Neo: Towards Efficient Fully Homomorphic Encryption Acceleration using Tensor Core

Dian Jiao*
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
jiaodian@iie.ac.cn

Xianglong Deng*
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
dengxianglong@iie.ac.cn

Zhiwei Wang
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
wangzhiwei@iie.ac.cn

Shengyu Fan
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
fanshengyu@iie.ac.cn

Yi Chen
Huazhong University of Science and Technology
Wuhan, China
chenyi22@hust.edu.cn

Dan Meng
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
mengdan@iie.ac.cn

Rui Hou
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
Beijing, China
hourui@iie.ac.cn

Mingzhe Zhang†
Computing System Lab
Ant Group
Beijing, China
smartzmz@gmail.com

Abstract

Fully Homomorphic Encryption (FHE) is an emerging cryptographic technique for privacy-preserving computation, which enables computations on the encrypted data. Nonetheless, the massive computational demands of FHE prevent its further application to real-world workloads. To tackle this problem, several studies focus on the ASIC-based acceleration for FHE. However, the rapid evolution of FHE algorithms poses challenges to the generality of ASIC accelerator design. By contrast, a number of works rely on GPGPUs for FHE accelerations, due to the high parallelism and flexibility provided by GPGPUs.

In this work, we propose a GPGPU-based acceleration solution that supports the Cheon-Kim-Kim-Song (CKKS) scheme by further exploiting Tensor Core(TCU) capabilities. In our study, we first analyze the FHE applications based on GPGPUs and emphasize the poor data reuse of some FHE kernels. Subsequently, we highlight the inefficient usage of TCUs due to the unused *FP64* components. To address these issues, we propose a novel FHE acceleration solution based on GPGPU named Neo, which features: 1) algorithmic optimizations that transform scalar multiplication and element-wise multiplication into matrix multiplication; 2) data layout optimizations for FHE kernels to optimize for matrix multiplications; 3) Accelerating matrix multiplication with the floating-point components in TCUs to leverage the strengths of various components within the GPU. Experimental results demonstrate that within NVIDIA A100 GPGPU, Neo outperforms TensorFHE by 3.28×.

CCS Concepts

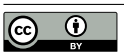
• **Computer systems organization** → **Single instruction, multiple data**; • **Security and privacy**;

Keywords

Fully Homomorphic Encryption, GPGPU, CKKS, Tensor Core

ACM Reference Format:
Dian Jiao, Xianglong Deng, Zhiwei Wang, Shengyu Fan, Yi Chen, Dan Meng, Rui Hou, and Mingzhe Zhang. 2025. Neo: Towards Efficient Fully

*Both author contributed equally to this research.
†Corresponding author.



ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731408

Homomorphic Encryption Acceleration using Tensor Core. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731408

1 Introduction

In the era of cloud computing, the proliferation of models has led to an unprecedented surge in the volume of user data being shared with remote servers. This data now extends to the most intimate and confidential realms, such as personal financial transactions and health records [41, 46]. Despite heightened awareness of the critical need for data protection, conventional cybersecurity measures have not kept pace with the evolving landscape of digital threats. A particularly concerning aspect is the risk posed by side-channel attacks, which exploit the exposure of plaintext data, presenting a significant threat to the integrity and confidentiality of user information [21, 42]. Therefore, exposing plaintext data online facilitates malicious actions by attackers in the AI-advanced era.

To address this problem, Fully Homomorphic Encryption (FHE) has become popular for privacy data protection. FHE allows direct computation on encrypted data without decryption during the application computation on third-party resources, while users can still obtain the desired computation results [13]. Thus, FHE fundamentally eliminates the problem of privacy data leakage. Regardless of the attacker's intelligent model, they cannot obtain usable plaintext information from encrypted data due to strong lattice cryptography protection [13].

While FHE provides powerful protection for privacy data, the computational workload significantly increases compared to unencrypted data. Thus, executing FHE applications on CPU platforms is very expensive, prompting many researchers to design various acceleration methods on different platforms to reduce execution time [1, 11, 12, 22]. GPGPUs are the most widely used computing platform in the world, with millions already sold and capable of forming mature computing systems [5]. Thus, designing acceleration methods on the GPGPU platform is the most cost-effective and practical solution for utilizing FHE in real-world applications.

Prior works optimize dataflow to improve data reuse through kernel fusion, achieving significant performance improvements compared to CPU solutions. However, this approach lacks of algorithm optimization, leading to underutilization of TCUs [22]. Some optimization works involve mapping the Number-Theory Transformation (NTT) to TCUs by splitting large values into low-bit computations [12], which increases computational complexity. Additionally, previous work only leveraged TCUs to compute the NTT and lacked supporting kernels, resulting in the underutilization of TCUs for all FHE kernels.

Directing attention to these limitations, we present Neo, a high-performance GPGPU-base solution. The contributions of this work can be summarized as follows:

- We proposed a novel method that transforms critical FHE kernels - BConv and IP - into matrix multiplication, and demonstrate the first implementation of their acceleration through TCU. This dual-pronged approach simultaneously enhances data reuse, while more effectively harnessing the TCUs' superior computational density.

Table 1: Symbols in this paper

Symbol	Definition
N	Degree of a polynomial.
L	Maximum level of ciphertexts.
l	Current level of ciphertexts.
R_a	Cyclotomic polynomial ring, $\mathbb{Z}_a/(X^N+1)$.
Q	$\prod_{i=0}^{L} q_i$, the product of all prime moduli.
P	$\prod_{i=0}^{K} p_i$, the product of special prime moduli.
PQ	The product of P and Q.
T	$\prod_{i=0}^{\alpha'} t_i$, the product of prime moduli mutually exclusive with QP, used in KLSS method.
$BatchSize$	The number of ciphertexts batched in one operation.
d_{num}	The number of digits in KeySwitch. [19]
α	$\alpha = \lceil \frac{L+1}{d_{num}} \rceil$, the number of limbs in a digit.
β	$\beta = \lceil \frac{l+1}{\alpha} \rceil$, the number of groups after digit decomposition.
$\widetilde{\alpha}$	The hyperparameter to ascertain α' and $\widetilde{\beta}$ in KLSS method.
α'	The number of limbs in one group in R_T in KLSS method, related to α, β, and $\widetilde{\alpha}$, etc.
$\widetilde{\beta}$	$\widetilde{\beta} = \lceil \frac{l+\alpha+1}{\widetilde{\alpha}} \rceil$, the number of digits after Inner Product in KLSS method.
λ	Security parameters.
$WordSize$	The bit width of q_i and p_i.
$WordSize_T$	The bit width of t_i in KLSS method.

- we proposed a TCU optimization to leverage the float-point components in TCUs, enhancing the utilization efficiency of TCU. This may inspire future works to focus on the utilization of float-point and matrix multiplication components.
- Our work demonstrate that leveraging architectural insights to guide software optimization can effectively enhance overall system performance through improved hardware utilization/efficiency and optimized resource allocation, rather than reduce the algorithmic complexity. The proposed approach establishes an effective methodology for bridging architectural knowledge and software optimization.

2 Background

This section provides a brief overview of the CKKS schemes and the KLSS method. We briefly introduce the GPGPU architecture and its precision support in our implementation. Table 1 lists the notations used throughout the paper.

2.1 CKKS Scheme

The CKKS (Cheon-Kim-Kim-Song) scheme is a homomorphic encryption algorithm that supports approximate arithmetic on encrypted data [8]. This scheme is particularly suited for applications where exact precision is not critical but efficient computation is required, such as in machine learning [36] and data analysis [43]. CKKS allows for the addition and multiplication of ciphertexts, preserving the structure of the underlying plaintexts with a controlled noise growth. The efficiency and practicality of CKKS make it a popular choice for secure data processing in various fields.

To achieve high performance with the CKKS scheme, it packs $N/2$ complex numbers (referred to as *slots*) into one plaintext polynomial, denoted as $\mathbf{m}(X) = \sum_{i=0}^{N-1} d_i X^i$ for a given N. This plaintext is then encrypted into a ciphertext comprising two polynomials, denoted as $\mathbf{ct}(X) = (\mathbf{c}_0, \mathbf{c}_1)$, as shown in Eq. 1.

$$\begin{aligned}\mathbf{ct}(X) &= (c_0, c_1) \\ &= (b(X), a(X)) \\ &= (a(X) \cdot s(X) + m(X) + e(X), a(X))\end{aligned} \tag{1}$$

Here, $s(X)$ is the secret key, $a(X)$ is a random polynomial, and $e(X)$ is a small Gaussian error polynomial required for the LWE security guarantee [24]. To obtain the decrypted plaintext from the ciphertext, we compute $m^{'}(X) = \langle \mathbf{ct}(X), s(X) \rangle = c_0 - c_1 \cdot s(X) = m(X) + e(X)$, which approximates the original $\mathbf{m}$(X) with small errors. Next, we introduce the basic operations of the CKKS scheme, which serve as the building blocks for more complex FHE) operations, such as convolution and other linear computations. We define the ciphertexts $\mathbf{ct}_0$ $((a_0, a_1))$ and $\mathbf{ct}_1$ $((b_0, b_1))$, and the plaintext $\mathbf{m}$. Using these definitions, we present the primitive operations of CKKS.

