul, ModAdd
PMult	Multiply a ciphertext by a plaintext	ModMul, ModAdd
HRotate	Homomorphic rotation of ciphertext	NTT, BConv, IP, ModMul, ModAdd, Auto
HAdd	Add two ciphertexts	ModAdd
PAdd	Add a ciphertext and a plaintext	ModAdd
Rescale	Reduce the level of a ciphertext	NTT, ModAdd

- **BConv.** BConv adjusts the coefficient modulus of a polynomial. The process involves multiplying a polynomial matrix of size $\alpha \times N$ by a base matrix of size $l \times \alpha$, effectively changing the modulus from q_α to q_l [23].
- **IP.** IP multiplies a polynomial with the evaluation key (evk) during KeySwitch. IP operations can also be structured as a multiplication between a vector $[\tilde{d_0}, \ldots, \tilde{d}_{dnum-1}]$ and an evk matrix of size $dnum \times 2$
- **ModMul.** ModMul performs the element-wise modular multiplication of two polynomials that are in evaluation representation.
- **ModAdd.** ModAdd performs the element-wise modular addition of two polynomials.
- **Auto.** Auto performs automorphism transform on a polynomial, which maps the indices of each coefficient

Algorithm 1: `Hybrid KeySwitch` ($[d]_{C_l}$, evk)

```
Require: $[d]_{C_l}$: a polynomial with level $l$, evk: the evaluation key
$[d']_{C'_i}$ = Decompose($[d]_{C_l}$, $\beta$)
$\tilde{ct}_1$ = 0; $\tilde{ct}_2$ = 0
for i = 0; i < $\beta$; i = i + 1 do
    $[\tilde{d}]_{D_\beta}$ = BConv$_{C'_i \to D_\beta}$($d'$[i])
    $[\tilde{d}]_{D_\beta}$ = NTT($[\tilde{d}]_{D_\beta}$)
end for
for j = 0; j < 2; j = j + 1 do
    for i = 0; i < $\beta$; i = i + 1 do
        $[\tilde{ct}_j]_{D_\beta}$ = $[\tilde{ct}_j]_{D_\beta}$ + $[\tilde{d}]_{D_\beta}$ * evk$_{i,j}$
    end for
    $[\tilde{ct}_j]_{D_\beta}$ = iNTT($[\tilde{ct}_j]_{D_\beta}$)
    $ct_{mult_j}$ = $[\tilde{ct}_j]_{C_l}$ - BConv$_{B \to C_l}$($[\tilde{ct}_j]_B$)
end for
return $ct_{mult_1}$, $ct_{mult_2}$
```

Algorithm 2: `TFHE PBS` (c, tv, bsk, ksk)

```
Require: $c$: a LWE ciphertext, $c$ = ($\mathbf{a}$, $b$); $tv$: the test vector; $bsk$: the bootstrapping key; $ksk$: the key switching key
$\tilde{c}$ = ($\tilde{\mathbf{a}}$, $\tilde{b}$) = ModSwitch($c$) // ModSwitch
ACC$_0$ = Rotate($tv$, $b$)
// Blind Rotation
for i = 1 to $n_{lwe}$ do
    tmp = (Rotate(ACC$_{i-1}$, $\tilde{a}_i$) - ACC$_{i-1}$)
    tmp = Decompose(tmp, $l_b$)
    // External Product
    for j = 1 to $(k+1)l_b$ do
        tmp_acc = tmp_acc + NTT(tmp[j]) * $bsk$[i][j]
    end for
    ACC$_i$ = ACC$_{i-1}$ + INTT(tmp_acc)
end for
// SampleExtract
$c'$ = ($\mathbf{a}'$, $b$) = ($a'_1$,...,$a'_{kN}$, $b'$)
                = SampleExtract(ACC$_n$, 0)
// TFHE KeySwitch
$\mathbf{a}''$ = Decompose($\mathbf{a}'$, $l_k$)
$c''$ = (0,...,0,$b'$) - $\sum_{i=0}^{kN} \sum_{j=0}^{l_k}$ $\mathbf{a}''_i$[j] * $ksk$[i][j]
return $c''$
```

from i to $\sigma_r(i)$ ($\sigma_r(i) = i \cdot 5^r \mod N$).

Based on the above kernels, CKKS provides several homomorphic operations, which are shown in Table II.

Against the significant performance and implementation overhead of the CKKS scheme, researchers have proposed a number of optimizations, which are introduced as follows:

- **Residue Number System (RNS).** In CKKS schemes, the coefficient modulus is extremely large, which can be over 1000 bits. Therefore, the CKKS scheme based on RNS has been proposed [11]. This CKKS scheme decomposes a polynomial into multiple smaller ciphertexts with smaller coefficient moduli.
- **Hybrid KeySwitch.** KeySwitch is used in HMult and HRotate to ensure that the ciphertext remains decryptable using the original secret key. KeySwitch is a time-consuming operation in CKKS. To further reduce the performance overhead of KeySwitch, Hybrid KeySwitch [18] is proposed, which reduces the temporary moduli in KeySwitch. A detailed procedure of Hybrid KeySwitch is shown in Algorithm 1.

B. TFHE scheme

TFHE is a logic FHE, which encrypts one message into a ciphertext and can enable logic operations including comparison. Here, we briefly introduce the ciphertexts, kernels, and operations in TFHE.

GLWE and GGSW ciphertexts. GLWE ciphertexts are a generalized version of RLWE for encrypting polynomial-type messages. A GLWE ciphertext is structured as $(A_1(x), \ldots, A_k(x), B(x))$ within R_q^{k+1}. The vector $(A_1(x), \ldots, A_k(x))$ is known as the GLWE mask. GGSW ciphertexts extend GLWE and are described as a matrix of polynomials of size $(k+1) \times ((k+1)l_b)$, where l_b represents the decomposition level of the GGSW ciphertext.

External Product. The External Product, a crucial operation in programmable bootstrap (PBS), involves multiplying a GLWE ciphertext by a GGSW ciphertext. Initially, the GLWE ciphertext is decomposed into l_b smaller parts. These parts, forming a vector of size $(k+1)l_b$, are multiplied by a bootstrapping key matrix bsk, a GGSW ciphertext of dimensions $(k+1)l_b \times (k+1)$.

PBS. PBS, central to TFHE, facilitates arbitrary function evaluation during bootstrapping as detailed in Algorithm 2. PBS includes a series of operations: ModSwitch, Blind Rotation, SampleExtract, and TFHE KeySwitch. ModSwitch adjusts the scale of a ciphertext, computing $\tilde{c} = (\lfloor \frac{2N}{q} \cdot \mathbf{a} \rceil, \lfloor \frac{2N}{q} \cdot b \rceil)$. Blind Rotation transforms an LWE ciphertext into a noise-free GLWE ciphertext through n_{lwe} iterations of External Products. TFHE KeySwitch performs the transformation $c'' = (0, ..., 0, b') - \sum_{i=0}^{kN} \sum_{j=0}^{l_k} \mathbf{a}'_j \cdot ksk[i][j]$, where ksk is comprised of $kN \times l_k$ LWE ciphertexts in $Z_q^{n_{lwe}+1}$, with l_k representing the decomposition level of ksk.

Substituting FFT with NTT. While FFT is commonly used in TFHE to accelerate External Product, it is possible to substitute FFT with NTT by selecting a prime modulus p, which satisfies $p \equiv 1 \mod 2N$ and is chosen to be the closest prime to q [20], [37]. This facilitates the reuse of NTT hardware in TFHE.

C. The Scheme Conversion Between CKKS and TFHE

Scheme Conversion is used to convert between arithmetic FHE and logic FHE. Taking the conversion between CKKS and TFHE as an example, we briefly introduce the Scheme Conversion algorithm.

