HI-CKKS: Is High-Throughput Neglected? Reimagining CKKS Efficiency with Parallelism

Fuyuan Chen, Jiankuo Dong, Xiaoyu Hu, Zhenjiang Dong, Wangchen Dai, Jingqiang Lin, Fu Xiao

***Abstract*—The proliferation of data outsourcing and cloud services has heightened privacy vulnerabilities. CKKS, among the most prominent homomorphic encryption schemes, allows computations on encrypted data, serving as a critical privacy safeguard. However, performance remains a central bottleneck, hindering widespread adoption. Existing optimization efforts often prioritize latency reduction over throughput performance. This paper presents HI-CKKS, a throughput-oriented High-performance Implementation of CKKS homomorphic encryption, addressing these challenges. Our HI-CKKS introduces a batch-supporting asynchronous execution scheme, effectively mitigating frequent data interactions and high waiting delays between hosts and servers in service-oriented scenarios. We analyze the fundamental (I)NTT primitive, which is critical in CKKS, and develop a hierarchical, hybrid high-throughput implementation. This includes efficient arithmetic module instruction set implementations, unified kernel fusion, and hybrid memory optimization strategies that significantly improve memory access efficiency and the performance of (I)NTT operations. Additionally, we propose a multi-dimensional parallel homomorphic multiplication scheme aimed at maximizing throughput and enhancing the performance of (I)NTT and homomorphic multiplication. In conclusion, our implementation is deployed on the RTX 4090, where we conduct a thorough throughput performance evaluation of HI-CKKS, enabling us to pinpoint the most effective parallel parameter settings. Compared to the CPU implementation, our system achieves throughput increases of $175.08\times$, $191.27\times$, and $679.57\times$ for NTT, INTT, and HMult, respectively. And our throughput performance still demonstrates a significant improvement, ranging from $1.54\times$ to $693.17\times$ compared to the latest GPU-based works.**

***Index Terms*—CKKS, Homomorphic Multiplication, Number Theoretic Transform (NTT), Parallel Processing, GPU.**

I. Introduction

As information technology advances and digital transformation accelerates, the collection, processing, and sharing of data have become increasingly frequent. This not only brings enhancements in convenience and efficiency but also raises deep concerns about data privacy protection. In service-oriented scenarios, enterprises and service providers often need to handle a large amount of user data, including identity information, contact details, and location data [1]. This requires the implementation of effective privacy protection measures to ensure the security of user information. Additionally, service-oriented scenarios typically have high demands for concurrent performance [2], which places higher performance requirements on privacy protection technologies. In this context, privacy protection measures with high concurrency capabilities are necessary.

Traditional solutions only ensure that data remains encrypted during storage and transmission, and must be decrypted before processing, posing a risk of privacy leakage. Fully Homomorphic encryption (FHE), often hailed as the "Holy Grail" of cryptography [3], offers a novel approach to securing privacy: it allows application services to operate directly on encrypted data, enabling computations on encrypted messages without the need for decryption. Under this paradigm, even if attackers successfully breach the defense perimeter and infiltrate the system, they cannot access the original critical data, fundamentally reducing the risk of information leakage. Figure 1 shows the application of a FHE scheme in service-oriented scenarios. Compared to other privacy protection technologies such as federated learning [4], secure multi-party computation [5], and differential privacy [6], privacy protection schemes based on FHE can provide comprehensive security and privacy protection from the data collection stage throughout the entire data lifecycle, demonstrating broader application prospects. However, the performance of data ciphertext computations is 10^5 to 10^7 times lower than that of plaintext computations [7], making the huge performance overhead the main bottleneck restricting the further development and application of homomorphic encryption technology.

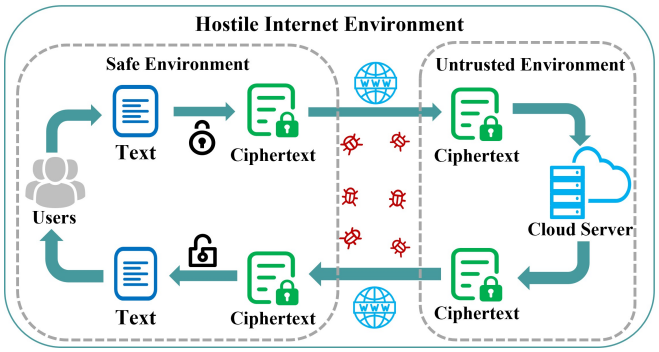


Fig. 1: FHE in Service-oriented Scenarios

In recent years, the rapid advancement of GPUs has introduced new solutions to address the performance challenges of FHE. In 2006, NVIDIA introduced the Single-Instruction Multiple-Thread (SIMT) architecture in GPUs [8], featuring thread-level parallelism where multiple independent threads can execute the same instruction concurrently. This charac-

Fuyuan Chen, Jiankuo Dong, Xiaoyu Hu, Zhenjiang Dong and Fu Xiao are with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China. Wangchen Dai is with Zhejiang Lab, Hangzhou, China. Jingqiang Lin is with the School of Cyber Security, University of Science and Technology of China, Hefei, China.

teristic provides GPUs with exceptional parallel processing ability and high throughput, thereby significantly accelerating intensive cryptographic computations and optimizing cryptographic performance. Research [9] indicates that asymmetric cryptographic algorithms implemented on GPUs can achieve performance improvements multiple times, even up to tenfold greater than contemporary CPUs, ASICs, and FPGAs. Thus, leveraging GPU characteristics to optimize the throughput performance of the FHE algorithm has tremendous potential.

A. *Related Works*

Since the concept of FHE was first proposed [10] and implemented [11], several representative computational schemes have emerged, including BGV [12], BFV [13], and CKKS [14]. The CKKS algorithm, in particular, has attracted attention for its ability to handle floating-point operations in encrypted data. However, its practical application is limited due to its high complexity, extensive computational resource and storage requirements, and the accumulation of noise. To address these challenges, CPU-based open-source libraries such as SEAL [15] and HElib [16] have been developed to facilitate the engineering application of CKKS. Despite these libraries performing well in terms of compatibility and usability, their computational performance still falls short in high concurrency service scenarios.

The development of GPUs and their extensive application in the field of cryptography have opened new opportunities for performance optimization of the CKKS algorithm. In 2016, Dai et al. [17] developed a homomorphic encryption algorithm library called cuHE, based on polynomial rings, which fully utilizes the extensive parallelism and high memory bandwidth provided by CUDA GPUs. They employed methods based on the Number Theoretic Transform (NTT) and the Chinese Remainder Theorem (CRT) to construct arithmetic functions capable of handling very large polynomial operands, thereby achieving significant performance improvements. In 2021, Jung et al. [18] identified that the primary performance bottleneck for the CKKS algorithm on GPUs was the high demand for main memory bandwidth. They introduced a memory-centric optimization strategy for the CKKS bootstrapping process, achieving more than $100\times$ improvement in performance. This significant advancement has established a crucial foundation for subsequent research in the area.

Shen et al. [19] proposed the CARM to implement CUDA-accelerated homomorphic multiplication in heterogeneous IoT systems, marking the first GPU-optimized implementation to cover homomorphic multiplication operations for BGV, BFV, and CKKS schemes. Fan et al. [20] proposed TensorFHE, a GPGPU-based FHE acceleration scheme, utilizing Tensor Cores Units (TCU) to enhance NTT operations. Fan et al. [21] implemented a set of low and intermediate-level FHE primitives based on three data precisions (INT32, INT64, and FP64) to accelerate CKKS implementations on GPU platforms. Yang et al. [22] proposed Phantom is the first scheme to integrate BGV, BFV, and CKKS on a GPU platform.

However, these studies mainly focus on how to reduce the computational latency of the CKKS algorithm, particularly the homomorphic multiplication operations, often using more GPU resources during computation to increase processing speed. This results in prohibitively high costs for implementing FHE-based privacy protection schemes in service scenarios with significant concurrent computational demands.

B. *Our Contributions*

Service-oriented scenarios often involve the need for real-time data processing at a large scale while ensuring data privacy. This necessitates FHE algorithms with high concurrency capabilities. FHE entail computationally intensive tasks, involving numerous identical computational operations on data. Leveraging the unique SIMT architecture of GPUs can accelerate these computations. Current optimization based on GPU efforts for FHE primarily concentrate on reducing overall computation latency. However, there remains significant potential for improvement in computational throughput capability.