- **HMULT**. This operation performs inter-ciphertext multiplication for $\mathbf{ct}_0$ and $\mathbf{ct}_1$. It first generates three polynomials: d_0, d_1, and d_2, where $d_0 = a_0 \cdot b_0$, $d_1 = a_0 \cdot b_1 + a_1 \cdot b_0$, and $d_2 = a_1 \cdot b_1$. Next, d_2 undergoes a KeySwitch operation to maintain the decryptability of the resultant ciphertext. Finally, the output of the KeySwitch operation is added to d_0 and d_1. Although the KeySwitch operation preserves ciphertext decryptability, it significantly increases noise, necessitating a Rescale operation to reduce this noise increment.
- **PMULT**. This operation performs plaintext-ciphertext multiplication, where a plaintext $\mathbf{m}$ is multiplied with a ciphertext $\mathbf{ct}$. The resulting product is another ciphertext that encrypts the product of the original plaintext and the plaintext corresponding to the original ciphertext. This operation is more efficient than HMULT because it does not require a KeySwitch operation.
- **HADD**. This operation performs inter-ciphertext addition, combining two ciphertexts $\mathbf{ct}_0$ and $\mathbf{ct}_1$. The resulting ciphertext encrypts the sum of the plaintexts corresponding to the original ciphertexts.
- **PADD**. This operation performs plaintext-ciphertext addition, where a plaintext $\mathbf{m}$ is added to a ciphertext $\mathbf{ct}$. The resulting ciphertext encrypts the sum of the original plaintext and the plaintext corresponding to the original ciphertext. Similar to HADD, this operation is efficient and does not increase noise.
- **HROTATE**. This operation rotates the slots of a ciphertext $\mathbf{ct}$ by a specified number of positions. The rotation is performed within the polynomial ring representation of the ciphertext. To maintain decryptability after rotation, a KeySwitch operation is required. This operation is useful for homomorphic evaluations that require data reorganization.
- **Rescale**. This operation reduces the noise level in a ciphertext after a sequence of homomorphic operations. The Rescale operation involves scaling down the ciphertext and its associated modulus to reduce accumulated noise, thereby ensuring the continued decryptability and correctness of the ciphertext. This step is crucial for maintaining the precision and accuracy of homomorphic computations.
- **Double Rescale** (DS). This operation is primarily used in Bootstrapping to ensure precision during application execution, especially with small word length configurations [25]. It requires the last two levels of limbs to generate noise reduction limbs, unlike **Rescale**, which only requires the last level. This operation also consumes two ciphertext levels, which significantly affects the execution conditions and performance of FHE applications. **DS is an essential operation when the *WordSize* is smaller than 36 bits.**

Table 2: Kernel Complexity of Hybrid Method and KLSS Method

Breakdown	Hybrid	KLSS
Mod Up	$\beta l \alpha$	$\beta \alpha \alpha'$
NTT	$\beta(l+\alpha)$	$\beta\alpha'$
Inner Product	$2\beta(l+\alpha)$	$\widetilde{\beta}\beta\alpha'$
Inverse NTT	$2\beta(l+\alpha)$	$2\widetilde{\beta}\alpha'$
Recover Limbs	-	$2\alpha'(l+\alpha)$
Mod Down	$2(l\alpha+l)$	$2(l\alpha+l)$

2.2 Advanced KeySwitch Method

As introduced in the previous section, the KeySwitch operation is crucial for FHE applications. It ensures that ciphertexts remain decryptable and significantly impacts application performance. The advanced KeySwitch method, which is called KLSS[28], involves six steps: *Mod Up*, *NTT*, *IP* (*Inner Product*), *INTT*, *Recover Limbs*, and *Mod Down*, as illustrated in Fig. 5. The primary distinction between KLSS method and Hybrid method lies in that the majority of computations in KLSS occur over the extended ring R_T, thus transforming the complexity of each step. Since the prime moduli t_i is selectable, its bit width $WordSize_T$ also becomes a variable parameter.

Applied both in the KLSS method and the Hybrid method, digit decomposition (also known as gadget decomposition) is a step at the initial stage of *Mod Up* to decompose the input ciphertext $\mathbf{c} = (c_0, c_1, \ldots, c_{l-1})$. $\mathbf{c}$ is decomposed into β groups, denoted by $(\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{\beta-1})$, and each $\mathbf{c}_i$ is referred to as a digit.

Mod Up in Hybrid method maps each digit to the composite ring R_{PQ} (Eq. 2). In contrast, KLSS method maps each digit to R_T (Eq. 3).

$$\text{Hybrid: } (c_i^0, \ldots, c_i^{\alpha-1}) \xrightarrow{BConv} (\hat{c}_i^0, \ldots, \hat{c}_i^{l+\alpha}) \tag{2}$$

$$\text{KLSS: } (c_i^0, \ldots, c_i^{\alpha-1}) \xrightarrow{BConv} (\hat{c}_i^0, \ldots, \hat{c}_i^{\alpha'}) \tag{3}$$

Since α' is generally much smaller than $l+\alpha$, the complexity of *Mod Up* and *NTT* in KLSS method is lower, as shown in Table 2.

KLSS method exhibits higher complexity of *IP* due to its requirement of performing multiplication-accumulation operations with $\widetilde{\beta} \times \beta$ evaluation keys. After *IP*, β groups of limbs are transformed into $\widetilde{\beta}$ groups, which also impacts the complexity of the subsequent *INTT*. Besides, *Recover Limbs* is required in KLSS method to recover limbs back into R_{PQ}. Notwithstanding these overhead, judicious parameter selection enables the KLSS method to achieve a lower overall complexity. However, a constraint must be satisfied to ensure security:

$$\alpha' \geq \left\lceil \frac{\log(2\beta N \cdot B\tilde{B})}{WordSize_T} \right\rceil \tag{4}$$

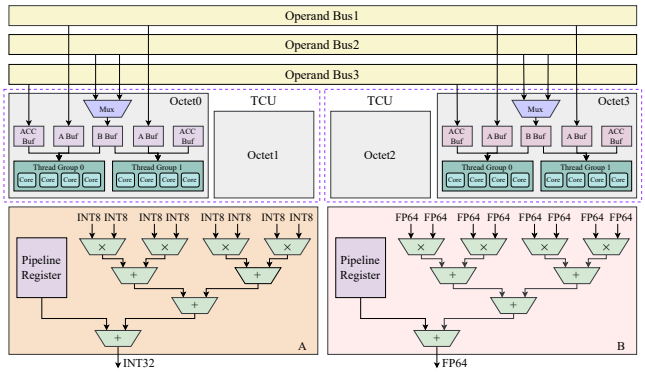


Figure 1: Abstract Schematic of Tensor Core Microarchitecture. Component A is used for computing input with the data type of *INT8*, while Component B is used for computing input with the data type of *FP64*.

β, B and $\tilde{B}$ are solely dependent on $\widetilde{\alpha}$ and d_{num}. Therefore, selecting a larger $WordSize_T$ can reduce the overall algorithmic complexity. However, due to hardware resource constraints, the choice of $WordSize_T$ cannot be excessively large, which will be analyzed in more detail in Section 3.

2.3 GPGPU Architecture and Precision Support

The evaluation keys depend on the specific FHE operations. For the `HMULT` operation, the evaluation key is concerned with the noise generated by `KSGen(`s`,` s^2`)`. For the `HROTATE` operation, the evaluation key also incorporates the rotation index, generated by `KSGen(`s`, Rot(`s`, Index))`. Additionally, the `IP` operation requires two sets of $\beta\widetilde{\beta}\alpha'$ polynomial keys, which significantly impact overall performance. The *Mod Up*, *Recover Limbs*, and *Mod Down* steps involve converting or grouping limbs in different modulus spaces. Thus, it needs the `BConv` (Base Conversion) to complete this function, thus the throughput of the `BConv` kernels greatly affects the efficiency of these steps. The `NTT` operation is crucial for accelerating polynomial multiplication. Thus, the performance of key kernels, such as `NTT`, `BConv`, and `IP`, primarily determines the overall execution efficiency. Therefore, in this paper, we will focus on optimizing these kernels to enhance the entire application's performance.

Given the high parallelism, GPGPU has become promising hardware for accelerating CKKS[12, 22]. GPGPU features an execution model of Single Instruction Multiple Data (SIMD) and is composed of multiple Streaming Multiprocessors (SMs). To suit the requirement of AI, except for the CUDA core, the SM in modern GPGPU includes TCUs, which offer high-performance support for matrix multiplications.

TCUs support both high-precision and low-precision data types including *FP64* and *INT8*, as depicted in Fig. 1. For the NVIDIA A100 GPU architecture utilized in this study, the CUDA Core architecture delivers a peak FP64 performance of 9.7 TFLOPS, while the TCU achieves 19.5 TFLOPS for FP64 operations[35] – approximately double the throughput. Notably, TCU's peak INT8 computational capacity reaches 624 TFLOPS, significantly surpassing its FP64 throughput. However, as will be empirically demonstrated in Section 3, the FP64 components exhibit superior performance under the high-bit-width requirements inherent to FHE workloads. In this work, we exploit the TCU's capabilities to offer high-performance support for CKKS.

3 Motivation

3.1 Rationale for FHE Optimization on GPGPU

While numerous studies have proposed ASIC designs for accelerating FHE, which exhibit impressive performance, GPGPU holds unique and irreplaceable significance and value for FHE acceleration. The advantages of using GPGPU are as follows:

- GPGPUs are already widely deployed in data centers, leveraging existing infrastructure for seamless integration with new workloads. As commercial hardware, they offer significant cost efficiency and benefit from a mature application ecosystem, facilitating practical implementation and widespread adoption.
- FHE is undergoing rapid development, with frequent algorithmic iterations. GPGPUs, characterized by their high programmability and flexibility, are well-suited to accommodate the evolving demands of such algorithms.
- The computational and memory resources of GPGPUs are inherently fixed by their hardware architecture, providing a well-defined research boundary for algorithm optimization. Investigating methods to accelerate algorithms under resource-constrained conditions not only enhances the computational efficiency of GPGPUs but also offers valuable theoretical insights and practical inspiration for the design of future dedicated accelerators.

Given these advantages, GPGPU-based FHE acceleration schemes merit deeper exploration. However, the inherent architectural constraints of GPGPUs mentioned above also pose significant challenges. Thus, adopting algorithm-architecture co-optimization strategies is essential to achieve efficient and scalable FHE solutions on general-purpose parallel computing platforms.

3.2 Support for variable *Wordsize*

Previous implementations, like TensorFHE, have only utilized *WordSize* smaller than 32 bits. However, SHARP[25] has demonstrated that *WordSize* of 36 bits is essential for ensuring precision. A larger *WordSize* has rendered 32-bit integer type inadequate for storing polynomials. Although *WordSize* is only 36 bits, full 64-bit computational width is needed in GPGPU, which approximately halves the GPGPU's throughput, representing an actual waste of hardware resources. To this issue, adopting an alternative novel `KeySwitch` method, KLSS, presents a promising solution.