LWE and RLWE ciphertext. LWE ciphertext is used as the basic ciphertext in TFHE, while RLWE is used in

Algorithm 3: Scheme Conversion from CKKS to TFHE (ct)

```
Require: $ct$: a RLWE ciphertext $(a, b)$
for i = 0 to $n_{\text{slot}}$ - 1 do
    $ct'_{i+1}$ = SampleExtract($ct$, i)   ▷ Sample Extraction
end for
$lwect \leftarrow \{ct'_1,...,ct'_{n_{slot}}\}$
return $lwect$
```

Algorithm 4: PackLWEs(**ct**)

```
Require: ct: a vector of RLWE ciphertexts, ct = $\{ct_1,...,ct_{n_{slot}}\}$
if $n_{slot}$ = 1 then
    $ct$ = $ct_1$
else
    $ct_{even}$ = PackLWEs($\{ct_{2j}\}_{j\in[1,n_{slot}/2]}$)
    $ct_{odd}$ = PackLWEs($\{ct_{2j-1}\}_{j\in[1,n_{slot}/2]}$)
    $ct$ = ($ct_{even}$ + Rotate($ct_{even}$, $N/n_{slot}$)) +
        HRotate($ct_{even}$ - Rotate($ct_{odd}$, $N/n_{slot}$), $n_{slot}$ + 1)
end if
return $ct$
```

CKKS. LWE ciphertext encrypts a scalar message, while RLWE ciphertext packs multiple plaintexts.

Scheme Conversion from CKKS to TFHE [10]. Scheme Conversion from CKKS to TFHE converts an RLWE ciphertext into multiple LWE ciphertexts, as shown in Algorithm 3. The procedure includes n_{slot} multiple SampleExtract operations, where each operation extracts a specific coefficient from the message polynomial encrypted in the RLWE ciphertext.

Scheme Conversion from TFHE to CKKS [10]. Scheme Conversion from TFHE to CKKS packs multiple LWE ciphertexts into a single RLWE ciphertext, as shown in Algorithm 4 and 5. The transformation involves three key steps: Ring Embedding, Ciphertext Packing, and Field Trace. Ring Embedding first transforms an LWE ciphertext into an RLWE format. Ciphertext Packing then merges multiple RLWE ciphertexts using Rotate and HRotate techniques. Finally, Field Trace modifies the combined ciphertext by eliminating unused coefficients of the plaintext polynomial, facilitating further homomorphic operations. Scheme Conversion TFHE to CKKS includes operations such as Rotate and HRotate. Introduced respectively in the TFHE and CKKS schemes, these operations enable the integration and reuse of methodologies from both schemes.

Note that, although there are already several FHE conversion schemes [4], [10], [26], we take the scheme from Ref [10] as the example to fulfill scheme conversion in this paper, since this algorithm achieves a higher precision than other schemes. Nonetheless, since the Scheme Conversion algorithms (such as OpenFHE [4] and Pegasus [26]) can also be decomposed into CKKS and TFHE operations, Trinity can still support these algorithms.

Algorithm 5: Scheme Conversion from TFHE to CKKS ($lwect$)

```
Require: $lwect$: $n_{\text{slot}}$ LWE ciphertexts $(\mathbf{a}_j, b_j)$
Set $ct_j = (a_j, b_j) \in R_q^2$ for each $j \in [n_{slot}]$, where
    $a_j = \sum_{i\in[N]} \mathbf{a}_j[i] \cdot X^i$.   ▷ Ring Embedding
$ct$ = PackLWEs($ct_1$,...,$ct_{n_{slot}}$)   ▷ Ciphertext Packing
for k = 1 to $\log(N/n_{slot})$ do
    $ct$ = $ct$ + HRotate($ct$, $2^{\log N-k+1}$)   ▷ Field Trace
end for
return $ct$
```

III. Motivation

This section first introduces the requirements for various FHE schemes. Subsequently, we introduce the challenges for using a single type of hardware to support the CKKS, TFHE and the Scheme Conversion between them. Then, we introduce opportunities for optimization.

A. The necessities for supporting multi FHE schemes

Different applications own various characteristics, such as data-intensive, logic-intensive, and a mix of both. Therefore, different kinds of computation are required to meet the various requirements. For example, linear computation is considered as the suitable choice for data-intensive applications due to its SIMD support, while it is less efficient for logic-intensive applications. When considering the FHE schemes, the schemes with linear operation support (i.e., BGV, BFV, CKKS) are proper choice for data-intensive applications, while the FHE scheme with logical operation support (i.e., TFHE) is suitable for logic-intensive applications. Moreover, for the FHE applications with mixed requirements (i.e., FHE database), both linear FHE scheme and logical FHE scheme are simultaneously required. Therefore, for a real-world system, it is important to meet the requirements of different kinds of applications and thereby require efficient support for multi FHE schemes.

B. ***Challenge 1:*** *Various polynomial lengths in CKKS and TFHE*

In TFHE, due to security considerations and a relatively small coefficient modulus, a polynomial length ranging from 256 to 4096 is sufficient. In contrast, in CKKS, particularly for configurations that include Bootstrapping, N is usually set to 2^{16}. We investigate the utilization of F1-like NTT and FAB-like NTT under various polynomial lengths in Figure 1. The result demonstrates that a fixed hardware design cannot achieve high utilization across various polynomial lengths: 1) F1-like NTT achieves its highest utilization when N = 2^{16}, while the utilization starts to decrease when N decreases; 2) FAB-like achieve its highest utilization when N = 2^8, while the utilization starts to decrease when N increases. Therefore, it would be hard for specific hardware to efficiently support the NTT across various polynomial lengths, when supporting CKKS and TFHE schemes. This will incur low utilization and thereby performance degradation.

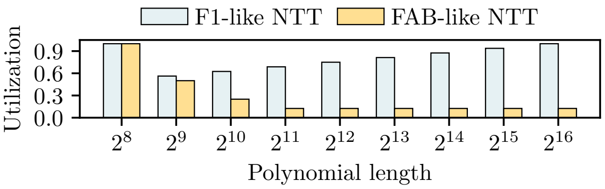


Fig. 1: Utilization of F1-like NTT and FAB-like NTT when computing NTT of varying lengths. For a fair analysis, both the F1-like NTT and the FAB-like NTT are configured with comparable modular multipliers. The F1-like NTT includes eight stages of butterfly units and processes 256 elements in parallel per cycle. In contrast, the FAB-like NTT consists of a single butterfly stage capable of processing 2048 elements in parallel per cycle. Both NTT employ radix-2 NTT and support four-step NTT. The utilization rate is computed considering a single butterfly stage as the finest granularity.

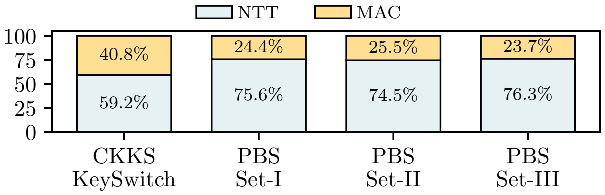


Fig. 2: The computational amount breakdown of NTT and MAC operation in CKKS KeySwitch (L = 23, dnum = 3) and TFHE PBS.