In this paper, we propose the HI-CKKS scheme, which is optimized for high concurrency demand for FHE algorithms in service-oriented scenarios on GPU hardware platform, aimed at achieving optimal throughput performance. The main contributions are as follows:

- **Throughput-Oriented Architecture.** To address the high volume of concurrent data requests in service-oriented scenarios and reduce the latency and security risks associated with data interactions, HI-CKKS designs a throughput-oriented asynchronous computation framework. This framework supports batch processing, leveraging the CPU to handle precomputed parameters and integrate computational tasks, while the GPU launches computation kernels to execute high-throughput calculations. The GPU can concurrently process multiple homomorphic computation requests, significantly reducing data transfer and waiting latency between the host and server.
- **Throughput-Oriented Optimization Strategies.** HI-CKKS proposes a throughput-oriented performance optimization scheme that balances memory access efficiency and parallelism to achieve optimal throughput performance. We analyzed the performance bottlenecks of NTT and homomorphic multiplication on the GPU and designed a hierarchical and hybrid NTT implementation along with multi-dimensional parallel optimizations for homomorphic multiplication. These optimizations include unified kernel fusion, hybrid memory access strategies, and constant-time arithmetic instruction set implementations. By reducing the latency caused by data access and fully utilizing the limited hardware resources of the GPU, we significantly improved the throughput performance of the computations.
- **Throughput-Oriented Peak Testing.** HI-CKKS conducts comprehensive performance testing with a focus on high throughput, adjusting computational parameters to identify the throughput peak and ultimately determining the optimal parallel parameters. We compare our results with those from CPU platforms and recent related works. Compared to CPU implementations, the throughput of our NTT, INTT,

and homomorphic multiplication increased by $175.08\times$, $191.27\times$, and $679.57\times$, respectively. Compared to the GPU implementation, the improvements were $26.91\times$, $32.92\times$, and $693.16\times$, respectively. And compared to current related work, the increase in throughput ranged from $1.54\times$ to $55.8\times$.

The rest of our paper is organized as follows. Section II explains some preliminaries including CKKS, NTT and GPU. Section III describes the proposed strategies for the HI-CKKS implementation. Section IV performs our optimized implementation and compares it with related works. Section V concludes the paper.

II. Preliminaries

In this section, we briefly introduce the fundamental concepts and operational functions of the CKKS scheme. Following this, the key modules NTT in the scheme is explored. Finally, we provide an introduction to GPU and its memory architecture.

A. Notation

The security of CKKS is based on the hardness of the Ring Learning with Errors (R-LWE) problem. Let $\mathbb{Z}$ and $\mathbb{C}$ be sets of integers and complex numbers, where $\mathbb{Z}_q$ represents the finite ring $\{0, 1, ..., q-1\}$, in which the arithmetic is performed modulo q. Then, the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\{x^N+1\}$ consists of degree $N-1$ polynomials with integer coefficients in $\mathbb{Z}_q$, where N is a power of 2 and represent the degree of the polynomial ring. A polynomial $\mathbf{a}(x) = \sum_{i=0}^{N-1} a_i \cdot x^i$ in $\mathcal{R}_q$ can also be represented as a vector over $\mathbb{Z}_q$, so that $\mathbf{a} = [a_0, a_1, \ldots, a_{N-1}]$. CKKS has the message space consisting of $N/2$-dimensional complex vectors $\mathbb{C}^{N/2}$, plaintext space $\mathcal{R}_q$, and ciphertext space may expand to more than two polynomials. λ and Δ represent the security parameter and the scaling factor, respectively.

B. CKKS

CKKS is a fully homomorphic encryption scheme that supports real number operations. Compared with BGV and BFV, both of which are fully homomorphic encryption schemes that support integer-exact computation, CKKS has a wider range of applications, especially in the emerging computational fields such as machine learning and artificial intelligence.

CKKS consists of **KeyGen**, **Encode**, **Decode**, **Encrypt**, **Decrypt**, **Add**, **Multiply**, **Relinearize** and **Rescale**, which are the basic homomorphism operations. Below we describe these operations briefly, and the detailed introduction can be found in the original CKKS article [14]:

- **KeyGen**: Based on the given security parameter λ, a set of keys required for CKKS computation is generated, including the secret key (sk), public key (pk) and evaluation key (evk). sk and pk are used for the encryption and decryption, respectively, and evk is used for homomorphic evaluation operations.
- **Encode (Ecd)**: For a $(N/2)$-dimensional vector $z \in \mathbb{C}^{N/2}$, encoding it as a message m. The encoding procedure keeps the precision by multiplying a scaling factor Δ.
- **Decode (Dcd)**: It is the inverse operation of **Ecd**. For an input polynomial $m \in \mathcal{R}$, output the vector z.
- **Encrypt (Enc)**: For a given plaintext m and a public key pk, output a ciphertext c, where an encryption c of m is generated as $c = (v \cdot pk + (m + e_0, e_1) \bmod q_l$, in which e_0, e_1 are two Gaussian-distributed vectors.
- **Decrypt (Dec)**: For a ciphertext c at level l, output a polynomial m' for the secret key sk.
- **Add**: Add two input ciphertexts c_0 and c_1. For $c_0, c_1 \in \mathcal{R}_{q_l}^2$, the output ciphertext $c_{Add} = (c_0[0] + c_1[0], c_0[1] + c_1[1]) \bmod q_l$.
- **Multiply (Mul)**: For a pair of ciphertexts $(c_0, c_1) \in \mathcal{R}_{q_l}^2$, output a ciphertext $c_{mul} \in \mathcal{R}_{q_l}^3$, where $c_{Mul} = (c_0[0]c_1[0], c_0[0]c_1[1] + c_0[1]c_1[0], c_0[1]c_1[1]) \bmod q_l$.
- **Relinearize (RL)**: Decrease the size of the ciphertext back to 2 after multiplication, so that we always have the same size ciphertext with the same decryption circuit. Let $c_{Rl} = ((c_0, c_1) + \lfloor P^{-1} \cdot evk \rceil) \bmod q_l$, where P is a large auxiliary number used for basis conversion.
- **Rescale (RS)**: Manage the noise and avoid noise overflow after the multiplication. The rescale procedure reduces the $level$ l of the ciphertext, limiting the number of further homomorphic operations that can be performed. **RS** computes $c_{Rs} = \lfloor \frac{q_{l'}}{q_l} c \rceil \bmod q_{l-1}$.

The computational scheme of CKKS can be split into three layers: the bottom arithmetic computation, the intermediate computational primitive and the upper homomorphic computation layer. The bottom arithmetic layer contains operations such as addition, subtraction, multiplication, division, displacement, and reduction, the intermediate computational primitives mainly include NTT and CRT, and the upper homomorphic computation layer mainly includes the homomorphic computation operations of CKKS. The complete computational framework is shown in Figure 2.

C. Number Theoretic Transform

NTT is a special form of Discrete Fourier Transform (DFT) in a finite integer field, which can be used to accelerate multiplication operations and is widely applied in the field of cryptography. Any vector $a = [a_0, a_1, \ldots, a_{N-1}]$ which has N elements in the polynomial domain can be transformed to another vector $\bar{a} = [\bar{a}_0, \bar{a}_1, \ldots, \bar{a}_{N-1}]$ which also has N elements in the NTT domain, converting polynomial multiplication into element-wise multiplication of vectors in NTT form. This reduces the computational complexity of polynomial ring multiplication from $O(N^2)$ to $O(NlogN)$. The forward and inverse NTTs are defined as followed:

$$\bar{a}_i = \sum_{j=0}^{N-1} a_j \omega^{i \times j} \bmod q \text{ for } i = 0, 1, \ldots, N-1 \quad (1)$$

$$a_i = \frac{1}{N} \sum_{j=0}^{N-1} \bar{a}_j \omega^{-i \times j} \bmod q \text{ for } i = 0, 1, \ldots, N-1 \quad (2)$$

The NTT(Equation 1) and INTT (Equation 2) calculations require the powers of a constant value $\omega \in \mathbb{Z}_q$ referred as the twiddle factor.

D. NVIDIA GPU

A graphics processor Unit (GPU) is a piece of hardware specifically designed to handle graphics and parallel computing tasks, initially widely used in the field of deep learning. SIMT is a parallel computing architecture commonly used by GPUs to improve computational efficiency by performing the same operation on multiple data elements at the same time, and has also been widely used in cryptographic computing in recent years given this excellent data parallel processing capability. Compute Unified Device Architecture (CUDA) is a parallel computing architecture and programming model introduced by NVIDIA. CUDA can be used to divide computing tasks into multiple thread blocks and grids to achieve efficient parallel computing.

According to the physical location of the memory, The memory organization in CUDA can be divided into two categories: on-chip memory and off-chip memory. Off-chip memory has large capacity but high access latency, including global memory, constant memory and texture memory; on-chip memory includes registers and shared memory, with fast access speed but small memory capacity. In order to ensure the security of homomorphic computation, the schemes usually need a large space to store plaintext and ciphertext polynomials, so how to reasonably plan the memory resources on GPUs to improve the performance of CKKS computation is a major challenge.