The KLSS method transfers the limbs to R_T with a selective *WordSize*, represented as $WordSize_T$, achieving a reduction in overall complexity, as introduced in Section 2. Unlike the most frequently utilized `KeySwitch` approach, the Hybrid method, where parameters are independent of *WordSize*, the parameter α' in the KLSS method varies with $WordSize_T$. Specifically, the larger the $WordSize_T$, the smaller α' becomes, and consequently, the overall complexity of the KLSS method decreases.

Nevertheless, a larger $WordSize_T$ is not inherently better. Since almost all computations in the KLSS method are performed in R_T, $WordSize_T$ also affects the complexity of some kernels, especially

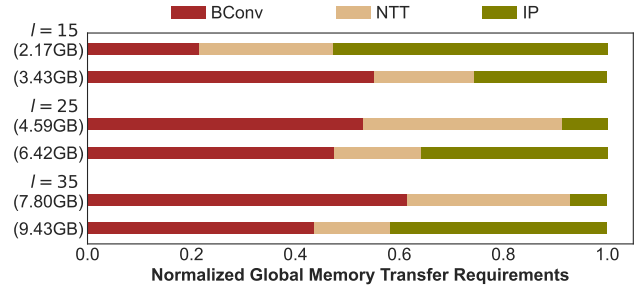


Figure 2: The proportion of data transfer of BConv, IP, and NTT kernels in the total data transfer of `KeySwitch` at different levels(l). Each level consists of two bars: the upper bar represents the data transfer requirements of the kernels in Hybrid method, while the lower bar indicates that in KLSS method. The total amounts of data transfer requirements are marked next to each bar.

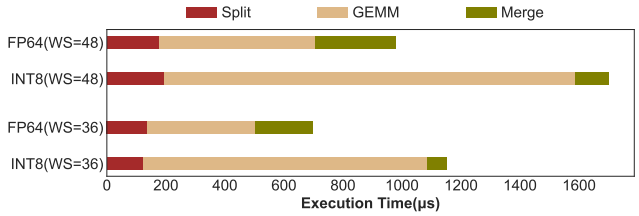


Figure 3: The comparison of the time required for three steps of matrix multiplication using *INT8* and *FP64* types. The displayed time is the total time of performing matrix multiplication with $M \times N \times K$ parameters corresponding to $2^{19} \times 16 \times 16$. WS represents *WordSize* in the figure.

those utilizing TCUs. TCUs natively support matrix multiplication operations only for limited data types, with the maximum being FP64 or INT32, necessitating the adoption of Booth's algorithm to decompose wide-bit-width operands into multiple segments for partial computation. This architectural constraint of GPGPU introduces the ***Challenge 1: Adopting a larger $WordSize_T$ leads to increased splitting and merging operations, as well as a higher number of partial computations.*** Consequently, the larger the $WordSize_T$, the higher the complexity of one NTT operation. Therefore, there is a trade-off between the complexity of the algorithm and the complexity introduced by hardware implementation.

3.3 Memory Requirement Analysis for FHE applications

We analyze the global memory requirements for `KeySwitch` during execution on the GPGPU platform within the parameter of Set-B as shown in Table. 4. As illustrated in Fig. 2, in both the Hybrid and KLSS methods, the global memory transfer requirements for BConv and IP constitute the majority in the entire `KeySwitch` process, achieving 43.4% and 41.8% when $l = 35$ in KLSS method.

However, fetching data from global memory to SM's local memory incurs high latency, which adversely affects application execution time [33]. Therefore, to further improve performance for FHE applications, it is crucial to reduce global memory access times. We have observed that Bconv and IP exhibit a significant amount of element-wise multiplication operations, with limbs being accessed multiple times as elements in these multiplications. Consequently, each limb is transferred to local memory on multiple occasions, resulting in substantial I/O overhead. This raises ***Challenge 2: Conventional computation-memory access paradigms introduce inefficient redundant memory accesses, resulting in substantial additional memory transfer requirements.***

Harnessing the superior spatial locality of matrix multiplication, we advocate for its implementation to replace the element-wise multiplications and aggregation procedures. This approach is projected to elevate the efficiency of data reuse, consequently diminishing the incidence of memory transfer.

3.4 Inefficient Usage of TCUs

TCUs are specialized processing units integrated within GPGPU, designed to accelerate the execution of mixed-precision GEMM (General Matrix Multiplication), which has the following form:

$$D = \alpha A \times B + \beta C$$

With α and β set to 1 and 0, TCUs can perform standard matrix multiplication. In prior work[12], the utilization of TCUs is limited to NTT operation, relying solely on integer components. Besides, TCUs only support a limited set of shapes for the operand matrix A, B, and C (mentioned as fragment) across different data types. The parameters M, N, and K represent the three dimensions of matrix multiplication operation:

- M is the number of rows from A that a warp computes.
- N is the number of columns from B that a warp computes.
- K is the number of columns in A (and equivalently, the number of rows in B) that are involved in the intermediate multiplication step before accumulation.

The fragment shape for the matrix multiplication of type *INT8*, are limited to three configurations: $16 \times 16 \times 16$, $32 \times 8 \times 16$ or $8 \times 32 \times 16$. For *FP64* matrix multiplication, TCUs only support the shape configuration of $8 \times 8 \times 4$. Due to the larger fragment shapes supported by *INT8* components, when the dimension of the input matrix is relatively small, padding will be inevitably introduced, leading to wasted computation. This constitutes ***Challenge 3: The fragment shape of INT8 components is incompatible with low-dimensional matrix operations, and their use imposes excessive decomposition of large-integer computations, thereby failing to fully exploit the computational advantages of TCU.***

In Fig. 3, we compare the time required to compute 36-bit and 48-bit integer matrix multiplication using *FP64* and *INT8* types. Although the *INT8* components perform one matrix multiplication of the same size much faster than the *FP64* components, it can be seen from Fig. 3 that the total time required to perform *INT8* matrix multiplications is more than the other, owing to its complexity when handling big integer. After splitting the 36-bit integer matrices A and B into 5 *INT8* matrices, a total of 25 matrix multiplications in a cross manner are required.

In contrast, emulating a 36-bit integer matrix multiplication requires only three *FP64* matrix multiplications. The *FP64* format offers 53 bits of precision, which is sufficient to represent integers up to 2^{53} without loss of precision. Consequently, it is feasible to divide matrix B into three matrices of *FP64* type, with each storing 12 bits of each 32-bit integer. During matrix multiplication, the

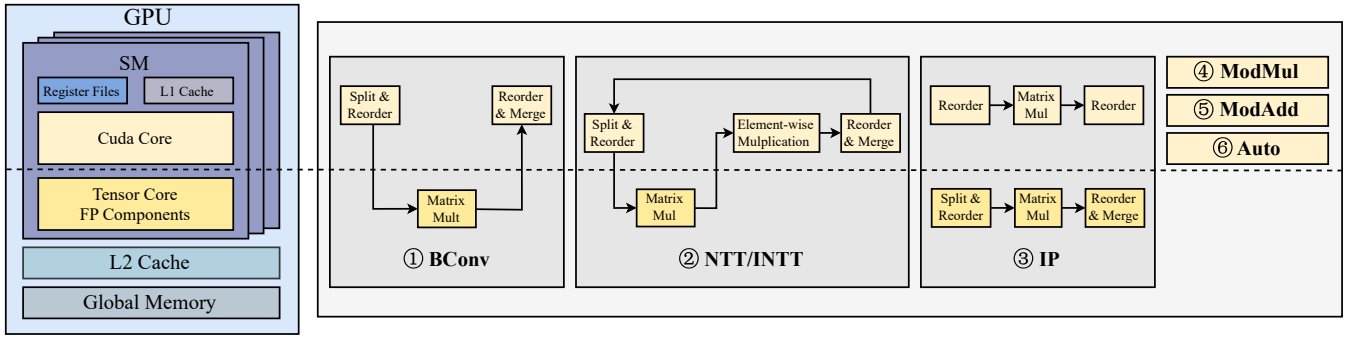


Figure 4: The mapping relationships between six kernels and computing components in GPGPU, including CUDA Cores and *FP64* components in TCUs. Among them, BConv, IP, and NTT kernels are the focus of our optimization, and their respective steps are shown in detail how they are mapped to different computing components. The other three kernels do not involve matrix multiplication, so only CUDA Cores are used for computation.

bit-width of the computational result is guaranteed not to exceed $2^{36} \times 2^{12} \times 16 = 2^{52} < 2^{53}$. The multiplication by 16 is due to the fact that the maximum value of the K dimension in practice is 16, hence it is necessary to consider the maximum bit-width resulting from the accumulation of 16 computational outcomes.

According to Fig. 3, when *WordSize* is 36, the computation speed of *FP64* components is 1.65 times that of *INT8* components. When *WordSize* is increased to 48, the 48-bit integers need to be split into 6 parts for computation with *INT8* components, resulting in a Booth complexity of 36 for matrix multiplication. In contrast, *FP64* components only require a complexity of $2 \times 2 = 4$, achieving a computation speed that is 1.74 times that of *INT8*. Consequently, the *FP64* components are a more advantageous choice for computations involving larger *WordSize* compared to the *INT8* components.

3.5 Opportunity

- **Algorithm optimization**: We observe that it is possible to transform BConv and IP from element-wise multiplication computation to matrix multiplication computation. This provides an opportunity to reduce the memory transfer requirements by exploiting the data reuse in matrix multiplication, thus addressing ***Challenge 2***.
- **TCU optimization**: As analyzed in the previous subsection, 36-bit and 48-bit matrix multiplications using *FP64* types are both faster than that using *INT8* types. As a more efficient alternative, the utilization of *FP64* components in TCU holds significant value for solving ***Challenge 1*** and ***Challenge 3***.
- **Trade-off between algorithm and hardware:** Selecting a larger $WordSize_T$ can reduce the overall algorithmic complexity of the KLSS method, but it leads to an increase in the Booth complexity of matrix multiplications computed on TCUs. Concerning ***Challenge 1***, finding the right balance between these two aspects is worth investigating.