C. *Challenge 2: Imbalanced computation workload*

As introduced in Section II-A, KeySwitch is a pivotal operation in the CKKS scheme, employed in both HRotate and HMult, whereas PBS is a fundamental operation for arbitrary function evaluation in TFHE. KeySwitch consists primarily of NTT, BConv, and Inner Product computations, where NTT and BConv can be computed using MAC. PBS predominantly involves external products that include NTT and MAC computations. In the FHE accelerator, the implementation of the NTT unit commonly utilizes multiple butterfly stages, while the computation of MAC utilizes a systolic array. Due to this difference, there are dedicated computational components for NTT and MAC, respectively. However, as shown in Figure 2, due to the computation of FHE scheme, there is workload imbalance: 1) when running CKKS KeySwitch, the computational load of NTT accounts for 59.2% of the total in CKKS KeySwitch, while MAC constitutes 40.8%; 2) For PBS, NTT represents on average 75.5% of the total computational load for PBS, with MAC comprising the remaining 24.5%. In this situation, the computation of NTT becomes the computational bottleneck, where the NTT unit is running with full workloads while the systolic array remains idle. The above analysis suggests that it is difficult to use the same hardware configuration to efficiently support CKKS and TFHE.

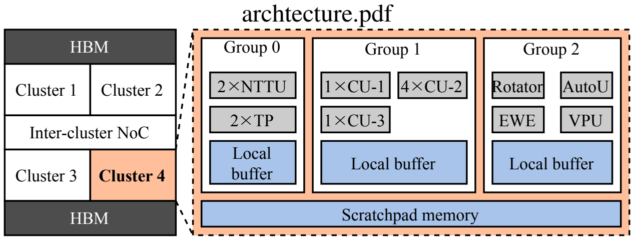


Fig. 3: Overall Architecture of *Trinity*. NTTU denotes the NTT unit. TP denotes transpose unit. CU-x denotes a configurable unit with x-column PEs.

D. *Opportunities*

For the analysis and challenges presented, we conducted a thorough investigation and arrived at the following observations and opportunities:

- We observe that it is possible to substitute FFT with NTT in TFHE by selecting a proper value for q [20], [37]. Given that prior CKKS accelerators [22], [23], [34] implement NTT units, this provides the opportunity to reuse the NTT unit of CKKS accelerators in the computation of TFHE.
- By comparing F1-like NTT and FAB-like NTT, we observe that the optimal polynomial lengths supported by NTT units vary according to the number of butterfly stages. This inspires us to deploy heterogeneous components to support NTT with various polynomial lengths.
- As discussed in Section II-A, both BConv and IP can be executed using a systolic array. Furthermore, the computation pattern for NTT also aligns similarly with that of a systolic array [28]. This alignment provides an opportunity to design a configurable component capable of supporting both NTT and MAC computations. By using parts of this component for NTT computations, we can ensure a balanced computational workload across both CKKS and TFHE schemes, thereby enhancing performance.

In this work, we propose a novel and configurable accelerator named Trinity, which is designed to provide efficient and flexible support for the computation of the CKKS, TFHE scheme, and the conversion between them.

IV. Design

A. *Overview*

Figure 3 presents the overall architecture of *Trinity*, which mainly contains the following components:

- **Cluster**. The cluster functions as a high-performance and configurable computing engine, efficiently supporting the operations of the CKKS and TFHE schemes and facilitating conversions between them. As illustrated in the right section of Figure 3, each cluster has a scratchpad memory and three heterogeneous groups: Group 0, Group 1, and Group 2. Group 0 comprises two transpose units (TP) and two NTT units (denoted as NTTU). Group 1

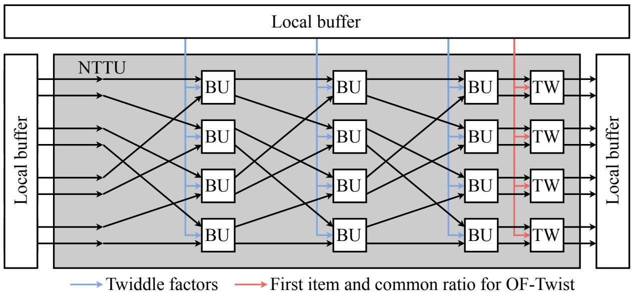


Fig. 4: The structure of NTTU. We denote the number of rows of the BU array as M. In the default configuration of *Trinity*, we set M as 128, and NTTU processes 256 elements each cycle. For simplicity, here we take M = 4 as an example.

includes a set of configurable units. These configurable units contain multiple columns of processing elements (PE). Here, we denote the configurable unit as CU-x, which means a configurable unit with x-column PEs. Besides, we use CU to refer to all kinds of CU-x. Group 1 includes six CUs, designated one CU-1, four CU-2, and one CU-3. CU is pivotal in *Trinity*, supporting both NTT and MAC computations (detailed in Section IV-C). Group 2 contains a Rotator, an automorphism unit (AutoU), an element-wise engine (EWE), and a vector processing unit (VPU).

- **Inter-cluster NoC**. This network-on-chip (NoC) enables all-to-all data exchange among different clusters, playing a crucial role in switching between two data layouts (detailed in Section IV-I). This NoC employs a fully-connected topology.
- **On-chip memory**. There are two types of on-chip memory, namely scratchpad memory and local buffer. Scratchpad memory is shared across the groups in a cluster, while the local buffer is shared across the functional units in a group. Each group has a local buffer, while each has a scratchpad memory.
- **HBM**. *Trinity* facilitates two HBM2 interfaces for off-chip data exchange, providing a total bandwidth of 1TB/s.

B. NTTU structure

As shown in Figure 4, The NTTU supports NTT computations, which contains multiple butterfly units (BUs) with $\log_2(2M)$ stages and one-stage twisting units (TWs). BU computes the butterfly operation, while TW computes the twisting operation in four-step NTT. In the design of the BU array in NTTU, We employ constant-geometry [9] NTT [30], which can maintain a consistent access pattern for the computation of BUs in each stage. To minimize memory bandwidth requirements, we implement on-the-fly twisting factor generation (OF-Twist), similar to the approach in ARK [23]. NTTU enables the computation of $2M$-point NTT and works in a fully pipelined manner, processing $2M$ elements per cycle.

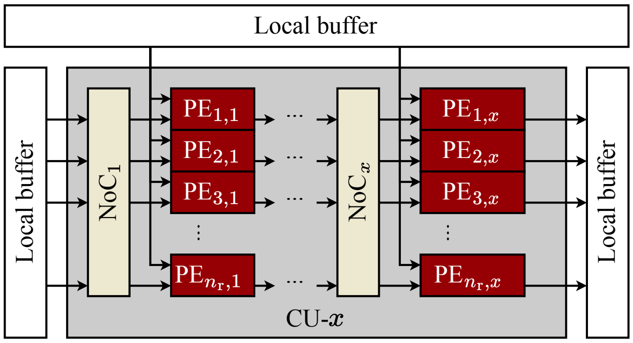


Fig. 5: CU-x architecture

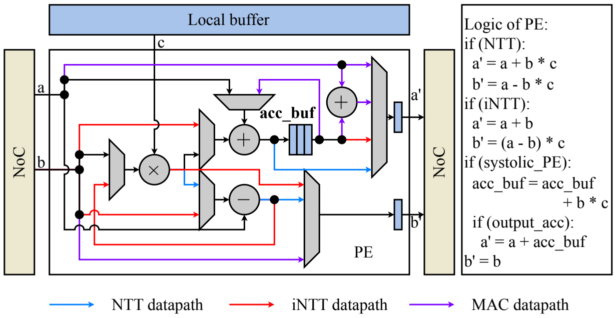


Fig. 6: PE structure

For the transmission of auxiliary data, the twiddle factors are fed in BUs from the local buffer as shown in Figure 4. For twisting factors, only the first item and common ratios are fed in TW for the on-the-fly twisting factor generation.

In *Trinity*, we set M as 128, which means an NTTU can facilitate the computation of 256-point NTT. By collaborating with CU, NTTU can compute the NTT of polynomial lengths ranging from 256 to 65536.