III. Methodology

In this section, we present the implementation details and optimization techniques of HI-CKKS. First, we propose an asynchronous execution framework with batch processing support, aimed at reducing the latency caused by extensive data interactions. Second, we design a throughput-oriented performance optimization scheme, which includes a layered and hybrid implementation of the Number Theoretic Transform and multi-dimensional parallel homomorphic multiplication. By balancing memory access efficiency and the scale of parallelism, we achieve optimal throughput performance.

A. Throughput-Oriented Asynchronous Framework

In practical applications, the performance bottleneck of utilizing GPUs lies in the significant overhead associated with time and spatial dimensions. The traditional computational workflow involves the CPU aggregating computation data and transmitting it to the GPU upon generating a computational demand from the application service. After the GPU completes its calculations, the results are copied back to the CPU for the next computation task. In this approach, the frequent data transfers between the CPU and GPU introduce substantial latency, while the CPU must also wait for the GPU to execute calculations and copy data. This transmission delay and resource redundancy can exponentially increase with rising concurrency levels.

Additionally, we observe that typical applications refrain from altering application parameters—such as security parameters, data size N, and modulus chain length—within certain time intervals to ensure stability. This stability in application parameters results in computational data exhibiting similar characteristics, including data size and computational parameters that require processing during preprocessing. Based on this, we propose a batch-processing CPU-GPU asynchronous execution scheme to more effectively leverage GPU platforms for enhancing the computational performance of CKKS homomorphic multiplication, as illustrated in Figure 2.

In the asynchronous execution model we propose, tasks that do not require high concurrency, such as the pre-calculation of security parameters and rotation factors, are assigned to the CPU platform. Meanwhile, computationally intensive tasks, such as encoding and decoding, encryption and decryption, and homomorphic operations like addition and multiplication, are allocated to the GPU platform. In this asynchronous mode, the CPU first sends pre-processed input data and static data (such as security parameters and rotation factor matrices) to the GPU. Subsequently, according to computational demands, the CPU issues tasks and transfers data needing computation to the GPU without waiting for computational synchronization with the GPU. Once the computation data is transferred to the GPU, it can carry out continuous homomorphic computations without the need for repeated data transfers, thus better meeting the practical application requirements.

Figure 3 illustrates the computational workflow of our

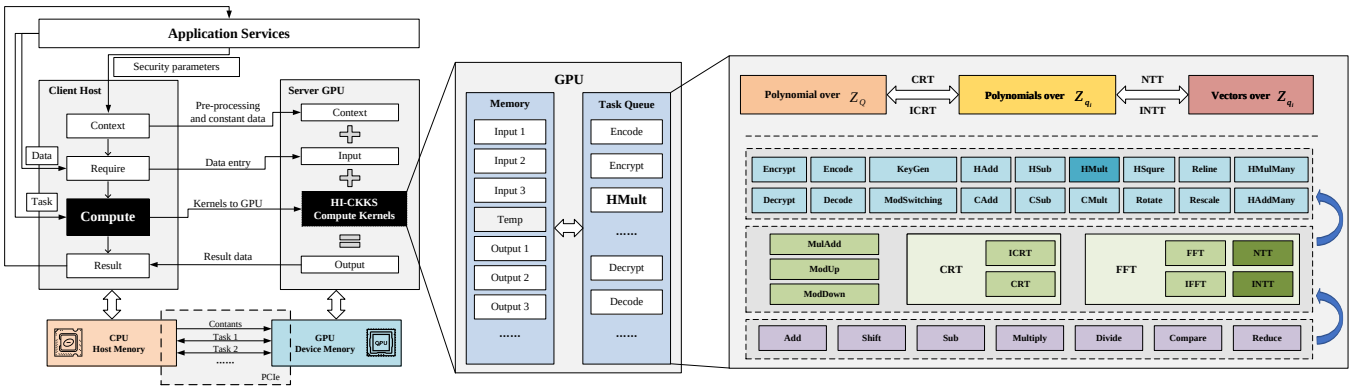


Fig. 2: Batch-Supported Asynchronous Execution Scheme and CKKS Layered Computing Framework

framework during batch processing tasks, compared to traditional synchronous schemes. In our asynchronous approach, after transmitting the data required for the first task to the GPU, there is no need to wait for the GPU to complete its computations; instead, the transmission of data required for the second task can commence immediately. When tasks two and three have data dependencies, the completion of task two allows for the direct computation of task three without the need to transfer intermediate results back and forth.

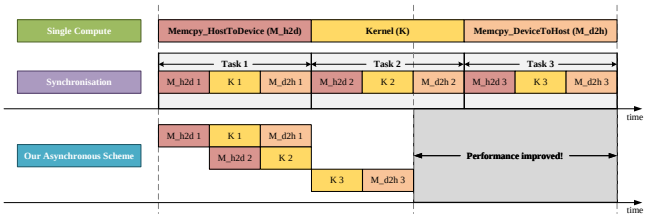


Fig. 3: Comparison of Workflows Between Asynchronous and Synchronous Schemes

In the context of multi-node, large-scale, and high-concurrency practical application demands, our proposed batch-processing asynchronous execution scheme supports: 1) the efficient execution of multiple homomorphic computation requests from the host for the same set of input data. Once the CPU completes the data transmission, multiple computation requests can be sent to the GPU in succession, reducing the frequency of data transfers and effectively mitigating the risk of side-channel attacks; 2) the packaging of operations for identical computation requests across multiple sets of input data. By leveraging the unique SIMD architecture of GPUs, we enhance data concurrency efficiency, significantly improving the computational performance of CKKS in real-world application scenarios. This asynchronous computation model substantially reduces both synchronization delays between the CPU and GPU and the latency associated with extensive data transfers.

B. Throughput-Oriented Hierarchical Hybrid NTT

In Figure 2, we present the layered architecture of the CKKS computation scheme, where (I)NTT is a critical component. The traditional multiplication of polynomials $c(x) = a(x) \times b(x)$, where $a(x), b(x) \in \mathcal{R}_q$, can be performed like the Equation 3:

$$c(x) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \times b_j \times x^{i+j} \mod q_l \tag{3}$$

Due to the quadratic computational complexity, using (I)NTT can reduce the complexity of multiplication by transforming polynomials from coefficient representation to point-value representation.

$$\bar{c}(x) = NTT(a(x)) \odot NTT(b(x)) \mod q_l \tag{4}$$

$$c(x) = INTT(\bar{c}(x)) \mod q_l \tag{5}$$

The multiplication in the NTT domain is described in Equation 4 and Equation 5. For the NTT-based polynomial multiplication operation, we have $\bar{a}(x) = NTT(a(x))$ and $\bar{b}(x) = NTT(b(x))$, after multiplication the result $\bar{c}(x)$ in the NTT domain is obtained, and after INTT obtaine $c(x)$.

As shown in Equation 1 and Equation 2, (I)NTT can be decomposed into smaller computational units, which exhibit significant data parallelism, making them highly suitable for performance optimization using GPUs. Therefore, we first analyze the implementation of NTT operations on the GPU, as illustrated in Algorithm 1.

Algorithm 1 Native NTT Implementation on GPU

Input:
A polynomial $a(x) \in \mathbb{Z}_q[x]/\{x^N + 1\}$ in standard order
The modulus q_i where $0 < i < l$
The twiddle factor ω
The polynomial coefficient n
Output:
Polynomial $\bar{a}(x) \in \mathbb{Z}_q[x]/\{x^N + 1\}$ in bit-reversed order
Step (1): Perform NTT
1: $layer \leftarrow 1$
2: **while** $layer \leq \log N$ **do**
3: Launch kernel **kNTT**, let thread number $t \in [0, N/2]$
4: $gap \leftarrow 1 \ll (\log N - layer)$
5: $rid \leftarrow (1 \ll (layer - 1)) + (t \gg (\log N - layer)$
6: $index \leftarrow ((t \gg (\log N - layer) \ll (\log N - layer)) + (t \ \& \ (gap - 1))$
7: **for** $i = 0$ *to* l **do**
8: $2q \leftarrow 2 \times q_i$
9: $j \leftarrow (i \ll \log N) + index$
10: $U \leftarrow a_j$
11: $V \leftarrow a_{j+gap} \cdot \omega_{rid} \pmod{q_i}$
12: **if** $U > 2q$ **then**
13: $U \leftarrow U - 2q$
14: **end if**
15: $a_j \leftarrow U + V$
16: $a_{j+gap} \leftarrow U + 2q - V$
17: **end for**
18: $layer \leftarrow layer + 1$
19: **end while**
Step (2): Bound $a(x)$ **between** $[0, q]$
20: Launch kernel **kBound**, let thread number $t \in [0, N]$
21: **for** $i = 0$ *to* l **do**
22: $2q \leftarrow 2 \times q_i$
23: $j \leftarrow i \times N + t$
24: **if** $a_j \geq 2q$ **then**
25: $a_j \leftarrow a_j - 2q$
26: **end if**
27: **if** $a_j \geq q_i$ **then**
28: $a_j \leftarrow a_j - q_i$
29: **end if**
30: **end for**

We observe that the NTT implementation comprises several arithmetic operation modules, including addition (LINE 15-16), multiplication (LINE 11), Barrett reduction (LINE 11), and conditional subtraction (LINE 12-14, 24-19). These arithmetic modules require numerous registers to store intermediate variables during execution and also suffer from issues related

to redundant machine instructions. Furthermore, we find that the NTT operations require a total of $\log N + 1$ kernels and $\frac{N}{2}\log N + N$ threads. As N increases, this computational overhead also escalates. Additionally, in traditional NTT implementations, all data is directly read from and written to global memory, presenting significant opportunities for optimization.