4 Neo

4.1 Overview

We introduced a brand-new implementation of the CKKS scheme, named Neo. The foundation of Neo is an enhanced set of basic kernels, including BConv, NTT/INTT, IP, ModMUL, ModADD, and AUTO, as demonstrated in Fig. 4. These kernels and their mapping relationships to different components of GPGPU are also displayed. Within the kernels, steps that share the same color as the CUDA Cores are mapped onto the CUDA Cores, while steps that match the color of the TCUs are mapped onto the TCUs. A particular exception is IP, which has two distinct execution flows on both the CUDA Cores and the TCUs, indicating that IP employs two different mapping strategies under varying parameters, which will be elucidated in detail in the subsequent subsection.

In Neo, all operations are composed of the six basic kernels mentioned above. In Fig. 5, we use HMULT and HROTATE as examples; these two operations encompass all types of basic kernels, and the data flow between the kernels is also illustrated.

We have observed that the computational process involving "element-wise multiplying multiple limbs by specific arrays and accumulating the results" can be transformed into the form of matrix multiplication. NTT has already been implemented in the form of matrix multiplication. BConv, and IP in KLSS methods can also be converted into matrix multiplication form. This transformation doesn't only improve the data reuse of computation on CUDA Cores but also lays the groundwork for leveraging TCUs to expedite matrix multiplication operations. Based on this, each kernel is mapped to the most efficient computational units available in Neo.

4.2 Algorithm Optimization for Data Reuse

4.2.1 BConv Optimization. BConv kernel converts a set of residue polynomials ($level = \alpha$) to another set ($level = \alpha'$). In the original BConv algorithm, each coefficient of all limbs is sequentially read, scalar-multiplied by the base conversion factors $\widehat{\mathbf{q}}^{-1}$ and $\widehat{\mathbf{q}}$, and accumulated to generate a new coefficient at the corresponding position in the output limbs, as shown in Algorithm 1. This scalar multiplication and accumulation process must be executed once for each output level. Consequently, each coefficient is accessed from global memory α' times, resulting in suboptimal data reuse.

Due to the originally inefficient computational approach, we have designed an improved BConv algorithm, which is shown in Algorithm 2. The input of BConv can be viewed as a three-dimensional tensor with dimensions of $\alpha \times BatchSize \times N$, which means there are a total of α levels, and each level has $BatchSize$ limbs with N coefficients. We decompose BConv into two steps. The

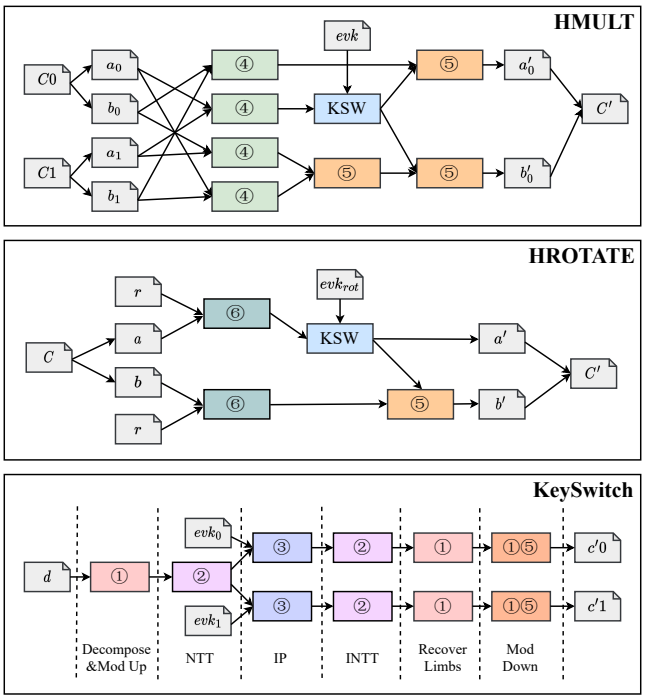


Figure 5: HMULT and HROTATE are selected as examples to demonstrate how kernels form the workflow of operations, and the workflow of `KeySwitch`, which is included in both operations, is also presented in detail. ①, ②, ③, ④, ⑤, ⑥ represent the six kernels labeled in Fig. 4

.

first step involves scalar multiplication and reordering the tensor to the dimensions of $N \times BatchSize \times \alpha$. The rearranged tensor A' can thus use its lower two dimensions as matrices to perform matrix multiplications with the $\alpha \times \alpha'$ matrix B composed of $\widehat{\mathbf{q}}$.

4.2.2 IP Optimization. In the previous works, the IP kernel is constructed using the Modular Multiplication (ModMUL) kernel, which performs element-wise multiplication under a modulus, as shown in Algorithm 3. The original algorithm of the IP kernel was characterized by poor data reuse. There exists a total of β groups, each with $BatchSize$ limbs at all α' levels. For each limb, an element-wise multiplication with $\widetilde{\beta}$ evaluation key limbs is performed, necessitating the accumulation of results across the β dimension to yield the updated limbs for the $\widetilde{\beta}$ groups, resulting in each coefficient being read repeatedly $\widetilde{\beta}$ times.

To enhance the efficiency, we have consolidated these multiple independent ModMUL kernels into a unified kernel, and avoid repeated data access through matrix multiplications. As depicted in Algorithm 4, we added data rearrangement before and after the calculation. Through this method, we convert the repeated element-wise multiplications into matrix multiplications with much fewer iterations.

Algorithm 1: Algorithm of origin BConv

Input: $\mathbf{c} = (c_{ibl})_{\alpha \times BS \times N}$
Output: $\widetilde{\mathbf{c}} = (\widetilde{c}_{jbl})_{\alpha' \times BS \times N}$

```
for for i from 0 to α − 1 do
    for for j from 0 to α' − 1 do
        for for b from 0 to BS − 1 do
            c̃_jb += (c_ib × q̂_i^{-1} mod q_i) × q̂_i mod p_j
                                              ▷ Scalar Multiplication
        end for
    end for
end for
return c̃
```

Algorithm 2: Algorithm of new BConv

Input: $\mathbf{c} = (c_{ibl})_{\alpha \times BS \times N}$
Output: $\widetilde{\mathbf{c}} = (\widetilde{c}_{jbl})_{\alpha' \times BS \times N}$

```
c' = (c'_lbi)_{N×BS×α} ⇐ (c_ibl × q̂_i^{-1} mod q_i)_{α×BS×N}
q̂'_{α×α'} = (q̂_ij)        ▷ Scalar Multiplication and Data Reorder
for for l from 0 to N − 1 do
    for for b from 0 to BS do
        c̃'_lb = c'_lb · q̂'                          ▷ Matrix Multiplication
    end for
end for
c̃_{α'×BS×N} = (c̃_ibl)_{α'×BS×N} ⇐ (c̃'_lbi)_{N×BS×α'}
                                                  ▷ Data Reorder
return c̃
```

4.3 Data Preprocessing and Postprocessing

In order to transform the computations in BConv and IP into matrix multiplications, we introduce data preprocessing and postprocessing before and after the matrix multiplication, as outlined in Algorithm 2 and Algorithm 4.

4.3.1 Bconv. The preprocessing of BConv involves data rearrangement and bit-splitting which is implemented to align with the hardware-specific features of TCUs.

In the original BConv kernel, the data within limbs is stored continuously. In the improved algorithm introduced in the previous section, we rearranged the input $\alpha \times BatchSize \times N$ matrix into $N \times BatchSize \times \alpha$ format. The choice of α as the innermost dimension is due to the accumulation along the α dimension in BConv, hence α needs to serve as the K dimension in the matrix multiplication.

Therefore, we change the memory layout of limbs to α continuous, that is, the coefficients of the same term of each limb are sequentially arranged together in α order, as shown in Fig. 6. The rearranged three-dimensional matrix can be viewed as N two-dimensional matrices of size $BatchSize \times \alpha$. This ensures that the rearranged data layout meets the memory access aggregation requirements for matrix operations. When each two-dimensional

Algorithm 3: Algorithm of origin IP

Input: $\mathbf{c} = (c_{jkbl})_{\beta\times\alpha'\times BS\times N}$,
$\mathbf{evk} = (evk_{ijkl})_{\widetilde{\beta}\times\beta\times\alpha'\times N}$
Output: $\widetilde{\mathbf{c}} = (\widetilde{c}_{ikbl})_{\widetilde{\beta}\times\alpha'\times BS\times N}$

for for i from 0 to $\widetilde{\beta}-1$ **do**
  for for j from 0 to $\beta-1$ **do**
    for for k from 0 to $\alpha'-1$ **do**
      for for b from 0 to BS **do**
        $\widetilde{\mathbf{c}}_{ikb} \mathrel{+}= \mathbf{c}_{jkb} * \mathbf{evk}_{ijk}$
        ▷ Element-wise Multiplication
      end for
    end for
  end for
end for
return $\widetilde{\mathbf{c}}$

Algorithm 4: Algorithm of new IP

Input: $\mathbf{c} = (c_{jkbl})_{\beta\times\alpha'\times BS\times N}$,
$\mathbf{evk} = (evk_{ijkl})_{\widetilde{\beta}\times\beta\times\alpha'\times N}$
Output: $\widetilde{\mathbf{c}} = (\widetilde{c}_{ikbl})_{\widetilde{\beta}\times\alpha'\times BS\times N}$

$\mathbf{c}' = (c'_{lkbj})_{N\times\alpha'\times BS\times\beta} \Leftarrow (c_{jkbl})_{\beta\times\alpha'\times BS\times N}$
$\mathbf{evk}' = (evk'_{lkji})_{N\times\alpha'\times\beta\times\widetilde{\beta}} \Leftarrow (evk_{ijkl})_{\widetilde{\beta}\times\beta\times\alpha'\times N}$
▷ Data Reorder

for for l from 0 to $N-1$ **do**
  for for k from 0 to $\alpha'-1$ **do**
    $\mathbf{c}''_{lk} = \mathbf{c}'_{lk} \cdot \mathbf{evk}'_{lk}$ ▷ Matrix Mulitiplication
  end for
end for

$\widetilde{\mathbf{c}} = (\widetilde{c}_{ikbl})_{\widetilde{\beta}\times\alpha'\times BS\times N} \Leftarrow (c''_{lkbi})_{N\times\alpha'\times BS\times\widetilde{\beta}}$ ▷ Data Reorder

return $\widetilde{\mathbf{c}}$

matrix is multiplied by matrix B mentioned in Algorithm 2, the data is all reused.