C. Configurable Unit

1) Architecture of Configurable Unit: To enhance computational efficiency across varying polynomial sizes and TFHE parameter sets, we propose a configurable unit named CU. Figure 5 presents the structure of CU-x, which consists of x columns and n_r rows of PE. These PE are interconnected via a Network-on-Chip (NoC) that facilitates the specific access patterns required for NTT computations and systolic arrays. The NoC can be configured as a 2D-mesh topology for systolic array and butterfly topology for NTT. Similar to NTTU, our implementation adopts the butterfly topology based on the constant-geometry algorithm, maintaining consistent access patterns across all butterfly stages. This approach simplifies the NoC design and enhances the configurability of the CU. By employing this approach, the area overhead of the NoC constitutes only 0.2% of the total area of CU-x.

2) PE structure: While the NoC in CU-x provides different access patterns for NTT and systolic array, the PEs in CU-x provide different compute patterns for NTT and MAC

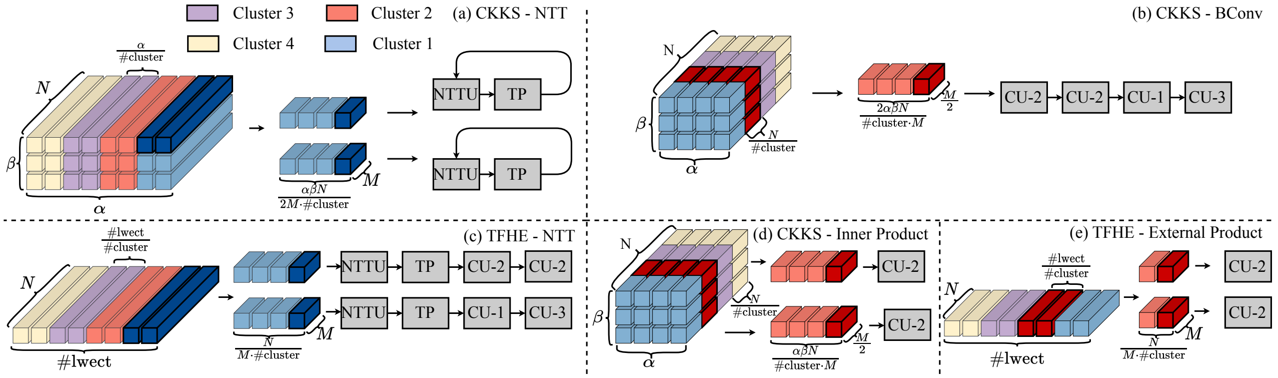

Cluster 3
Cluster 2
Cluster 4
Cluster 1
(a) CKKS - NTT
$\frac{\alpha}{\#\text{cluster}}$
N
β
α
$\frac{\alpha\beta N}{2M\cdot\#\text{cluster}}$
M
NTTU
TP
NTTU
TP
(b) CKKS - BConv
N
β
α
$\frac{N}{\#\text{cluster}}$
$\frac{2\alpha\beta N}{\#\text{cluster}\cdot M}$
$\frac{M}{2}$
CU-2
CU-2
CU-1
CU-3
(c) TFHE - NTT
$\frac{\#\text{lwect}}{\#\text{cluster}}$
N
#lwect
$\frac{N}{M\cdot\#\text{cluster}}$
M
NTTU
TP
CU-2
CU-2
NTTU
TP
CU-1
CU-3
(d) CKKS - Inner Product
N
β
α
$\frac{N}{\#\text{cluster}}$
CU-2
$\frac{\alpha\beta N}{\#\text{cluster}\cdot M}$
$\frac{M}{2}$
CU-2
(e) TFHE - External Product
$\frac{\#\text{lwect}}{\#\text{cluster}}$
N
#lwect
CU-2
$\frac{N}{M\cdot\#\text{cluster}}$
M
CU-2

Fig. 7: Mapping strategy and data layout for kernels in different schemes, including a) NTT in CKKS; b) BConv in CKKS; c) NTT in TFHE; d) Inner Product in CKKS; e) External Product in TFHE.

operations. Figure 6 presents the structure and datapath of PE. This PE supports multiple computing patterns. As shown in Figure 6, the arrow with different colors denotes a kind of datapath for a specific computation, while the arrow with black color denotes the shared datapath for more than two computations. The PE can be configured to compute NTT, iNTT, and MAC.

Each CU works in a fully-pipelined manner. With the capability of PE, CU supports both NTT and MAC computations. Note that the throughput is different when CU is computing NTT and when CU is computing MAC. When computing NTT, each CU processes $2n_r$ elements per cycle. When computing MAC, each CU processes n_r elements per cycle. In *Trinity*, n_r is set to 128, and each CU processes 256 elements per cycle when computing NTT and processes 256 elements per cycle when computing MAC.

D. Transpose unit, element-wise engine, automorphism unit, Rotator, vector processing unit

Transpose unit (TP). The TP manages the transposition of polynomials ranging in size from 512 to 65536. TP follows the same design of [33], containing multiple stages of quad-swap units. Unused stages are bypassed for smaller polynomials.

Element-wise engine (EWE). The design of EWE follows [22], which can handle modular operations such as ModAdd and ModMult.

Automorphism unit (AutoU). The design of AutoU follows [23], which includes multiple stages of shuffle units for automorphism computations.

Rotator. Rotator enables vector rotation and SampleExtract operations, featuring buffers, a vector rotate unit, and a vector negation unit.

Vector processing unit (VPU). VPU handles Modulus Switch and KeySwitch operations, following the design of Morphling [31].

Among all the above components, each EWE can process 512 elements in parallel per cycle, whereas other components handle 256 elements per cycle.

E. Efficient support for NTT with various polynomial lengths

When computing $2M$-point NTT, the TWs are bypassed. When computing the NTT with a larger polynomial length than $2M$, we employ the four-step NTT method [5], which divides NTT into two smaller NTTs, namely phase-1 NTT and phase-2 NTT. Besides, we develop two computing strategies for different cases. When computing $4M^2$-point NTT, NTTU computes both the phase-1 NTT and phase-2 NTT. When computing NTT with the polynomial length ranging from $4M$ to $2M^2$, the phase-1 NTT is computed by NTTU, while the phase-2 NTT is computed by CU (detailed in Section IV-C).

F. Efficient support for imbalanced FHE workloads

In this section, we introduce the data mapping of kernels for different schemes on *Trinity*, focusing on NTT, BConv, Inner Product, and External Product—key operations in both CKKS and TFHE. Figure 7 presents the data mapping for kernels in CKKS (N = 65536) and TFHE (N = 4096). Our strategy prioritizes fulfilling NTT requirements first. Subsequently, unutilized CUs are allocated for the computations of BConv, Inner Product, and External Product. Specifically, in the CKKS scheme, two NTTUs and two TPs are dedicated to NTT computation, as shown in Figure 7(a). For BConv computations, one CU-1, one CU-3, and one CU-4 are utilized, as shown in Figure 7(c). Notably, we modify the computation of the Inner Product by deploying two CU-2s instead of using the EWE, alleviating the computational load on the EWE and enhancing CKKS performance.

For TFHE, as shown in Figure 7(c), NTTU, CU-1, CU-3, and two CU-2 are deployed for NTT. This deployment allows for parallel processing of two NTTs, achieving high throughput and high utilization. Besides, the External Product is computed using two CU-2s, as shown in Figure 7(d).

Through this approach, *Trinity* leverages the configurability of CU-x to dynamically meet the diverse requirements of NTT and MAC operations across varying polynomial sizes and different schemes. This method significantly enhances performance across various scenarios while maintaining a modest area.