Therefore, based on the characteristics of the GPU hardware platform, we designed a layered and hybrid implementation of the NTT. This approach includes replacing complex arithmetic modules with constant-time PTX instruction sets, fusing a large number of computation kernels, and employing optimization strategies such as hybrid memory access utilizing the unique memory architecture of GPUs. We conducted comprehensive experimental tests on these optimization strategies to identify the balance point between access efficiency and parallel dimensions in order to achieve maximum throughput performance.

*1) **Constant-Time Instruction Implementation**:* The fundamental arithmetic modules used in the CKKS scheme include addition, multiplication, multiply-accumulate, Barrett reduction, and conditional subtraction for modulus reduction. We have implemented these arithmetic operations in constant time using the CUDA PTX instruction set. This article presents the inline function implementations for multiply-accumulate, conditional subtraction, and Barrett reduction.

Multiply-accumulate. In the homomorphic multiplication calculations of the CKKS scheme, it is necessary to perform multiply-accumulate operations of the form

$$c_1 = a_0 b_1 + a_1 b_0 \quad a_0, a_1, b_0, b_1 \in [0, 2^{64})$$

where c_1 is the intermediate vector result of the multiplication calculation, while a_0, a_1 and b_0, b_1 are two vectors representing the multipliers a, b respectively, both of which are 64-bit operands.

In traditional 64-bit multiply-accumulate operations, eleven 64-bit registers are typically needed to store four 64-bit multiplication operands, two 128-bit multiplication results, one 128-bit addition operand, and one rounding bit. The process starts with two 64-bit multiplications, followed by a 128-bit addition, requiring a total of 6 instructions. In our optimized scheme, however, the 64-bit multiply-accumulate operation can be completed in just 4 steps and 6 registers using the `mad` and `madc` instructions to perform the high 64-bit multiply-accumulate operations, with the carry flag implicitly stored in the `madc` instruction.

```
asm volatile(
    mul.lo.u64 %0, %2, %5;
    mad.lo.cc.u64 %0, %4, %3, %0;

    madc.hi.u64 %1, %2, %5, 0;
    mad.hi.u64 %1, %4, %3, %1;

    : "+l"(c[0]), "+l"(c[1])
    : "l"(a0), "l"(b0), "l"(a1), "l"(b1)
);
```

Conditional Subtraction. In traditional conditional subtraction calculations, logical judgment instructions are commonly used to perform approximate subtraction to reduce the operands to the range of $[0, 2q_l)$ and $[0, q_l)$, like

$$op = \begin{cases} op - 2q_l & \text{if } op > 2q_l \\ op - q_l & \text{if } op > q_l \text{ and } op \leq 2q_l \\ op & \text{otherwise} \end{cases}$$

where q_l is the modulus and op is the input data. However, GPUs are not adept at handling logical judgment instructions, which can affect the efficiency of large-scale computations. Therefore, we use addition, subtraction, bit shifting, and simple instructions instead of logical judgment instructions to implement conditional subtraction.

```
asm volatile(
    sub.s64 %0, %2, %3;
    shr.s64 %1, %0, 63;
    and.b64 %1, %1, %3;
    add.s64 %0, %0, %1;

    : "+l"(result), "+l"(tmp)
    : "l"(op), "l"(modulus)
);
```

Barrett Reduction. The Barrett reduction algorithm is widely used in the CKKS scheme, primarily aimed at replacing complex large integer division operations with simpler additions, subtractions, multiplications, and bit shifts, thereby enhancing computational efficiency. We have adopted the fast implementation method proposed by Shoup [23] to reduce computational overhead. HI-CKKS provides optimized implementations for both 64-bit and 128-bit subtraction algorithms. Below is an inline function example of Barrett reduction using 128-bit operands. During the Barrett reduction process, we use a fixed parameter $\gamma = \lfloor \frac{2^i}{q_l} \rfloor$ related to the modulus, which is pre-calculated on the CPU and stored in the GPU constant memory. This approach effectively reduces the usage of registers and enhances memory efficiency. We denote the lower bits of γ as `ratio_0` and the upper bits as `ratio_1`.

```
asm volatile(
    mul.hi.u64 %0, %1, %4;
    mad.lo.cc.u64 %0, %1, %5, %0;

    madc.hi.u64 %3, %1, %5, 0;
    mad.lo.cc.u64 %0, %2, %4, %0;

    madc.hi.u64 %3, %2, %4, %3;
    mad.lo.u64 %3, %2, %5, %3;

    mul.lo.u64 %3, %3, %6;
    sub.u64 %3, %1, %3;

    sub.s64 %3, %3, %6;
    shr.s64 %0, %3, 63;
    and.b64 %0, %0, %6;
    add.s64 %3, %3, %0;

    : "+l"(tmp), "+l"(op[0]), "+l"(op[1]),
    "+l"(result)
    : "l"(ratio_0), "l"(ratio_1),
    "l"(modulus)
);
```

*2) **Unified Kernel Fusion**:* According to Algorithm 1, we observe that traditional NTT computation requires launching $\log N + 1$ kernel functions to complete the operations, including $\log N$ kNTT kernel functions (Lines 2–19) for performing butterfly operations and 1 kBound kernel function (Lines 20–30) for reduction operations. In total, this demands $\frac{N}{2}\log N + N$ thread resources. Meanwhile, due to the RNS system, each operand has l corresponding modular operations (where $0 < l < L$), which increases global memory access by a factor of l.

However, in practical applications with large computational workloads, the frequent launches of $\log N + 1$ kernels per NTT operation and the $\frac{5}{2}lN$ global memory accesses cannot meet the high-throughput demands. Therefore, we performed unified kernel fusion and hierarchical parallel optimization on the (I) NTT computation kernels.

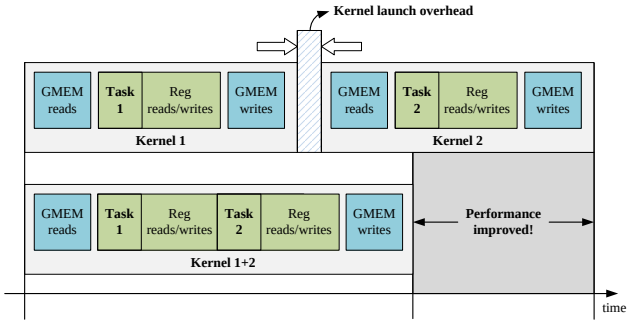


Fig. 4: Illustration of Kernel Fusion: A Dual Kernel Example

We fused the $\log N + 1$ kernels into a single kernel computation function, launching $N/2$ threads to execute the (I) NTT operation. Figure 4 illustrates the dual-kernel fusion scheme. The fusion of kernel execution offers two major advantages. On the one hand, it is well known that the kernel boot process needs to be warmed up, and the boot latency is long compared to the kernel computation execution time, kernel fusion for (I)NTT can reduce the boot latency of $logN + 1$ kernels to 1, which greatly reduces the boot time. On the other hand, the kernel execution process includes reads and writes to global memory, which are more time-consuming than those to other on-chip memory. Executing kernel fusion can make more effective use of the memory resources on the GPU, reduce global memory reads and writes, and reduce the number of $(2logN + 2) \times l$ operations to $2l$ operations, which can effectively improve data reuse efficiency and computation efficiency.

*3) **Hybrid Memory Access Optimization Strategy**:* The CUDA GPU memory architecture is highly complex, with significant differences in access efficiency between on-chip and off-chip memory. The speed of accessing different types of memory on the GPU can be ranked as follows: registers $>$ shared memory $>$ constant memory $>$ texture memory $>$ global memory. Traditional (I) NTT computation typically only utilizes global memory. To fully exploit the unique memory architecture of the GPU and improve memory access efficiency, we designed a hierarchical and hybrid memory access strategy. First, input and output data are stored in global memory, while precomputed twiddle factors and other spatially structured data are stored in texture memory. Precomputed modulus values and other unchanging deterministic data are stored in constant memory. During computation, registers and shared memory are selectively used to store intermediate variables.