Upon completion of the matrix multiplication, the postprocessing phase amalgamates the partial products into the definitive outcome and rearranges the data back to the original continuous order within the limb.

4.3.2 IP. The preprocessing of IP, akin to that of BConv, encompasses data rearrangement and bit-splitting.

We rearranged the input data to shift the calculation mode of IP from element-wise multiplication to matrix multiplication. Fig. 8 displays how input data of IP is reordered. Since accumulation occurs in the β dimension, β is set as the K-dimension in matrix multiplication. Limbs are reorganized into $N \times \alpha' \times BatchSize \times \beta$ tensor. Evaluation keys are also rearranged in the corresponding pattern into $N \times \alpha' \times \beta \times \widetilde{\beta}$ tensor. The lower two dimensions of the two tensors can be regarded as matrices, so there are $N \times \alpha'$ matrices of size $BS \times \beta$, and $N \times \alpha'$ matrices of size $\beta \times \widetilde{\beta}$. As illustrated in Fig. 7, the $N \times \alpha$ matrices can be paired to perform matrix multiplication of scale $BS \times \beta \times \widetilde{\beta}$.

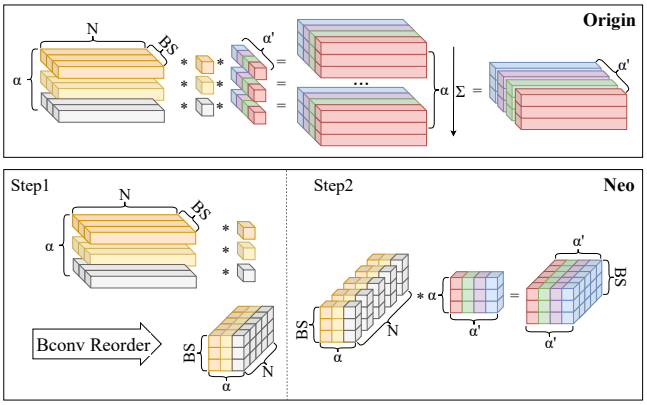


Figure 6: BConv Data Layout and Workflow

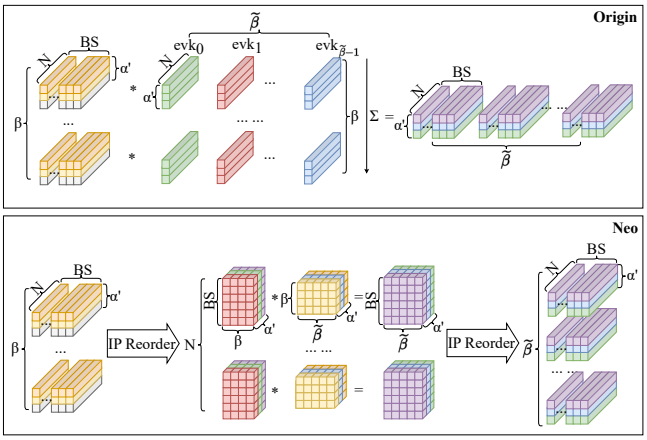


Figure 7: IP Data Layout and Workflow

Through this optimized algorithm and data layout, each matrix only needs to participate in one matrix multiplication operation, and the data inside the matrix is fully reused. Thus, we ensured that each datum requires only a single memory transfer to fulfill all computational requirements.

The postprocessing of IP also encompasses data rearrangement and bit-merging.

4.4 Radix-16 NTT on GPGPU

We have employed a Radix-16 NTT method from SHARP[25] to reduce the complexity of NTT kernel. To accommodate the GPGPU architecture, we have made specific modifications to Radix-16 NTT, substituting the butterfly operations with matrix multiplications. As depicted in Fig. 9, in the original four-step NTT, the limbs are divided into two dimensions, N_1 and N_2, and matrix multiplications are performed with *twiddle factors* of size $N_1 \times N_1$ and $N_2 \times N_2$ on these two dimensions, with respective complexities of $N_1 \times N_2 \times N_1$ and $N_2 \times N_1 \times N_2$. Between the two matrix multiplications, the limbs

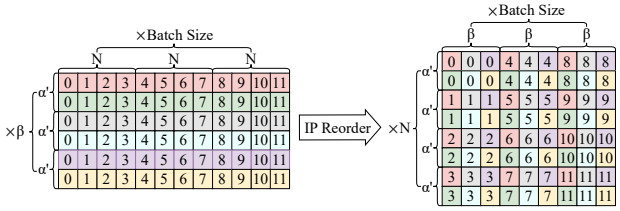


Figure 8: The mapping relationship between input data before and after reordering in the IP kernel. For ease of presentation, we set the parameters of the input data to $N = 4$, $BatchSize = 3$, $\alpha = 2$, $\beta = 3$.

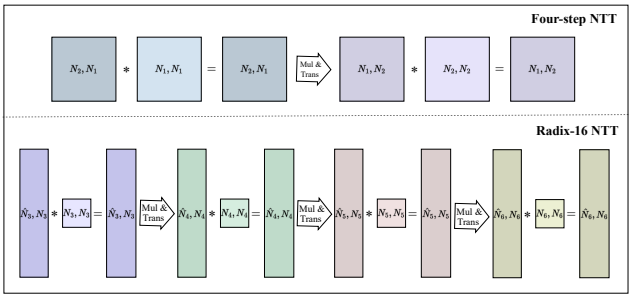


Figure 9: The computational process of Four-step NTT and Radix-16 NTT. "Mul & Trans" represents two steps: scalar multiplication of limbs with $twisting\ factors$ and matrix transposition. $\hat{N}_i$ represents $\frac{N}{N_i}$ in the figure.

are scalar-multiplied by $twisting\ factors$, and then transposed, which inverts the N_1 and N_2 dimensions. Further decomposing the four-step NTT, N_1 is decomposed into N_3 and N_4, and N_2 into N_5 and N_6, transforming the original two matrix multiplications into four smaller-scale matrix multiplications, as depicted in Fig. 9. Without considering batching, when N takes the default value of 2^{16}, N_1 and N_2 are each 2^8, resulting in a matrix multiplication complexity of 2^{24}. After further decomposition in Radix-16 NTT, where N_3, N_4, N_5, and N_6 are all 16, the complexity of the matrix multiplication becomes $(16 \times 16 \times 16) \times 16 \times 16$, which is 2^{20}. Consequently, the total complexity of the matrix multiplications in Radix-16 NTT is 2^{22}, which is $\frac{1}{8}$ of the 2^{25} complexity of four-step NTT. Despite the increased overhead of scalar multiplications with $twisting\ factors$ and transposition, Radix-16 NTT still significantly reduces the overall computational complexity of NTT kernel.

4.5 Kernel Mapping for TCUs and CUDA Cores

As shown in 4, the mapping relationship of all six kernels to computing components is as follows:

- ModADD, ModMUL and AUTO are all mapped onto CUDA Cores.
- In BConv kernel, preprocessing and postprocessing are mapped on CUDA Cores, while matrix multiplication is mapped onto TCU.
- In IP kernel, preprocessing and postprocessing are mapped on CUDA Cores, while matrix multiplication is mapped to either the TCU or CUDA Cores based on the specific parameter configurations.

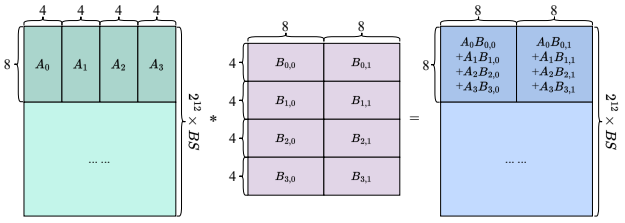


Figure 10: The data mapping of NTT on TCU *FP64* components.

- In NTT kernel, operations of data splitting and merging, element-wise multiplication, matrix transposition, and modular reduction are all mapped to the CUDA Cores, while matrix multiplication is mapped onto TCU.

The following demonstrates how the matrix multiplications in kernels are mapped to specific components of the TCU or CUDA Cores:

4.5.1 NTT. In Radix-16 NTT, the scale of the matrix multiplication is $(BatchSize \times \hat{N}_i) \times N_i \times N_i$, where N_i is the default parameter of 2^{16}, so both dimensions N and K are 16. As depicted in Fig. 10, on the TCU, after splitting, a slice of the limb is divided into four fragments along the K dimension, and the $twiddle\ factors$ are divided into eight fragments, with each fragment being an $8 \times 8 \times 4$ *FP64* type matrix. Fig. 10 labels each input fragment and indicates the calculation pattern for the output matrix.

4.5.2 BConv. Contrary to Radix-16 NTT where the matrix multiplication have dimensions N and K uniformly set to 16, the conversion of BConv and IP into matrix multiplications results in dimensions N and K corresponding to specific parameters. After data layout optimization, the scale of matrix multiplication in BConv is $(BatchSize \times N) \times \alpha' \times \alpha$.

As illustrated in Fig. 11, The bit-sliced limbs need to be transferred to the fragments on the TCU in Step 2 to participate in matrix multiplication. When employing the *INT8* components in TCU for computation, the optimal fragment size that can be selected is $32 \times 8 \times 16$, which introduces a significant amount of padding, and only 25% of the calculation is truly valid. The *FP64* components is significantly better, with no padding, thus 100% of the computations in the matrix multiplication are valid as shown in Fig. 11. Therefore, mapping the matrix multiplication steps in BConv to the *FP64* components can achieve better performance.

4.5.3 IP. After data layout optimization, the scale of matrix multiplication in IP is $(BatchSize \times N) \times \widetilde{\beta} \times \beta$. Similar to BConv, the dimensions N and K of matrix multiplication in IP are also influenced by external parameters. However, the values of β and $\widetilde{\beta}$ are even less aligned with the fragment shapes supported by TCU.