G. *Support for scheme conversion*

As introduced in Section II-C, the computation of scheme conversion reuses operations from CKKS and TFHE. The scheme conversion from CKKS to TFHE purely contains SampleExtract, which is performed by Rotator. The scheme conversion from TFHE to CKKS mainly includes HRotate, and Rotate. Rotate is performed by Rotator. HRotate is performed by AutoU, NTTU, CU, and EWE.

H. *Hardware control*

For the components except CU, these components are all separate and only support a single kernel from CKKS or TFHE. For CUs, despite being utilized for the computation of different schemes, Trinity does not simultaneously execute the operations from different schemes. Therefore, CUs do not cause contention and do not induce additional control overheads.

I. *Data layout*

In terms of data layout, *Trinity* employs two data layout patterns: limb-wise and slot-wise, aligning with the configurations in ARK [23]. As depicted in Figure 7, limb-wise layout is adopted for NTT computations in CKKS and all operations in TFHE, whereas slot-wise layout is used for BConv and Inner Product computations in CKKS. Switching between these layouts is managed via the inter-cluster NoC.

J. *On-chip memory system and on-chip network*

The on-chip memory system of *Trinity* comprises multi-level memory, which includes scratchpads and local buffers. The scratchpad facilitates data exchange with the HBM, other clusters, and within the cluster groups, coordinating data transfers to other clusters via the inter-cluster NoC and to different groups through the local buffers. Each local buffer, equipped with 256 lanes, features vectorized memory with each lane comprising five single-ported banks of 36-bit width. Each bank can store two polynomials of length 65536, providing a total capacity of 2.81 MB and a total bandwidth of 11.25 TB/s per local buffer. Similarly, the scratchpad comprises 256 lanes containing four single-ported banks, each 36-bit wide. This configuration allows each bank to store 40960 items, adequately supporting storage requirements for polynomials, evk, bsk, and ksk, with a total capacity of 45 MB and a bandwidth of 9 TB/s.

The on-chip network in *Trinity* can be categorized into three types. The first type is the inter-cluster NoC, which facilitates all-to-all data exchange among different clusters and switching between limb-wise and slot-wise data layouts. The second type is the inter-group NoC, which facilitates data exchange among the groups in a cluster. The third type is the intra-group NoC, which facilitates data exchange between different computational units within a group. The fourth type is the NoC inside the CU, which facilitates data exchange between consecutive butterfly stages.

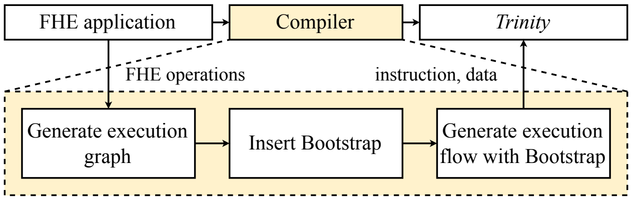


Fig. 8: Workload allocation procedure in *Trinity*.

TABLE III: The configuration of *Trinity*.

Parameter	**Value**
word size	36bit
# of cluster	4
Capacity of scratchpad memory	180MB
n_r of CU	128
M of NTTU	128
$\#_{\text{butterfly stage}}$ of NTTU	8
# of HBM2 stacks	2

K. *Workload Allocation*

Figure 8 illustrates the procedure of workload allocation for *Trinity*. For one FHE workload based on CKKS, TFHE, or Hybrid Schemes, it will be firstly decomposed as the kernel flow. Then, the kernel flow is carefully scheduled to eliminate the hardware hazards and guarantee hardware utilization. After that, the accelerator executes the scheduled kernel flow without distinguishing which FHE scheme the kernel comes from. In this way, Trinity enables the support for the FHE applications based on both single schemes and hybrid schemes, and even supports for simultaneous execution of multiple FHE applications, without hardware switching overhead.

V. Methodology

A. *Hardware modeling*

For the evaluation of power, area, and frequency, all the hardware modules in *Trinity* are implemented using Verilog and synthesized with the TSMC 7nm Process Design Kit (PDK). All the modules are fully pipelined, enabling operation at a frequency of 1GHz. For the estimation of SRAM components, we utilize the SRAM components provided by the PDK, all of which are double-pumped. For the estimation of NoC, we utilize ORION [21], a tool for estimating the area and power of NoC. Furthermore, we develop a cycle-accurate simulator to evaluate the performance under full FHE workloads. Table III presents the hardware configurations of *Trinity* that are used in the experiment.

B. *Benchmark suite*

In this section, we introduce the workloads that are used to compare with other ASIC accelerators. We have categorized the benchmarks we used into three categories: CKKS Benchmark, TFHE Benchmark, and Scheme Conversion Benchmark.

TABLE IV: CKKS and TFHE parameter sets.

	CKKS				
	N	L	dnum		λ
default	65536	35	3		128-bit
	TFHE				
	N	$\mathrm{n}_{\mathrm{lwe}}$	k	l_b	λ
Set-I	1024	500	1	2	80-bit
Set-II	1024	630	1	3	110-bit
Set-III	2048	592	1	3	128-bit

TABLE V: Comparison Scheme

Type	**Design**	**Description**
	CKKS	
CPU	Baseline-CKKS [34]	AMD Ryzen 3975WX
GPU	TensorFHE [16]	NTT with TCUs
ASIC	CraterLake [34]	1×CRB, 2×NTT, 1×Auto, 5×Mul, 5×Add
ASIC	BTS [24]	2048×PE, each with 1 ModMult, 1 MMAU, 1NTTU
ASIC	ARK [23]	4 clusters each with 1 NTTU, 1 BConvU, 1 AutoU, 2 MADU
ASIC	SHARP [22]	4 clusters, each with 1 NTTU, 1 BConvU, 1 AutoU, 1 EWE
ASIC	*Trinity*	4 clusters, each with 2 NTTU, 1 CU-1, 4 CU-2, 1 CU-3, 1 AutoU, 1 Rotator, 1 VPU, 1 EWE
	TFHE	
CPU	Baseline-TFHE [31]	Intel Xeon Platinum
GPU	NuFHE [29]	Nvidia Titan RTX
ASIC	Matcha [19]	32×IFFT, 8×FFT, 160×Mult, 192×Add
ASIC	Strix [32]	8×HSC, each with 2 VMA, 1 IFFT, 1 FFT, 2 Decomp, 2 Accum, 1 Rotator
ASIC	Morphling [31]	8×FFT, 16× IFFT, 64×VPE,1×VPU
ASIC	*Trinity*	-
	Hybrid scheme	
CPU	Baseline-SC [10]	I7-4770K
CPU	Baseline-Hybrid [6]	Intel Xeon Platinum
System	SHARP+Morphling	A system including SHARP and Morphling, assuming a 128GB/s PCIE 5 connection between them
ASIC	*Trinity*	-

1) CKKS Benchmark: We utilize the following CKKS applications to evaluate the CKKS effectiveness of *Trinity*:

- **Packed Bootstrapping [27]**: This example involves the operation of a fully packed bootstrapping. The level consumption of bootstrapping is 15.
- **Logistics Regression [17]**: This benchmark involves the training of a binary classification model using logistic regression. The batch size is set to 1024 and the number of training iterations is 32.
- **ResNet-20 [25]**: This benchmark involves a CNN inference with CIFAR-10 dataset using the CKKS-based ResNet-20 model. The size of the input image is 32 × 32 × 3.

TABLE VI: Performance for CKKS workloads (ms).