We observed that for (I) NTT operations, overall performance is optimal when data is parallelized by factors of 2^2 or 2^3. Based on this observation, we developed a hierarchical NTT implementation using the aforementioned optimization strategies. For example, in an NTT computation with 8 operands processed in parallel, the data order of the polynomial $a(x)$ alternates between standard order and bit-reversed order during the naive NTT computation. In the process of parallel optimization, sequentially reading operands does not yield significant performance improvements. Therefore, we assign each thread to compute two operands, a_j and a_{j+gap}, to enhance performance.

If parallel optimization is performed by fetching numbers sequentially, then $\frac{N}{8}$ threads will be launched, $t \in [0, \frac{N}{8}]$, with each thread fetching 8 operands

$$\{a_i \mid j \le i \le j+3\} \cup \{a_i \mid j + \text{gap} \le i \le j + \text{gap} + 3\}$$

where

$$\begin{aligned} index = &(4t \gg (\log N - layer) \ll (\log N - layer)) \\ &+ (4t \ \&\ (gap - 1)) \end{aligned}$$

where gap is the interval between operands that changes according to the layer, and $index$ is the offset of the operands a under each modulus, t is the thread number. In this case, after performing $4 \times l$ butterfly transformations, each thread will synchronize and exchange data with the global variable once, as shown in Figure 5c. At this time, the number of threads launched is reduced by $\frac{1}{2}N \cdot \log N + \frac{7}{8}N$ compared to the naive implementation, but the number of times threads synchronize and the amount of interaction with global variables has not significantly decreased.

If the characteristics of bit-reversed order are considered, selecting 8 operands at intervals of $gap = gap \gg 2$. The 8 operands selected by each thread are

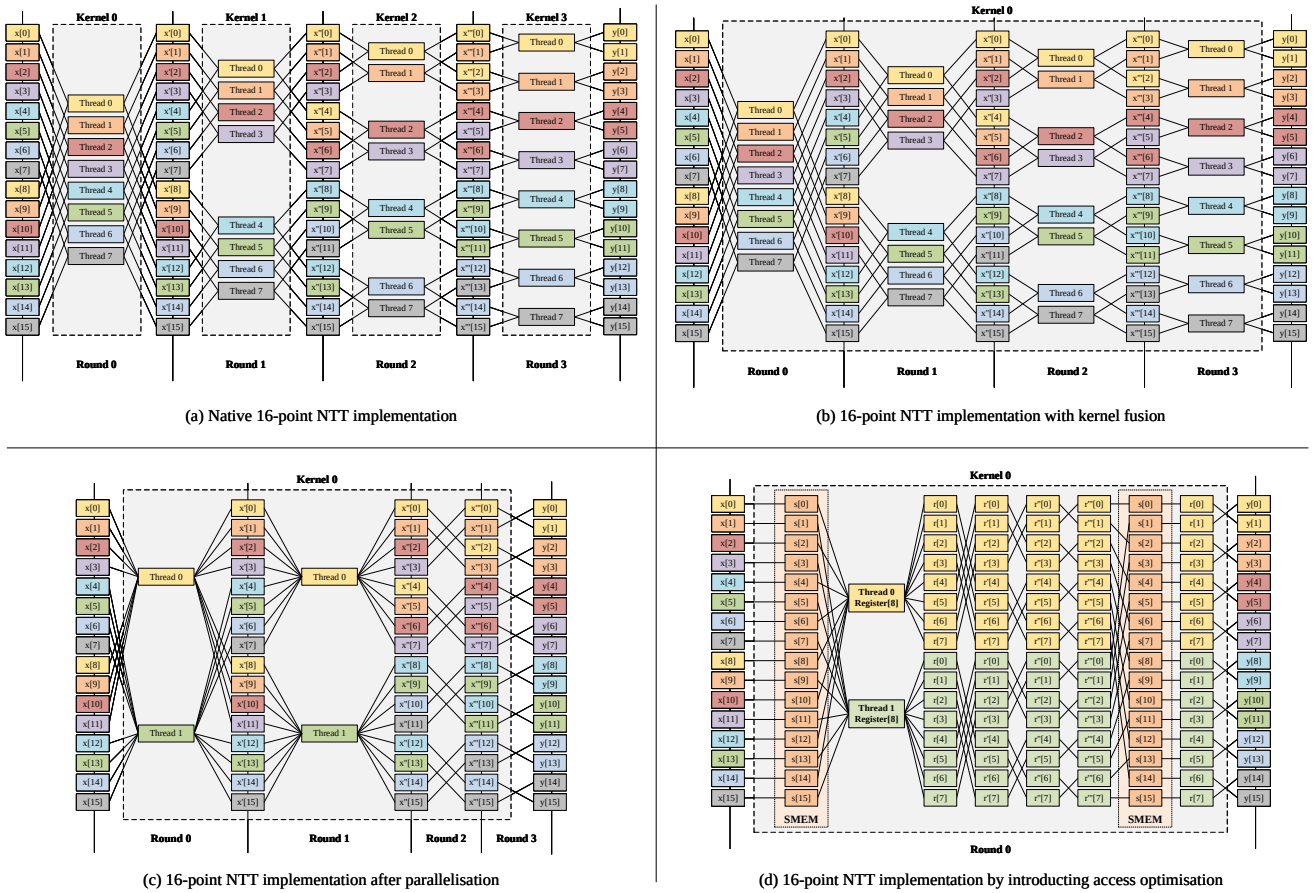
$$\left\{a_{j+\frac{k}{4}\text{gap}} \mid k = 0, 1, 2, \ldots, 7\right\}$$

where

$$index = 8 \times (gap \gg 2) \times (\frac{t}{gap \gg 2})) + (t \ \%\ (gap \gg 2))$$

In this case, after each thread has fetched the operands and performed 4 butterfly transformations for the layer level, it can still perform butterfly transformations for layers $layer + 1$ and $layer + 2$, reducing the interaction with global variables by 4 times. That is, for a parallel dimension of 2^m, where $1 \le m \le \log N$, fetching data once can perform butterfly transformations for $\log m$ layers, reducing the interaction with global variables by 2^{m-1} times.

We take the NTT implementation with N of 16 as an example, and Figure 5 details our 16-point NTT implementation after applying kernel fusion, parallelism and hybrid access



(a) Native 16-point NTT implementation

(b) 16-point NTT implementation with kernel fusion

(c) 16-point NTT implementation after parallelisation

(d) 16-point NTT implementation by introducting access optimisation

Fig. 5: Illustration of 16-Point NTT with Hierarchical Hybrid Optimisation on GPU

optimisation. 5(a) is a native 16-point NTT implementation. For an NTT with N of 16, it is necessary to perform $logN$ butterfly transformations, and $logN$ kernels were launched to perform each butterfly transformation, each accessing data from global memory, resulting in a total of $logN \times N/2 = 32$ threads being activated and 4 global memory synchronizations being performed. 5(b) is the 16-point NTT implementation after applying kernel fusion, which starts 1 kernel to perform 4 butterfly transformations, saving 3 kernel startup latencies. 5(c) is the NTT implementation after implementing the parallel strategy, we have chosen the case of a single thread processing 8 operands, where a total of $16/8 = 2$ threads are required to perform the computation for each butterfly transformation. At the same time, due to the kernel fusion strategy, 8 operands can be threaded to perform 3 consecutive butterfly transformations without global memory accesses, which reduces the number of required threads and global memory accesses. 5(d) shows the 16-point NTT implementation after using the hybrid access strategy. For NTT computation with $N = 16$, 16 operands can be stored in the shared memory, and then the corresponding 8 parallel operands can be fetched from the shared memory by the threads, and the global data synchronisation can be achieved by performing only thread synchronisation within the block after performing 3 butterfly transformations. Under this access strategy, the global memory accesses are reduced from 8 to 2, the number of global thread synchronisations is reduced from 4 to 1, of which the in-block synchronisation is performed twice, and the number of enabled threads is reduced from 32 to 2, thereby effectively improving resource utilization. When applied to larger values of N, the performance improvement will be more significant.

C. *Throughput-Oriented Multi-Dimensional Parallel HMult*

We analyzed the implementation of homomorphic multiplication on the GPU, as shown in Algorithm 2. During the computation, we first perform multiplication on two 64-bit operands in the NTT domain, followed by Barrett reduction on the 128-bit multiplication result. For multiplication with parameter N, N threads are launched, where each thread performs homomorphic multiplication on two ciphertext operands for the corresponding modulus.