Under the given parameters, α and α' remain constant as l decreases, whereas β and $\widetilde{\beta}$ diminish accordingly. This implies that the valid proportion of matrix multiplications in IP is not a fixed value as it is for NTT and BConv, but rather varies, with the specific variations detailed in Fig. 12. Due to the additional overhead associated with data splitting and merging, experimentally, the performance on TCUs surpasses that of the CUDA Core only when the valid proportion of matrix multiplications exceeds 80%. Consequently, in Neo, the mapping strategy for IP is contingent upon the current

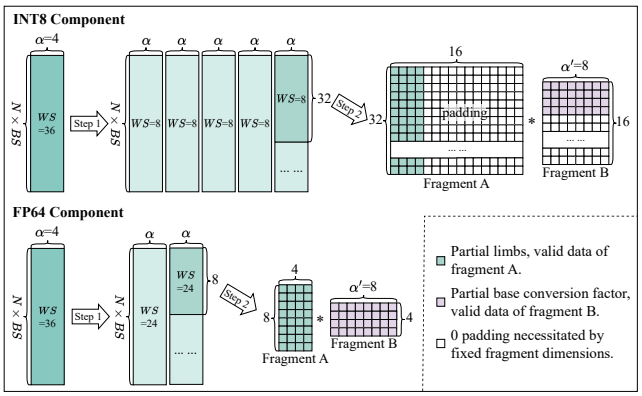


Figure 11: The data mapping of BConv using TCU *INT8* components and TCU *FP64* components under default parameters, with α=4 and α'=8. Step 1 represents bit-splitting the 48-bit input limbs into partial limbs; Step 2 represents selecting a portion from the partial limbs, filling it into a fragment of a given shape on TCU, and then performing matrix multiplication.

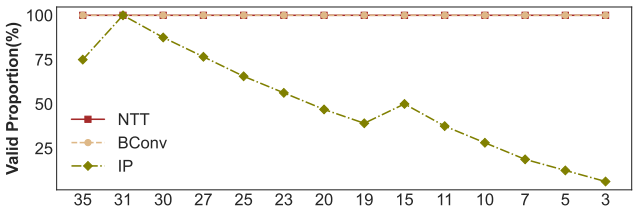


Figure 12: The valid proportion of matrix multiplication under different l(the x-axis in the figure) on *FP64* components in TCU of NTT, BConv and IP.

parameters. When the valid proportion calculated from the parameters exceeds 80%, the matrix multiplication steps of IP are mapped to the *FP64* components in TCUs; otherwise, they are mapped to the CUDA Cores.

When matrix multiplications in IP are mapped onto CUDA Cores, for the reordered limbs and Evaluation keys, a set of adjacent threads, determined by the count of $\widetilde{\beta}$, is employed to perform the multiplication and accumulation of one row (consisting of β elements) from the limbs matrix with a corresponding $\beta \times \widetilde{\beta}$ submatrix from the Evaluation keys. Each thread is tasked with handling a column of the Evaluation keys submatrix, thereby enabling the completion of all computations for a row of data in the limbs matrix by these threads collectively. Since the adjacent threads are located within the same SM, the data they utilize only needs to be transferred to the local memory of that SM once.

4.6 Other Optimization Approaches

In our quest to further optimize kernel performance, we have employed two strategies: Kernel fusion and Multi-stream processing.

Kernel fusion is a technique where multiple related kernel functions are consolidated into a single kernel, thereby reducing the memory access overhead for intermediate results and the overhead associated with kernel launches. Specifically, for all kernels that leverage TCU for accelerated matrix multiplication, we have integrated the steps of splitting & reordering, matrix multiplication, and merging & reordering into a single kernel, utilizing shared memory to minimize access to global memory.

Multi-stream processing allows for the parallel handling of different streams, enabling more efficient utilization of the GPGPU's different components. For kernels with substantial workloads, such as NTT, we have adopted multi-stream parallelism to partition the entire kernel into multiple streams across batches. Within a single NTT kernel, some computational steps exclusively utilize TCUs, while others solely employ CUDA Cores. When TCU operations within a stream are suspended due to data dependencies or memory latency, computations in another stream using CUDA Cores can immediately fill the idle cycles. This dynamic scheduling reduces core idle time, thereby enhancing overall utilization.

Table 3: System Configuration

Type	Model
CPU	Hygon C86 7285 32-core Processor
GPGPU	NVIDIA Amphere A100 GPGPU-40GB
Memory	512GB

Table 4: CKKS parameter sets.

	A	B	C	D‡	E	F	G	H
N	2^{16}	2^{16}	2^{16}	2^{16}	2^{16}	2^{16}	2^{16}	2^{16}
L	35	35	35	35	35	23	23	44
$WordSize$	36	36	36	60	60	36	36	60
$WordSize_T^{\dagger}$	-	-	48	64	-	-	48	-
d_{num}	1	3	9	36	36	1	6	45
$\widetilde{\alpha}^{\dagger}$	-	-	5	3	-	-	5	-
λ	≥128	≥128	≥128	≥128	≥128	≥128	≥128	≥98
$BatchSize$	128	128	128	128	-	128	128	-

† Parameter $WordSize_T$ and $\widetilde{\alpha}$ are only used when the KLSS method is applied.
‡ Set-D is configured with parameters consistent with those used in HEonGPU, facilitating a performance comparison between the two under identical parameter settings.

5 Methodology

We implement Neo based on GCC 8.4, CUDA 11.3, PyTorch 1.12, and Cupy 11.5. To assess the practical performance of Neo, we conduct an evaluation on a high-end server equipped with one NVIDIA A100 GPGPU. The detailed configurations of our experimental system are shown in Table 3.

Based on the sets of parameters displayed in Table 4, we undertook performance measurements of Neo across three distinct FHE applicaitons. By default, the Bootstrapping in the testing of the applications incorporates the DS.

(1) **PackBootstrap**: Bootstrapping[34] refreshes a ciphertext's multiplicative budget, mitigating the accumulated noise. The PackBootstrap process is analyzed by normalizing its runtime for ℓ_{eff}. The ciphertext with N = 65536 restores its security level to L=57. We use DS technology in PackBootstrap to meet the precision requirement.
(2) **HELR**: HELR[17, 18] constitutes a machine learning workload focused on training a binary classification model via logistic regression. It employs a 196-element weight to train using 14x14 MNIST images for distinguishing between the

Table 5: Application performance comparison among TensorFHE, HEonGPU and Neo. Execution times are measured in seconds. SS: the implementation integrated with single scaling (SS) [25].

Scheme	Param.	Pack-Bootstrap	HELR	ResNet-20	ResNet-32	ResNet-56
CPU*	-	17.2	356	1380	-	-
TensorFHE$^{\dagger}_{SS}$	Set-F	0.53	0.90	35.27	57.70	102.71
Neo$^{\dagger}_{SS}$	Set-G	0.17	0.19	9.11	14.90	26.48
TensorFHE‡	Set-A	0.67	0.96	41.07	67.18	119.49
TensorFHE‡	Set-B	0.74	0.78	38.77	64.22	114.15
TensorFHE‡	Set-C	0.85	0.73	40.68	66.19	117.30
HEonGPU	Set-E	0.36	0.26	16.42	27.00	47.99
Neo	Set-C	0.24	0.22	12.03	19.68	34.98
Neo	Set-D	0.27	0.25	13.39	21.83	38.78

* The application performance data of the CPU was obtained from Craterlake[40].
† These results are evaluated under L=23, which is a preferred parameter for the implementation with SS [26].
‡ We reimplemented TensorFHE with DS integrated since the absence of DS in TensorFHE leads to precision loss[25].

Table 6: Performance comparison of operations with the prior work. Execution times are measured in microseconds (μs), apart from CPU performance. Note that all operations are evaluated under the condition of $l = 35$, except for the CPU data, which originates from 100x[22].

Scheme	Param.	HMult	HRotate	PMult	HAdd	PAdd	Rescale
CPU [22]	Set-H	2.6s	2.6s	26.2ms	28.2ms	28.2ms	45.8ms
TensorFHE	Set-A	15304.6	15256.2	82.3	47.0	47.2	115.1
TensorFHE	Set-B	18689.4	18592.1	82.3	47.0	47.2	115.1
TensorFHE	Set-C	32523.6	32498.9	82.3	47.0	47.2	115.1
HEonGPU	Set-E	8172.6	8200.0	92.7	62.4	48.6	150.5
Neo	Set-C	3472.5	3422.1	81.7	46.1	46.4	114.3

Table 7: Performance comparison of kernels with the prior work under Param Set-B.

	#BConv/second	*#IP/second*	*#NTT/second*
TensorFHE	311526	621762	25478
Neo	854700	1617978	95329
Speedup	2.74×	2.60×	3.74×

digits 3 and 8. Experiments are conducted with batch sizes of 1,024 images. The model undergoes 32 iterations of training, while we show the time for one iteration in Section 6

(3) **ResNet-20**: For the Convolutional Neural Network (CNN) inference, the FHE CKKS implementation of the ResNet-20, ResNet-36 and ResNet-56 model in [32] is utilized on a 32x32x3 CIFAR-10 [31] image.

6 Results

We demonstrate the performance improvements of Neo by comparing it with the significant prior work, TensorFHE and HEonGPU[49]. We present performance comparisons across various applications and provide detailed performance breakdowns at the level of operations. Besides, we exhibit sensitivity studies for different parameters. Due to the adoption of batch processing for ciphertexts in Neo, all statistical timings represent the average time per batch.

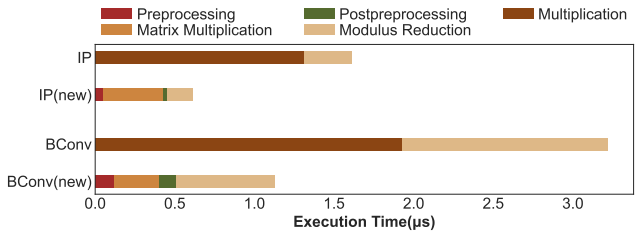


Figure 13: The comparison of the execution time breakdown for each optimized kernel step (BConv and IP) versus the total pre-optimization time, with statistical durations normalized to a single operation.