Scheme	**Bootstrap**	**HELR**	**ResNet-20**
Baseline-CKKS	17.2s	356s	23min
TensorFHE [16]	421.8	220	4,939
F1 [33]	-	639	2,693
CraterLake [34]	3.91	119.52	249.45
BTS [24]	22.88	28.4	1,910
ARK [23]	3.52	7.42	125
SHARP [22]	3.12	2.53	99
Trinity	1.92	1.37	89

Note that all the CKKS benchmarks are evaluated using the default CKKS parameter sets in Table IV.

2) TFHE Benchmark: For the TFHE effectiveness evaluation, the following applications are utilized as the benchmark:

- **Programmable Bootstrapping (PBS) [14]**: This benchmark involves programmable bootstrapping. We evaluate the performance of PBS under Set-I, Set-II, and Set-III, respectively (see in Table IV).
- **NN-x [14]**: This benchmark involves a CNN inference on the MNIST dataset. The size of input images is 28 × 28. The x denotes the depth of the neural networks. We tested the latency of NN-20, NN-50, and NN-100. The NN on baseline-TFHE is evaluated using 12 threads of the Intel Xeon Platinum 8280 CPU.

3) Scheme Conversion Benchmark: We use the conversion scheme introduced in Ref [10] to evaluate *Trinity*. This scheme involves the repacking procedure that converts a set of LWE ciphertexts into an RLWE ciphertext. The parameter n_{slot} denotes the number of LWE ciphertext. Note that, we do not test the conversion from CKKS to TFHE operation, since this operation is simple, and only contains multiple *SampleExtract* operations. For ease of comparison, we set $N = 2^{14}$ and $L = 8$ in this benchmark, which is consistent with Ref [10].

4) Hybrid scheme Benchmark: **HE3DB-x [6]**: This benchmark homomorphically performs Query 6 in the TPC-H benchmark [1], which is a kind of typical queries. This benchmark involves filter and aggregation, which are respectively executed in TFHE and CKKS domains. Scheme conversion is performed between filter and aggregation. We denote this benchmark as HE3DB-x, where x denotes the number of entries to be queried. We evaluate the latency of HE3DB-4096 and HE3DB-16384. The HE3DB on baseline-Hybrid is evaluated using a single thread of Intel Xeon Platinum 8280. To compare with SOTA accelerators, we assume a system named SHARP+Morphling, where its configuration is listed in Table V. In SHARP+Morphling, we assume the TFHE-to-CKKS conversion is performed on SHARP and the CKKS-to-TFHE conversion is performed on Morphling.

Together with the above benchmark, we can make a thorough analysis of the performance and efficiency of *Trinity* when executing full-FHE applications based on CKKS, TFHE and Scheme Conversion between them. The designs used for comparison are shown in Table V.

TABLE VII: Throughput for TFHE PBS (OPS).

Scheme	Set-I	Set-II	Set-III
Baseline-TFHE [38]	63	36	12
GPU	2,500	550	-
Matcha [19]	10,000	-	-
Strix [32]	74,696	39,600	21,104
Morphling [31]	147,615	78,692	41,850
Morphling$_{\text{1GHz}}$	123,012	65,576	34,875
Trinity-TFHE$_{\text{w/o CU}}$	83,333	49,603	26,393
Trinity-TFHE$_{\text{w/ CU}}$	150,015	85,034	45,246
Trinity	600,060	340,136	180,987

TABLE VIII: Performance when running NN-20, NN-50, NN-100. Strix$_{\text{best}}$ denotes the case with highest performance, while Strix$_{\text{128bit}}$ denotes the case when Strix achieves 128 bit security.

Scheme	Security	NN-20	NN-50	NN-100
Baseline-TFHE [38]	128-bit	64.60s	129.25s	263.54s
Strix$_{\text{128bit}}$ [32]	128-bit	434.44ms	1193.77ms	1511.77ms
Strix$_{\text{best}}$ [32]	80-bit	78.96ms	148.73ms	551.28ms
Trinity	128-bit	69.86ms	146.26ms	277.13ms

TABLE IX: Performance of Scheme Conversion algorithm (ms).

Scheme	$n_{\text{slot}} = 2$	$n_{\text{slot}} = 8$	$n_{\text{slot}} = 32$
Baseline-SC	364	492	1,168
Trinity	0.049	0.063	0.142

C. Compared scheme setting

For a thorough analysis for *Trinity*, we established several compared schemes.

CKKS. For CKKS, we established one compared scheme named *Trinity*-CKKS$_{\text{IP-use-EWE}}$. The difference between *Trinity*-CKKS$_{\text{IP-use-EWE}}$ and *Trinity* is that *Trinity*-CKKS$_{\text{IP-use-EWE}}$ uses EWE for the computation of IP, whereas *Trinity* uses some of the CU for this purpose.

TFHE. For TFHE, we set up three compared schemes: Morphling$_{\text{1GHz}}$, *Trinity*-TFHE$_{\text{w/ CU}}$ and *Trinity*-TFHE$_{\text{w/o CU}}$. Morphling$_{\text{1GHz}}$ has its frequency set to 1GHz, compared with the original design of Morphling. Both *Trinity*-TFHE$_{\text{w/ CU}}$ and *Trinity*-TFHE$_{\text{w/o CU}}$ maintain the same level of parallelism in the NTT unit as the FFT unit in Morphling. *Trinity*-TFHE$_{\text{w/ CU}}$ retains the architecture of *Trinity*, except it scales down the parallelism. *Trinity*-TFHE$_{\text{w/o CU}}$ is a fixed design that includes NTT units and a systolic array (SA) but lacks the capability for flexible mapping. The depth of SA in TFHE$_{\text{w/o CU}}$ is 12, which is consistent with the total depth of all CU in *Trinity*.

VI. Results

A. Performance

In this section, we compare the performance of *Trinity* against CPUs, GPUs, and state-of-the-art ASIC accelerators for the CKKS, TFHE, and Scheme Conversion tasks. Additionally, we conduct a series of comprehensive analyses to explain the sources of performance benefits.

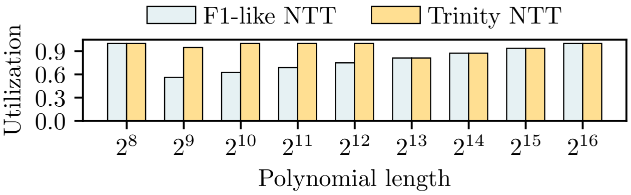


Fig. 9: Utilization comparison of NTT unit.

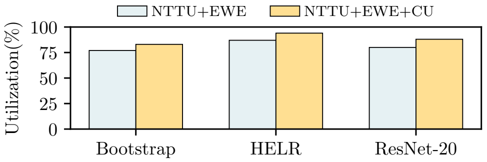


Fig. 10: Utilization of NTTU+EWE in *Trinity*-CKKS$_{\text{IP-use-EWE}}$ and NTTU+EWE+CU in *Trinity*.

TABLE X: Performance within hybrid-scheme apipilications.

Scheme	HE3DB-4096	HE3DB-16384
Baseline-Hybrid	3,012s	11,835s
SHARP+Morphling	5.64s	22.55s
Trinity	0.42s	1.68s

As shown in Table VI, *Trinity* achieves an average performance improvement over SHARP [22] by 1.49× and up to 1.85× when executing HELR. As shown in Table VII, in terms of PBS throughput, *Trinity* outperforms Morphling [31] by average 4.23× and up to 4.32× under Set-III. Additionally, as shown in Table VIII, when computing NN-x, *Trinity* demonstrates a speedup over Baseline-TFHE by an average 919.3× and up to 950.9× within the NN-100 model. Under the same security of 128-bit, compared with Strix, *Trinity* achieves a performance improvement of average 6.51×. Compared with the best case of Strix, *Trinity* achieves a performance improvement of average 1.31× with better security of 128-bit, while the best case of Strix achieves 80-bit security. When considering Scheme Conversion from TFHE to CKKS, as shown in Table IX, *Trinity* achieves an average speedup over baseline-SC by 7,814×. For hybrid FHE applications, *Trinity* outperforms baseline-hybrid on average by 7,107×. Compared with the SOTA accelerator, *Trinity* outperforms SHARP+Morphling on average by 13.42×.