We designed a multi-dimensional parallel scheme and optimized memory by implementing a strategy for contiguous memory copying to enhance the efficiency of the homomorphic multiplication.

*1) **Multi-Dimensional Parallel Processing**:* In this algorithm, traditional homomorphic multiplication is first carried out on two 64-bit operands, for two input polynomials $c(x) = (c_0, c_1)$ and $c'(x) = (c'_0, c'_1)$ in the NTT domain, obtain the multiplication result:

$$\bar{C} = (d_0, d_1, d_2) = (c_0 \cdot c'_0, c_0 \cdot c'_1 + c'_0 \cdot c_1, c_1 \cdot c'_1)$$

Algorithm 2 Baseline HMult Implementation on GPU

Input:
Two ciphertext vectors $c(x) = (c_0, c_1), c'(x) = (c'_0, c'_1)$
The modulus q_i where $0 < i < l$
A precomputed number $\mu = \lfloor \frac{2^{64}}{q_i} \rfloor = 2^{64} \cdot \mu_1 + \mu_0$
The polynomial coefficient N
Output:
Ciphertext vectors $\bar{C} = (d_0, d_1, d_2) = (c_0 \cdot c'_0, c_0 \cdot c'_1 + c'_0 \cdot c_1, c_1 \cdot c'_1)$

for $i = 0$ *to* 3 **do**
  $m_1 \leftarrow \min(i, 1)$
  $m_2 \leftarrow \min(i, 1)$
  $m_3 \leftarrow i - m_2$
  $step \leftarrow m_1 - m_3 + 1$
  Launch kernel **kHMult**, let thread number $t \in [0, N]$
  for $j = 0$ *to* l **do**
    for $k = 0$ *to* $step$ **do**
      $o_1 \leftarrow c_{(k \times l + j) \times N + t}$
      $o_2 \leftarrow c'_{(j - k \times l) \times N + t}$
      $res \leftarrow i \times N \times l + j \times N + t$
      Step (1): 64-bit Multiply
      $z_0 \leftarrow o_1 \times o_2$
      $z_1 \leftarrow (o_1 \times o_2) \gg 64$
      Step (2): 128-bit Barrett Reduction
      $carry \leftarrow (z_0 \times \mu_0) \gg 64$
      $t_0 \leftarrow z_0 \times \mu_1$
      $t_1 \leftarrow (z_0 \times \mu_1) \gg 64$
      $tmp \leftarrow t_0 + carry$
      if $tmp < t_0$ **then**
        $tmp' \leftarrow t_1 + 1$
      else
        $tmp' \leftarrow t_1$
      end if
      $t_0 \leftarrow z_1 \times \mu_0$
      $t_1 \leftarrow (z_1 \times \mu_0) \gg 64$
      $tmp \leftarrow z_1 \times \mu_1 + tmp' + t_1$
      $tmp' \leftarrow z_0 - tmp \times q_i$
      $\bar{C}_{res} \leftarrow \bar{C}_{res} + tmp'$
      if $\bar{C}_{res} \geq q_i$ **then**
        $\bar{C}_{res} \leftarrow \bar{C}_{res} - q_i$
      end if
    end for
  end for
end for

Then followed by Barrett reduction on the 128-bit multiplication result $\bar{C}$. For multiplication under the parameter N, N threads are required to perform the pointwise multiplication of two ciphertext polynomials, a process well-suited for parallel optimization on the GPU.

Therefore, for an N-dimensional polynomial, we designed computational schemes under multiple parallel dimensions $\{N, N/2, N/4, \ldots, 256\}$, where in each scheme, each thread sequentially processes $\{1, 2, 4, \ldots, N/256\}$ ciphertext operands.

After parallel optimization, for example, with each thread processing 8 ciphertext operands, only $N/8$ threads are launched, with each thread fetching 8 ciphertext operands from two consecutive ciphertext polynomials, reducing thread resource usage by a factor of 8. We conducted detailed performance tests on these optimization schemes to determine the optimal parallel dimension for maximizing throughput for a given parameter N.

*2) **Sequential Memory Copy**:* During the parallel optimisation of homomorphic multiplication, unlike NTT, the data in homomorphic multiplication computations do not exhibit strong interdependence, and parallel operation allows threads to process multiple consecutive operands at the same time, so we use `uint4` to perform an efficient memory copy operation in GPU.

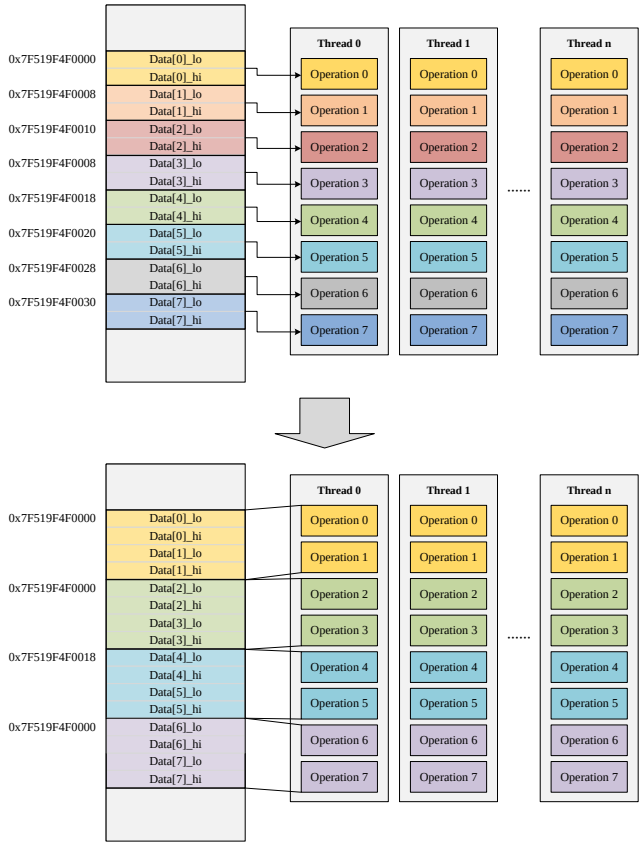


Fig. 6: Example of $N/8$-Dimensional Parallelism and Memory Sequential Copy

In GPU, data is considered as one word with 4 bytes (32 bits), while in CKKS the operand data occupies 8 bytes, which equals two words. `uint4` is a data representation in CUDA that represents a data structure containing four unsigned integers (`uint`). As shown in Figure 6, when we use `uint4` for data copying, a single operation can actually handle four `uint` elements, which equates to two 2 operands. This method of data copying, in comparison to copying elements individually, such as in a `for` loop, can reduce the number of memory accesses and more efficiently utilize memory bandwidth during computation. Additionally, using `uint4` for continuous memory copying allows a single instruction to process multiple pieces of data, reducing the number of instructions per thread by 50%. Processing multiple contiguous

data elements in a memory-aligned manner can reduce the total number of memory transactions required, as it allows for larger blocks of data to be processed at once on the GPU, rather than handling individual pieces of data in multiple transactions.

In homomorphic multiplication with polynomial coefficient N using contiguous memory copying, assuming that each thread performs m multiplications after parallel optimization, the number of thread launches can be reduced by $N - \frac{N}{m}$ times. Each thread reduces m times memory accesses, and a complete homomorphic multiplication operation can reduce memory accesses by a total of $m \times \frac{N}{m} = N$ times.

IV. Performance Evaluation

In this section, we implement a high-performance HI-CKKS based on a GPU platform and select the optimal configuration parameters through multiple rounds of testing. Finally, we compare the peak results with the related work.

A. *Experiment Setup*

In order to test the performance of HI-CKKS, we implemented it in the NVIDIA GeForce RTX 4090 environment. The detailed hardware configuration of the test environment and the corresponding platforms are summarised in Table I. Our experiments are all implemented based on the SEAL library. For all experiments, we compiled the programs using gcc 9.4.0 for the CPU and CUDA 12.3 for the GPU implementation. The experiments measured throughput performance in kop/s, representing thousands of transactions processed per second.

TABLE I: The Platform Configuration of HI-CKKS

Platform	Device
OS	Linux Ubuntu 20.04.1, CUDA 12.4.0
CPU	AMD Ryzen 5 5600G 64-bit, 6 cores, 1.4 GHz
GPU	NVIDIA GeForce RTX 4090, 16384 cores, 2.52Ghz

Our proposed optimization scheme HI-CKKS supports implementations across multiple values of N. In the experimental section, we selected $|N| = 2^{13}$ as an example for comprehensive testing to identify the performance peak.