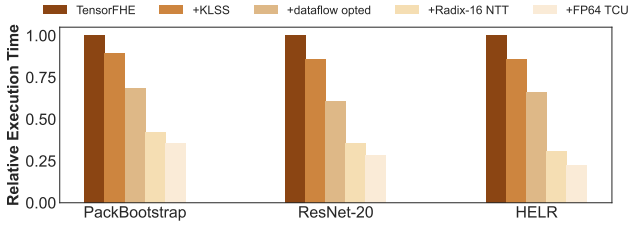


Figure 14: How relative execution time changes when incrementally applying four optimization steps: KLSS method(+KLSS), dataflow optimization of BConv and IP(+dataflow opted), ten-step NTT(+ten-step NTT), utilizing *FP64* components in TCU for matrix multiplication(+*FP64* TCU). For each application, the execution time is normalized to TensorFHE.

6.1 Performance

In this section, we present the performance of Neo at both the application level and the operation level, in comparison with TensorFHE.

6.1.1 Application Performance. As shown in Table 5, Neo achieved significant performance optimization compared to TensorFHE and CPU on all three applications. On average, Neo demonstrates a 3.41× speed improvement over TensorFHE across all applications under identical parameters, and a 3.28× speedup compared to the optimal parameter configuration of TensorFHE. Compared to relatively newer non-TCU work, HEonGPU, Neo also demonstrates an average 19.9% performance advantage across all applications. This proves that our acceleration strategies have achieved good results in real workload.

6.1.2 Operation Performance. The improvements integrated into Neo are specifically designed to optimize the `KeySwitch` operation, leading to a significant reduction in computational overhead for the most demanding operations, namely `HMULT` and `HROTATE`, which incorporate `KeySwitch`. This reduction is evident in Table 6.

6.1.3 Kernel Performance. As shown in Table 7, three crucial kernels, BConv, IP, and NTT, have all demonstrated significant performance improvements compared to TensorFHE, with the most time-consuming kernel, NTT, showing the highest improvement ratio, reaching 3.74 times that of TensorFHE. As illustrated in Fig. 13, the optimized BConv and IP kernels demonstrate a substantial reduction in total execution time compared to their pre-optimized

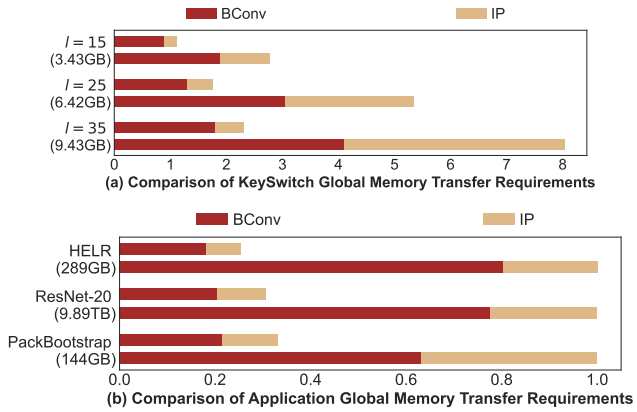


Figure 15: In figures (a) and (b), each cluster consists of two bars: the upper bar represents the data transfer requirement of the kernels after algorithm and data layout optimization, while the lower bar indicates the data transfer requirement of the original kernels.

versions, although introducing additional overhead from preprocessing and postprocessing stages - both constituting negligible proportions of the computational workflow. As the three kernels account for the majority of the cost in KeySwitch operations, their substantial optimizations also support the optimized performance of HMULT and HROTATE as indicated in Table 6.

6.2 Effectiveness of Optimization

Fig. 14 delineates the performance enhancements accrued incrementally from four optimization steps.

The instantiation of KLSS method on GPGPU, the first step, effectuates a enhancement in performance, attributable to its direct reduction in the total computational complexity of KeySwitch. The second step, transforming BConv and IP into matrix multiplication form greatly enhances data reuse so to diminish the frequency of kernel accesses to global memory, thereby further improving performance. As depicted in Fig. 15, this optimization results in a significant reduction in data transfer requirement associated with BConv and IP, consequently leading to performance improvements of applications in this step.

The third step involves the adoption of the ten-step NTT, which significantly reduces the computational load of NTT kernel. Since NTT constitutes the most time-consuming kernel in KeySwitch, the performance improvement achieved by employing ten-step NTT is markedly noticeable. Ultimately, by assigning the matrix multiplications in BConv, IP, as well as those in the NTT, to the *FP64* components of TCUs, we obtained superior performance compared to calculations performed on CUDA Cores or *INT8* components of TCUs.

6.3 Sensitivity study

In the KLSS method, both parameters, d_{num} and $\widetilde{\alpha}$, influence the performance of the application, specifically affecting the complexity of the KeySwitch operation. We maintained other parameters consistent with Set-B and, while fixing one parameter, adjusted the other to record the KeySwitch timing, as presented in Table 8. The default values of $d_{num} = 9$ and $\widetilde{\alpha} = 5$ yield the optimal performance for KeySwitch.

Table 8: Execution time of KeySwitch under different d_{num} and $\widetilde{\alpha}$. The optimal parameter set, with default values of $d_{num} = 9$ and $\widetilde{\alpha} = 5$, is highlighted in the table.

	$d_{num} = 4$	$d_{num} = 6$	$d_{num} = 9$	$d_{num} = 12$	$d_{num} = 18$
$\widetilde{\alpha} = 4$	5.34	4.30	3.81	3.84	4.00
$\widetilde{\alpha} = 5$	4.50	4.11	**3.22**	3.82	4.12
$\widetilde{\alpha} = 6$	4.53	3.67	3.39	3.51	4. 37
$\widetilde{\alpha} = 7$	4.39	3.30	3.51	3.61	4.03
$\widetilde{\alpha} = 8$	3.95	3.69	3.38	3.65	4.13
$\widetilde{\alpha} = 9$	3.57	3.55	3.48	3.99	4.61
$\widetilde{\alpha} = 10$	3.93	3.79	3.24	3.59	4.61

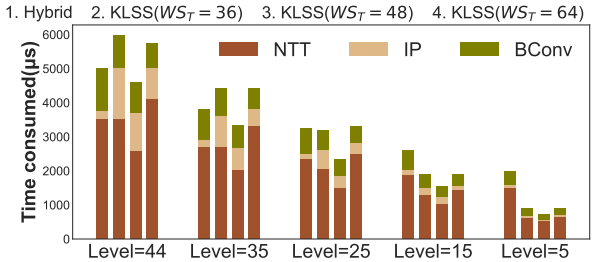


Figure 16: Execution time of KeySwitch operation using Hybrid method and KLSS method with three different $WordSize_T$. The four labels above the figure correspond to the four bars in each cluster, respectively.

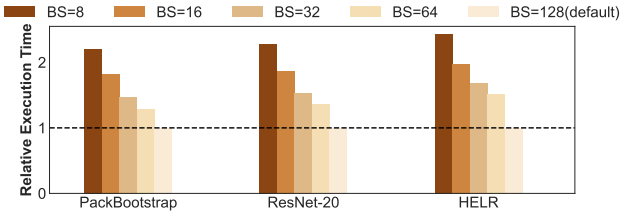


Figure 17: Relative execution time of three applications under five different *BatchSize* param(BS in the legend). *BatchSize*=128 is the default param, and the execution time of others is normalized to it.

As discussed in Section 3, the selection of $WordSize_T$ represents a trade-off between algorithmic complexity and hardware implementation complexity. We selected three $WordSize_T$: 36, 48, and 64, comparing the execution times of the KLSS method versus the Hybrid method, while keeping all other parameters consistent with Set-B. Fig. 16 shows that $WordSize_T$ of 48 exhibits the optimal performance. This is attributed to the fact that at the $WordSize_T$ of 36, the parameter α' is excessively large, resulting in a higher overall algorithmic complexity for the KLSS method. On the other hand, at the $WordSize_T$ of 64, although the algorithmic complexity is reduced, the matrix multiplications in NTT are hampered by excessive Booth complexity due to hardware constraints of the TCU, leading to increased overhead. Therefore, the default $WordSize_T$ of 48 is proven to be the optimal choice.

BatchSize refers to the number of ciphertexts processed simultaneously in one operation. An expanded *BatchSize* enables a single operation to leverage the resources of the GPGPU more fully, thereby increasing available data and resource throughput. As depicted in Fig. 17, the influence of varying *BatchSize* is notably significant. We experimented with 5 different values ranging from 8 to 128, while keeping all other parameters consistent with Set-B. The higher the *BatchSize*, the higher the parallelism, so the execution time per batch will decrease monotonically as the *BatchSize* increases. However, due to the limitations of GPGPU memory capacity, *BatchSize* cannot be increased indefinitely; hence, we ultimately choose 128 as the default parameter.

7 Related Work

Acceleration solution with `KeySwitch` optimizations: Due to the importance and time-consuming nature of `KeySwitch` operation in FHE applications, our work is dedicated to improving its performance. Many previous works have taken note of this and contributed to `KeySwitch` optimization.[10, 15, 20, 29, 40, 48] FAB[1] performed the first fully packed Bootstrapping on FPGA for a practical parameter set, with restructuring the data flow of IP kernel and employing scheduling strategies. Taiyi[11] conducted an in-depth optimization and innovation of the `KeySwitch` operation, incorporating dedicated hardware structures to address the IP kernel. In comparison, FAB still used the original Hybrid `KeySwitch` method, while Neo has adopted an improved KLSS method; Taiyi designed a specialized accelerator for the KLSS method, concentrating on optimizations at the architectural level. Neo, however, implemented KLSS on actual GPGPU, complementing this with a suite of optimizations tailored to the specific hardware platform. Moreover, Neo's enhancements extend beyond hardware considerations to include algorithmic optimization.

GPGPU-based solutions: Wang et al.[44] proposed the first GPGPU acceleration solution, based on [14]. Subsequently, several works[2, 3, 9] used GPGPU to accelerate BGV[6] and its variants BENZ[4] and HPS[16]. In the case of the CKKS scheme[8], an accelerating solution[23] achieved a preliminary implementation of NTT and INTT on GPGPU. 100x[22] is the first CKKS implementation on GPGPU, supporting Residue Number System (RNS) [7, 19] with large parameters. HE-Booster[45] proposed a scalable multi-GPGPU parallelization design, which utilizes data-level parallelism through fine-grained data partitioning strategies. However, TCUs in GPGPU were not leveraged. TensorFHE[12] attempts to enhance the performance of the NTT matrix multiplication, by splitting 32-bit data into *INT8* data and utilizing the integer components in TCUs. Despite this, it is constrained by the limitations of VRAM capacity and the overhead associated with Booth complexity casued by *INT8* type. Moreover, all the other kernels did not benefit from the utilization of TCUs. Neo harnesses the *FP64* components in the TCU, further enhancing the speed of matrix multiplication. Additionally, Neo implements algorithmic transformations and data layout optimizations, enabling a broader range of kernels to be accelerated by the TCUs.