B. Hardware efficiency for CKKS and TFHE

The performance impact of CU on CKKS. To validate the effectiveness of our design, we conducted a series of analyses. For CKKS workloads, we first examined the benefits of deploying CUs for NTT computation. As shown in Figure 9, the NTT design in *Trinity* demonstrates an average improvement in utilization by 1.2×, which can be attributed to the flexible mapping capabilities of CUs for NTT computation. Additionally, we analyzed the impact of using CUs for Inner Product computation. As shown in Figure 10, for CKKS workloads, *Trinity* shows a significant utilization improvement over *Trinity*-CKKS$_{\text{IP-use-EWE}}$ by 1.08×. This enhancement in utilization improves the *Trinity*'s performance for CKKS workloads. As shown in Figure 11, *Trinity* outperforms *Trinity*-

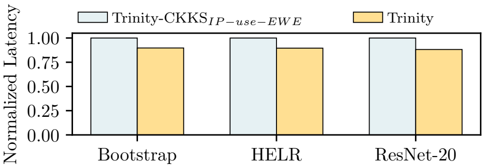


Fig. 11: Normalized latency comparison between *Trinity*-CKKS$_{\text{IP-use-EWE}}$ and Trinity within CKKS workloads (normalized to *Trinity*-CKKS$_{\text{IP-use-EWE}}$).

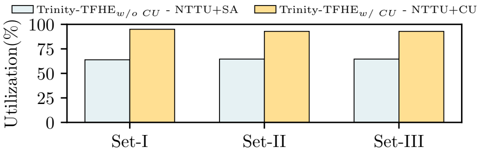


Fig. 12: Utilization of *Trinity*-TFHE$_{\text{w/o CU}}$ and *Trinity*-TFHE$_{\text{w/ CU}}$ when executing PBS.

TABLE XI: Circuit area and power.

Component	Area(mm^2)	Power(W)
2× NTTU	3.20	4.24
1×CU-1	0.18	0.31
4×CU-2	1.44	2.48
1×CU-3	0.55	0.93
AutoU	0.04	0.22
Rotator	2.40	8.57
EWE	1.87	4.47
VPU	0.05	0.07
NoC (intergroup and intragroup)	0.10	13.24
local buffer	6.45	1.41
cluster	16.28	35.94
4× cluster	65.12	143.76
inter-cluster NoC	20.60	27.00
scratchpad	41.94	26.80
HBM PHY	29.60	31.80
Total	157.26	229.36

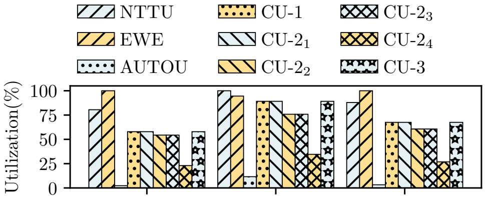


Fig. 13: Component utilization within CKKS workloads.

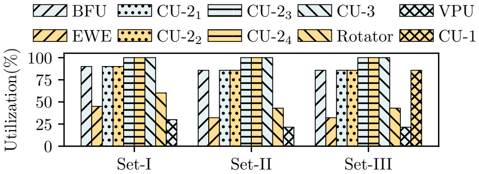


Fig. 14: Component utilization within TFHE PBS.

TABLE XII: Comparison with the state-of-the-art FHE accelerators. BW. denotes bandwidth, Cap. denotes capability.

	CraterLake [34]	SHARP [22]	Morphling [31]	*Trinity*
Scheme support	CKKS	CKKS	TFHE	CKKS; TFHE; CKKS⟷TFHE
Word Length	28-bit	36-bit	32-bit	36-bit
Core Freq.	1GHz	1GHz	1.2GHz	1GHz
Off-chip Mem BW.	1TB/s	1TB/s	310GB/s	1TB/s
On-chip Mem Cap.	282MB	198MB	11MB	191MB
On-chip Mem BW.	84TB/s	72TB/s	-	36TB/s(SPM); 135TB/s(local buffer)
Technology	12nm	7nm	28nm	7nm
Area	472.3mm^2 (12nm)	178.8mm^2 (7nm)	4mm^2(7nm) 13mm^2(12nm) 74mm^2(28nm)	157.26mm^2(7nm) 462.15mm^2(12nm)
Power	320W	-	53.00W	229.36W

CKKS$_{\text{IP-use-EWE}}$ by average 1.12× and up to 1.13× for ResNet-20.

The performance impact of CU on TFHE. As shown in Table VII, *Trinity*-TFHE$_{\text{w/o CU}}$ exhibits an average performance reduction by 27.1%, whereas *Trinity*-TFHE$_{\text{w/ CU}}$ achieves a significant performance improvement over Morphling by average 1.27×. The performance benefit of *Trinity*-TFHE$_{\text{w/ CU}}$ can be attributed to the balanced workload enabled by the flexible CU mapping and the reuse of CU in NTT computations. As shown in Figure 12, *Trinity*-TFHE$_{\text{w/ CU}}$ demonstrates an average utilization improvement over *Trinity*-TFHE$_{\text{w/o CU}}$ by 1.45×.

Utilization of other components. As shown in Figure 13, *Trinity* achieves an average utilization exceeding 48% when executing CKKS workloads. Similarly, Figure 14 shows that under three different parameter sets for TFHE PBS, *Trinity* maintains an average utilization above 64%. The consistently high utilization across both CKKS and TFHE workloads underscores the efficiency of *Trinity*, which is largely due to the capability of *Trinity* to balance workloads in both schemes.

C. Area and Power

Area. Table XI presents the circuit area of *Trinity* by components. Compared to the total area of SHARP and Morphling, *Trinity* achieves an area reduction of 15% and outperforms each individually. This result demonstrates the high computational efficiency of *Trinity*, which is attributed to its configurable architecture design.

Power. As illustrated in Table XII, in comparison with CraterLake, *Trinity* achieves a substantial reduction of 28.5% in circuit power while substantially outperforming CraterLake. This result indicates that *Trinity* has considerable power efficiency.

D. The overhead of supporting both CKKS and TFHE

To fully support both CKKS and TFHE, AutoU, EWE, Rotator and VPU are required in the accelerator. These components cannot be utilized in both schemes. Nonetheless, these components constitute only a small proportion of the total area of *Trinity*, which is 11.08%.

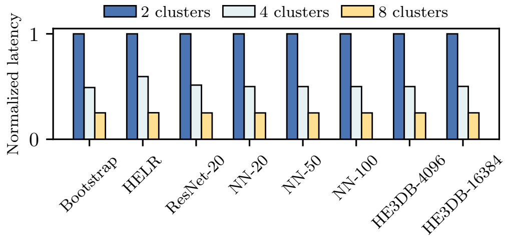
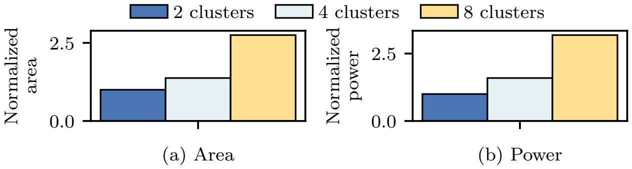


Fig. 15: Normalized latency within CKKS, TFHE and hybrid scheme applications under different accelerator configurations. Normalized to 2 clusters.