B. *Optimal Parallelism Parameter*

Our performance optimisation scheme employs a parallel strategy, so we performed a full range of throughput tests on the parallel dimensions of NTT, INTT and homomorphic multiplication in order to find the optimal parallel parameters in the current test environment. For (I)NTT, we observed different scenarios where a single computation handles $\{2, 4, 8, 16\}$ operands, corresponding to the number of threads used for a single computation being $\{4096, 2048, 1024, 512\}$. Different thread configurations exist for the same number of thread requirements, for example, under the 4096 parameter, we can use (4, 1024) to start 4 blocks with 1024 threads in each block, or we can use (8, 512) to start 8 blocks with 512 threads in each block. During the throughput test, we randomly generate m sets of input data and transfer them to the GPU. The value of m is determined by the number of threads required for computation and the thread configuration, ranging from $[8, 256]$.

As we use the optimisation idea of kernel fusion, there are dependencies between the data in multiple blocks during the computation process, and data access conflicts may occur in the parallel case resulting in computation errors. In the traditional (I)NTT computation process, multiple kernel starts will indirectly complete the global memory synchronisation. We used `grid.sync()` to complete the global memory synchronisation. To address the issue of intra-block data conflicts, we utilize the GPU's intra-block thread synchronization function, `syncthreads()`.

In general, we believe that a larger parallel dimension theoretically implies a larger throughput, as a single thread performs more operations, more computations can be executed in a single task, given the same total number of available threads, based on the nature of GPU thread parallelism. However, the experimental results in Figure 7 and Figure 8 demonstrate that while increasing the number of threads can improve throughput to some extent, this improvement is not entirely linear. For the same number of threads, in some cases, the smaller number of threads required for computation has better throughput, which is consistent with the theoretical analysis. In some cases, a smaller number of threads has higher latency and lower throughput performance, which may be due to GPU computational resource constraints.

On the other hand, for the same number of threads required for a computation, different thread-starting configurations can present different performance. It is often assumed that maxing out the number of threads maximises the use of resources on the GPU, i.e. a local optimum point occurs when 1024 threads are started per block. However, the actual test results show that not all results show this property and some local optima do not occur in this case. This may be due to the number of registers and other resource constraints on the GPU, resulting in starting 1024 threads with high latency and low throughput. When the number of required threads becomes large or the resource demand is small, the resource conflict may not be obvious, allowing the full occupancy of threads to reach the optimal point.

Finally, after experimental testing, we obtained the optimal parallel parameters for NTT computation as using 2048 threads for a single computation, with threads starting as (4, 512), which means starting 4 blocks with 512 threads in each block. The optimal parallelism parameter for INTT computation is a single computation using 1024 threads, with a startup configuration of (1, 1024), which means that 1 block computation is started and 1024 threads are started in each block. At this point, the number of registers that can be used by each thread is limited to 63 due to the limitation of the total number of registers on the GPU. Figure 9 shows the throughput test case for homomorphic multiplication, using 4096 threads to complete a single computation, with the optimal parallel configuration when threads are started in the (4, 1024) configuration.

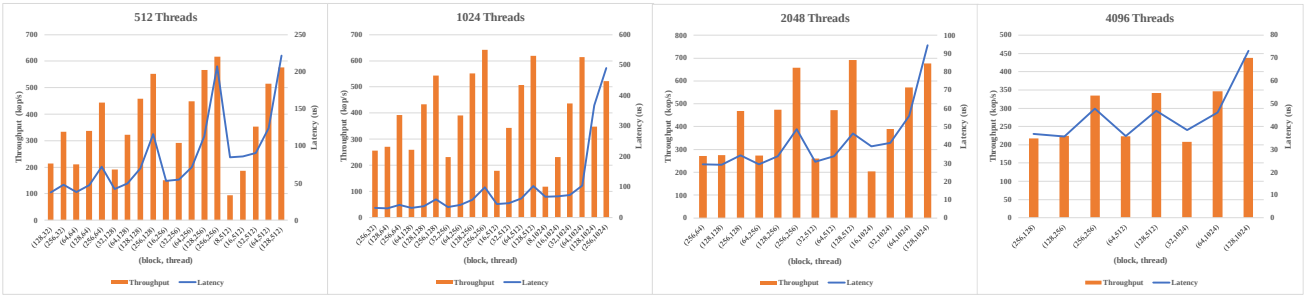


Fig. 7: Performance Results and Peak Behavior of NTT under Different Parallel Parameters and Dimensions

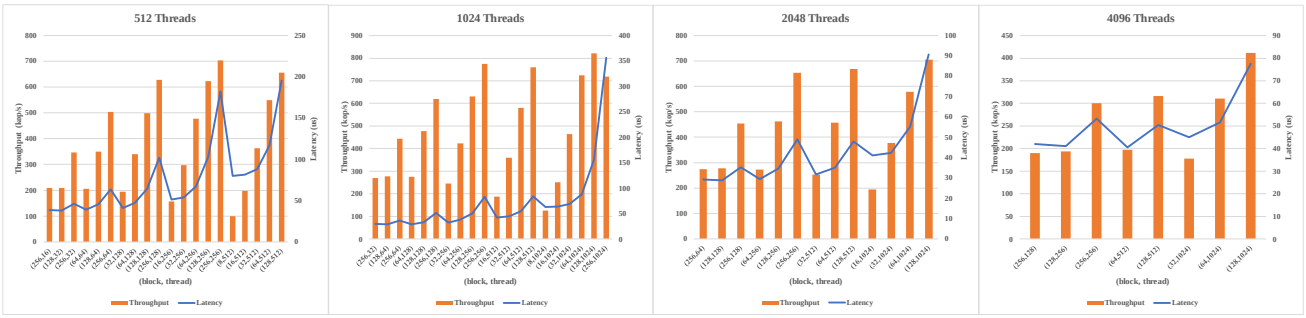


Fig. 8: Performance Results and Peak Behavior of INTT under Different Parallel Parameters and Dimensions

TABLE II: Implementation Performance Comparison

Implementation	**Platform**	**Throughput (kop/s)**					
		NTT		INTT		HMul	
SEAL [15]	AMD Ryzen 5 5600G 64-bit, 6 cores, 1.4 GHz	3.95	175.08×	4.29	191.27×	3.57	679.57×
Özcan et al. [24]	NVIDIA RTX 3060Ti	67.11	10.31×	58.14	14.12×	-	-
Yang et al. [22]	NVIDIA Tesla A100 80G PCIe GPU	49.60	13.95×	55.83	14.70×	-	-
TROY [25]	NVIDIA GeForce RTX 4090, 16384 cores, 2.52Ghz	25.72	26.91×	24.93	32.92×	3.50	693.17×
Shen et al. [19]	NVIDIA Tesla V100S PCIe	64.94	10.66×	60.98	13.46×	43.48	55.80×
Fan et al. [20]	NVIDIA A100-SXM-40GB	449.97	1.54×	449.08	1.84×	275.64	8.80×
Our HI-CKKS	NVIDIA GeForce RTX 4090, 16384 cores, 2.52Ghz	692.02		820.90		2426.08	

C. *(I)NTT and Homomorphic Multiplication*

In this section, we evaluated the throughput performance of HI-CKKS, comparing NTT, INTT, and homomorphic multiplication implementations on CPU, GPU, and against the related achievements, as shown in Table II. Throughput performance is calculated based on the number of operations that can be performed per second. In the case of $|N| = 2^{13}$, the baseline implementation on the CPU is based on SEAL v4.1 [15].

Based on the optimal parallelism parameters derived above, we tested the throughput performance of HI-CKKS. The optimal peak throughput performance obtained in the experimental environment is $692.09kop/s$ for NTT, $820.90kop/s$ for INTT, and $2426.08kop/s$ for homomorphic multiplication operation.

Experimental results indicate that our NTT implementation achieved a performance increase of $175.08\times$ compared to the CPU implementation. TROY [25], an open-source library developed based on SEAL for GPUs, has increased its performance by $26.91\times$ compared to the GPU baseline, and its throughput performance has improved by $1.54\times$ to $13.95\times$ compared to other related work. The INTT implementation saw a throughput performance increase of $191.27\times$ compared to the CPU, a $32.92\times$ increase compared to the GPU implementation, and an improvement of $1.84\times$ to $14.70\times$ compared to other related work. The throughput performance improvement for homomorphic multiplication was the most significant, with an increase of $679.57\times$ compared to the CPU implementation, a $693.17\times$ increase compared to the GPU implementation, and an improvement of $8.80\times$ to $55.80\times$ compared to other related work.

V. Conclusion

In this work, we propose HI-CKKS, a performance optimization scheme for CKKS homomorphic multiplication on GPU platforms, aimed at achieving optimal throughput.