Solutions based on FPGA and ASIC: Several previous works have accelerated the BFV and BGV schemes using FPGAs [38, 39]. For CKKS, HEAX [37], coxhe [20], Poseidon [47] and FAB [1] are notable works that utilize FPGA to accelerate the CKKS scheme. In terms of ASIC-based solutions, A multitude of enhancements to hardware architecture have been concurrently introduced in step with the advancement of FHE [11, 25, 27, 30, 40], which optimized the various operations of FHE at the hardware level and possessed excellent theoretical performance. Nonetheless, The software programming for FPGA is relatively complex and time-consuming, while ASIC's hardware circuitry cannot be modified after tape-out, resulting in limited generality. In contrast to the limited modifiability of FPGA and ASIC, GPGPU excels in its significant flexibility stemming from distinguished software programming support. Furthermore, given its broader range of applicability and much lower cost, we contend that GPGPU-based solutions are more practical and aligned with real-world implementation needs.

8 Conclusion

This paper presents , an acceleration solution for FHE applications on GPGPU. Faced with the increase in hardware overhead due to the expansion of *WordSize* to 36, we adopted the KLSS method for the `KeySwitch` operation and found an optimal point between algorithmic complexity and hardware implementation complexity, achieving performance superior to the Hybrid method. Delving deeper, we concentrate on the optimization of basic kernels, introducing a series of enhancements in . Based on memory analysis of kernels such as BConv and IP, we identified inefficiencies due to poor data reuse, and we proposed improvements from both algorithm and data layout perspectives, transforming repeated element-wise multiplication computations into matrix multiplication computations, which significantly enhances data reuse. In the case of the utilization of GPGPU, we observed that previous works did not fully exploit the TCUs. offloads the matrix multiplication operations of BConv onto the TCUs, thereby broadening the application range of TCUs in FHE. Furthermore, marks the premiere engagement of the floating-point components within TCUs, which deliver a decrease in algorithmic intricacy and a reduction in additional overhead relative to the previously employed integer components. Through these refinements, we have obtained performance that surpasses that of the existing fastest GPGPU-accelerated FHE solution. Considering the versatility and cost-effectiveness of GPGPUs, we are confident that Neo will further propel FHE towards practical application and set a strong stage for future research endeavors.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Natural Science Foundation of China for Distinguished Young Scholars under Grant No. 62125208, the National Key R&D Program of China under Grant No. 2023YFB4503200, and the Strategic Priority Research Program of Chinese Academy of Sciences under Grant No. XDB0690100.

References

[1] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 882–895.

[2] Ahmad Al Badawi, Yuriy Polyakov, Khin Mi Mi Aung, Bharadwaj Veeravalli, and Kurt Rohloff. 2019. Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. *IEEE Transactions on Emerging Topics in Computing* 9, 2 (2019), 941–956.

[3] Ahmad Al Badawi, Bharadwaj Veeravalli, Chan Fook Mun, and Khin Mi Mi Aung. 2018. High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 70–95.

[4] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. 2016. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*. Springer, 423–442.

[5] Toru Baji. 2018. Evolution of the GPU Device widely used in AI and Massive Parallel Processing. In *2018 IEEE 2nd Electron devices technology and manufacturing conference (EDTM)*. IEEE, 7–9.

[6] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual cryptology conference*. Springer, 868–886.

[7] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2019. 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, 347–368.

[8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. 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, 409–437.

[9] Jiyang Dong. 2016. Accelerating BGV scheme of fully homomorphic encryption using GPUs. *Ph. D. dissertation* (2016).

[10] Phap Ngoc Duong and Hanho Lee. 2023. Pipelined key switching accelerator architecture for CKKS-based fully homomorphic encryption. *Sensors* 23, 10 (2023), 4594.

[11] Shengyu Fan, Xianglong Deng, Zhuoyu Tian, Zhicheng Hu, Liang Chang, Rui Hou, Dan Meng, and Mingzhe Zhang. 2024. Taiyi: A high-performance CKKS accelerator for Practical Fully Homomorphic Encryption. *arXiv preprint arXiv:2403.10188* (2024).

[12] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. 2023. Tensorfhe: Achieving practical computation on encrypted data using gpgpu. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–934.

[13] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[14] Craig Gentry and Shai Halevi. 2011. Implementing gentry's fully-homomorphic encryption scheme. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 129–148.

[15] Jia-Zheng Goey, Wai-Kong Lee, Bok-Min Goi, and Wun-She Yap. 2021. Accelerating number theoretic transform in GPU platform for fully homomorphic encryption. *The Journal of Supercomputing* 77 (2021), 1455–1474.

[16] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. 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, 83–105.

[17] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2018. Efficient logistic regression on large encrypted data. *Cryptology ePrint Archive* (2018).

[18] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic regression on homomorphic encrypted data at scale. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 9466–9471.

[19] Kyoohyung Han and Dohyeong Ki. 2020. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*. Springer, 364–390.

[20] Mingqin Han, Yilan Zhu, Qian Lou, Zimeng Zhou, Shanqing Guo, and Lei Ju. 2022. coxHE: A software-hardware co-design framework for FPGA acceleration of homomorphic computation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1353–1358.

[21] Junaid Hassan, Danish Shehzad, Usman Habib, Muhammad Umar Aftab, Muhammad Ahmad, Ramil Kuleev, and Manuel Mazzara. 2022. [Retracted] The Rise of Cloud Computing: Data Protection, Privacy, and Open Research Challenges—A Systematic Literature Review (SLR). *Computational intelligence and neuroscience* 2022, 1 (2022), 8303504.

[22] Wonkyung Jung, Sangpyo Kim, Jung Ho Ahn, Jung Hee Cheon, and Younho Lee. 2021. Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with GPUs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), 114–148.

[23] Wonkyung Jung, Eojin Lee, Sangpyo Kim, Jongmin Kim, Namhoon Kim, Keewoo Lee, Chohong Min, Jung Hee Cheon, and Jung Ho Ahn. 2021. Accelerating fully homomorphic encryption through architecture-centric analysis and optimization. *IEEE Access* 9 (2021), 98772–98789.

[24] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. 2022. Approximate homomorphic encryption with reduced approximation error. In *Cryptographers' Track at the RSA Conference*. Springer, 120–144.

[25] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–15.

[26] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. ARK: Fully Homomorphic Encryption Accelerator with Runtime Data Generation and Inter-Operation Key Reuse. *arXiv preprint arXiv:2205.00922* (2022).

[27] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. 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, 1237–1254.

[28] Miran Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. 2023. Accelerating HE Operations from Key Decomposition Technique. In *CRYPTO*. 70–92.

[29] Sangpyo Kim, Wonkyung Jung, Jaiyoung Park, and Jung Ho Ahn. 2020. Accelerating number theoretic transformations for bootstrappable homomorphic encryption on gpus. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 264–275.

[30] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. Bts: An accelerator for bootstrappable fully homomorphic encryption. In *Proceedings of the 49th annual international symposium on computer architecture*. 711–725.

[31] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *International Conference on Machine Learning*. PMLR, 12403–12422.

[32] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. 2022. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *iEEE Access* 10 (2022), 30039–30054.

[33] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 72–86.

[34] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. 2020. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. 64–70.

[35] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. Technical Report. NVIDIA. https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[36] Samanvaya Panda. 2021. Principal component analysis using CKKS homomorphic scheme. In *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5*. Springer, 52–70.

[37] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. 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*. 1295–1309.

[38] Sujoy Sinha Roy, Kimmo Järvinen, Jo Vliegen, Frederik Vercauteren, and Ingrid Verbauwhede. 2018. HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation. *IEEE Trans. Comput.* 67, 11 (2018), 1637–1650.

[39] Sujoy Sinha Roy, Furkan Turan, Kimmo Jarvinen, Frederik Vercauteren, and Ingrid Verbauwhede. 2019. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 387–398.

[40] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 173–187.

[41] Amanpreet Kaur Sandhu. 2021. Big data with cloud computing: Discussions and challenges. *Big Data Mining and Analytics* 5, 1 (2021), 32–40.

[42] Chao Su and Qingkai Zeng. 2021. Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures. *Security and Communication Networks* 2021, 1 (2021), 5559552.

[43] Yunxuan Su, Xu An Wang, Weidong Du, Yu Ge, Kaiyang Zhao, and Ming Lv. 2023. A secure data fitting scheme based on CKKS homomorphic encryption for medical IoT. *Journal of High Speed Networks* 29, 1 (2023), 41–56.

[44] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2012. Accelerating fully homomorphic encryption using GPU. In *2012 IEEE conference on high performance extreme computing*. IEEE, 1–5.

[45] Zhiwei Wang, Peinan Li, Rui Hou, Zhihao Li, Jiangfeng Cao, XiaoFeng Wang, and Dan Meng. 2023. HE-Booster: an efficient polynomial arithmetic acceleration on GPUs for fully homomorphic encryption. *IEEE Transactions on Parallel and Distributed Systems* 34, 4 (2023), 1067–1081.

[46] Chaowei Yang, Qunying Huang, Zhenlong Li, Kai Liu, and Fei Hu. 2017. Big Data and cloud computing: innovation opportunities and challenges. *International Journal of Digital Earth* 10, 1 (2017), 13–53.

[47] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical homomorphic encryption accelerator. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 870–881.

[48] Yujia Zhai, Mohannad Ibrahim, Yiqin Qiu, Fabian Boemer, Zizhong Chen, Alexey Titov, and Alexander Lyashevsky. 2022. Accelerating encrypted computing on intel gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 705–716.

[49] Ali Şah Özcan and Erkay Savaş. 2024. HEonGPU: a GPU-based Fully Homomorphic Encryption Library 1.0. Cryptology ePrint Archive, Paper 2024/1543. https://eprint.iacr.org/2024/1543