(a) Area

(b) Power

Fig. 16: Normalized area and power under different accelerator configurations. Normalized to 2 clusters.

E. Sensitivity study to the number of clusters

In this section, we conduct a sensitivity study to the cluster number. As shown in Figure 15, when the cluster number increases from 4 to 8, *Trinity* achieves a performance improvement of average 2.04×, with an area increment of 2× (shown in Figure 16(a)). On the contrary, as the cluster number decreases, the end-to-end performance is lower, while *Trinity* achieves the reduction of circuit area and power consumption by 28% and 36%. Therefore, the user can modify the cluster number to meet the requirements for performance or hardware overhead.

VII. Related Work

CKKS Accelerators. Domain-specific accelerators provide significant performance improvements due to highly customized hardware designs and specific dataflow optimizations for particular scenarios. Consequently, numerous CKKS accelerators have been proposed [3], [22]–[24], [33], [34], [36]. When executing CKKS workloads, these accelerators demonstrate substantial performance enhancements. However, they support only the CKKS scheme and cannot leverage the advantages of both the CKKS and TFHE schemes. Furthermore, most feature fixed designs, which are difficult to adapt to the diverse workload characteristics of the CKKS and TFHE schemes. In contrast, *Trinity* features a flexible design with configurable units that can adjust the computational component ratio for different kernels, thereby achieving balanced workloads across both schemes.

TFHE Accelerators. The TFHE scheme imposes significant computational and memory demands. To address these challenges, several accelerators for TFHE have been developed [19], [31], [32], [35]. These accelerators generally achieve higher throughput compared to conventional CPUs. Nonetheless, the use of FFT, which relies on floating-point arithmetic, adds to the hardware complexity and increases power consumption. Although some implementations utilize FFT based on fixed-point or integer arithmetic, they still cannot mitigate the approximation errors inherent in FFT computations. In contrast, *Trinity* utilizes NTT for TFHE computations through minor changes on the schemes, taking the NTT advantage of not inducing additional error.

Scheme Conversion Algorithm. CKKS schemes offer the benefit of SIMD computation, while TFHE schemes enable arbitrary function evaluation. Consequently, it is natural to seek methods that combine the strengths of both schemes. To this end, various Scheme Conversion algorithms and frameworks have been introduced [4], [26]. These algorithms effectively convert between the CKKS and TFHE schemes, operate in the TFHE domain, and then revert to the CKKS scheme.

Scheme Conversion on FPGA. Recently, an FPGA accelerator targets efficient CKKS bootstrapping using Scheme Conversion has been proposed [2]. This accelerator is implemented based on 8 FPGA cards as a distributed parallel system. Nonetheless, Trinity is the first work to explore the construction of multi-modal FHE scheme and Scheme Conversion support on one specific ASIC accelerator. *Trinity* not only provides efficient support for CKKS and TFHE, but also supports Scheme Conversion, enabling the runtime switching between CKKS and TFHE.

VIII. Conclusion

In this paper, we conduct a thorough analysis of the challenges involved in designing an accelerator compatible with both CKKS and TFHE schemes, including the conversions between them. We then introduce *Trinity*, a high-performance fully homomorphic encryption (FHE) accelerator that introduces several optimizations: 1) the consistent application of NTT within both the CKKS and TFHE frameworks to minimize approximation errors and reuse computational components during computations; 2) the configurable units that facilitate flexible data mapping across different cryptographic schemes. Experimental results indicate that *Trinity* markedly outperforms SHARP, the leading CKKS accelerator, with a 1.49× average improvement in performance for CKKS workloads, and exceeds Morphling, the advanced accelerator for TFHE workloads, by achieving a 4.23× average performance increase for TFHE PBS. Moreover, *Trinity* demonstrates a 15% reduction in circuit area relative to the combined area of SHARP and Morphling.

Acknowledgment

The authors thank Liang Chang and the anonymous reviewers for their valuable comments. This research was supported by the National Key R&D Program of China (Grant No.2023YFB4503201) and the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No.XDB44030200). Qian Lou and Dingyuan Cao contributed independently to this work, without support from the aforementioned funding sources.

References

[1] "TPC BENCHMARK™ H," Transaction Processing Performance Council, San Francisco,CA, Tech. Rep., 2022.

[2] R. Agrawal, A. Chandrakasan, and A. Joshi, "HEAP: A Fully Homomorphic Encryption Accelerator with Parallelized Bootstrapping," in *International Symposium on Computer Architecture*, 2024.

[3] 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.

[4] A. Al Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee *et al.*, "Openfhe: Open-source fully homomorphic encryption library," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2022, pp. 53–63.

[5] D. H. Bailey, "Ffts in external or hierarchical memory," *The journal of Supercomputing*, vol. 4, pp. 23–35, 1990.

[6] S. Bian, Z. Zhang, H. Pan, R. Mao, Z. Zhao, Y. Jin, and Z. Guan, "He3db: An efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2930–2944. [Online]. Available: https://doi.org/10.1145/3576915.3616608

[7] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Annual cryptology conference*. Springer, 2012, pp. 868–886.

[8] 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.

[9] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. 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*, vol. 62, no. 1, pp. 157–166, 2014.

[10] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient homomorphic conversion between (ring) lwe ciphertexts," in *Applied Cryptography and Network Security*, K. Sako and N. O. Tippenhauer, Eds. Cham: Springer International Publishing, 2021, pp. 460–479.

[11] 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.

[12] 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.

[13] 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.

[14] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5*. Springer, 2021, pp. 1–19.

[15] L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.

[16] 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)*. IEEE, 2023, pp. 922–934.

[17] K. Han, S. Hong, J. H. Cheon, and D. Park, "Logistic regression on homomorphic encrypted data at scale," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 9466–9471.

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

[19] 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.

[20] M. Joye and M. Walter, "Liberating tfhe: Programmable bootstrapping with general quotient polynomials," in *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–11. [Online]. Available: https://doi.org/10.1145/3560827.3563376

[21] A. B. Kahng, B. Lin, and S. Nath, "Orion3. 0: A comprehensive noc router estimation tool," *IEEE Embedded Systems Letters*, vol. 7, no. 2, pp. 41–45, 2015.

[22] 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.

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

[24] 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.

[25] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, "Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions," in *International Conference on Machine Learning*. PMLR, 2022, pp. 12 403–12 422.

[26] W.-j. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "Pegasus: bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1057–1073.

[27] C. V. Mouchet, J.-P. Bossuat, J. R. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," 2020, pp. 6. 64–70. [Online]. Available: http://infoscience.epfl.ch/record/299025

[28] H. Nejatollahi, S. Shahhosseini, R. Cammarota, and N. Dutt, "Exploring energy efficient quantum-resistant signal processing using array processors," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 1539–1543.

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

[30] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, pp. 252–264, 1968.

[31] Prasetiyo, A. Putra, and J.-Y. Kim, "Morphling: A throughput-maximized tfhe-based accelerator using transform-domain reuse," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 249–262.

[32] 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," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1319–1331. [Online]. Available: https://doi.org/10.1145/3613424.3614264

[33] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[34] 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.

[35] M. Van Beirendonck, J.-P. D'Anvers, F. Turan, and I. Verbauwhede, "Fpt: A fixed-point accelerator for torus fully homomorphic encryption,"

in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 741–755. [Online]. Available: https://doi.org/10.1145/3576915.3623159

[36] 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.

[37] 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)*, 2022, pp. 1–7.

[38] Zama, "Concrete: TFHE Compiler that converts python programs into FHE equivalent," 2022, https://github.com/zama-ai/concrete.