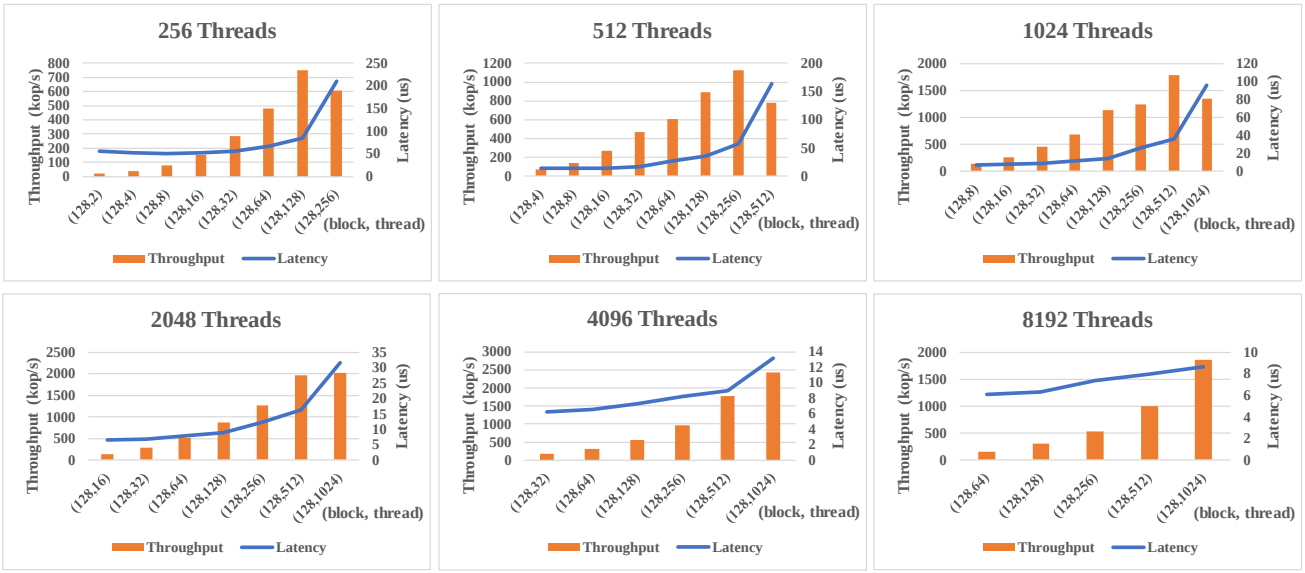


Fig. 9: Performance Results and Peak Behavior of HMult under Different Parallel Parameters and Dimensions

We design an asynchronous execution scheme supporting batch processing between the CPU and GPU, propose a hierarchical hybrid NTT and a multi-dimensional parallel optimization for homomorphic multiplication, and optimize the underlying algorithms and middle layer primitives for throughput improvement. Experimental results demonstrate the effectiveness of our approach. In the future, we plan to further leverage the hardware characteristics of GPUs to optimize additional middle-layer primitives and homomorphic operations of CKKS, aiming to improve the overall throughput performance of the CKKS algorithm.

References

[1] J. R. Saura, D. Ribeiro-Soriano, and D. Palacios-Marqués, "From user-generated data to data-driven innovation: A research agenda to understand user privacy in digital markets," *International Journal of Information Management*, vol. 60, p. 102331, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0268401221000244

[2] A. Ali, H. A. Maghawry, and N. Badr, "Performance testing as a service using cloud computing environment: A survey," *Journal of Software: Evolution and Process*, vol. 34, no. 12, p. e2492, 2022.

[3] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, "A guide to fully homomorphic encryption," *Cryptology ePrint Archive*, 2015.

[4] V. Mothukuri, R. M. Parizi, S. Pouriyeh, Y. Huang, A. Dehghantanha, and G. Srivastava, "A survey on security and privacy of federated learning," *Future Generation Computer Systems*, vol. 115, pp. 619–640, 2021.

[5] C. Zhao, S. Zhao, M. Zhao, Z. Chen, C.-Z. Gao, H. Li, and Y.-a. Tan, "Secure multi-party computation: theory, practice and applications," *Information Sciences*, vol. 476, pp. 357–372, 2019.

[6] C. Dwork, "Differential privacy," in *International colloquium on automata, languages, and programming*. Springer, 2006, pp. 1–12.

[7] V. Sidorov, E. Y. F. Wei, and W. K. Ng, "Comprehensive performance analysis of homomorphic cryptosystems for practical data processing," *arXiv preprint arXiv:2202.02960*, 2022.

[8] NVIDIA, "Cuda c++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2024, accessed: 2024-04-26.

[9] J. Dong, G. Fan, F. Zheng, T. Mao, F. Xiao, and J. Lin, "Tegras: An efficient tegra embedded gpu-based rsa acceleration server," *IEEE Internet of Things Journal*, vol. 9, no. 18, pp. 16 850–16 861, 2022.

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

[11] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*. Springer, 2011, pp. 129–148.

[12] 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.

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

[14] 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.

[15] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[16] Homomorphic Encryption Library (HElib) Community. (2021) HElib. [Online]. Available: https://github.com/homenc/HElib

[17] W. Dai and B. Sunar, "cuhe: A homomorphic encryption accelerator library," in *Cryptography and Information Security in the Balkans: Second International Conference, BalkanCryptSec 2015, Koper, Slovenia, September 3-4, 2015, Revised Selected Papers 2*. Springer, 2016, pp. 169–186.

[18] W. Jung, S. Kim, J. H. Ahn, J. H. Cheon, and Y. Lee, "Over 100x faster bootstrapping in fully homomorphic encryption through memory-centric optimization with gpus," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 114–148, 2021.

[19] S. Shen, H. Yang, Y. Liu, Z. Liu, and Y. Zhao, "Carm: Cuda-accelerated rns multiplication in word-wise homomorphic encryption schemes for internet of things," *IEEE Transactions on Computers*, 2022.

[20] 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.

[21] G. Fan, F. Zheng, L. Wan, L. Gao, Y. Zhao, J. Dong, Y. Song, Y. Wang, and J. Lin, "Towards faster fully homomorphic encryption implementation with integer and floating-point computing power of gpus," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 798–808.

[22] H. Yang, S. Shen, W. Dai, L. Zhou, Z. Liu, and Y. Zhao, "Phantom: A cuda-accelerated word-wise homomorphic encryption library," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[23] V. Shoup *et al.*, "Ntl: A library for doing number theory," 2001.

[24] A. Ş. Özcan, C. Ayduman, E. R. Türkoğlu, and E. Savaş, "Homomorphic encryption on gpu," *IEEE Access*, 2023.

[25] "Troy," https://github.com/lightbulb128/troy, Dec. 2023.

Fuyuan Chen direct Ph.D. student in Cyberspace Security at the School of Computer Science, Nanjing University of Posts and Telecommunications, received the B.E. degree from Internet of Things Engineering at the same university. Her primary research areas include homomorphic encryption and cryptographic engineering, to advancing the application and dissemination of cryptographic algorithms through hardware acceleration.

Jiankuo Dong received the B.E. degree from the Xi'an Jiaotong University, and the Ph.D. degree from the University of Chinese Academy of Sciences in 2014 and 2019, respectively. He is currently an Assistant Professor with School of Computer Science, Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering, public key cryptography and applied cryptography.

Xiaoyu Hu received the B.E. degree from Nanjing University of Posts and Telecommunications, and is pursuing master's degree in cyberspace security from Nanjing University of Posts and Telecommunications. His research interests include cryptographic engineering, public key cryptography and applied cryptography.

Zhenjiang Dong received the Ph.D. degree from Shanghai Jiao Tong University, Shanghai, China. He is currently a full professor at School of Computer Science in in Nanjing University of Posts and Telecommunications. His current research interests are mainly in security of data element flows, privacy computing, cyberspace security and blockchain.

Wangchen Dai received the Ph.D. degree in electronic engineering from the City University of Hong Kong in 2018. After that, he had appointments at Hardware Security Lab, Huawei Technologies Company Ltd., and the Department of CSSE, Shenzhen University. He is currently working as a Senior Researcher with Zhejiang Lab, Hangzhou, China. His research interests include cryptographic hardware and embedded systems, fully homomorphic encryption, and reconfigurable computing.

Jingqiang Lin School of Cyber Security, University of Science and Technology of China, Hefei, China. Jingqiang Lin (Senior Member, IEEE) received the M.S. and Ph.D. degrees from the University of Chinese Academy of Sciences, in 2004 and 2009, respectively. He is a Full Professor with the School of Cyber Security, University of Science and Technology of China. His research interests include applied cryptography and system security.

Fu Xiao received his Ph.D. degree in computer science and technology from the Nanjing University of Science and Technology, Nanjing, in 2007. He is currently a professor and a Ph.D. supervisor with the School of Computer Science and Technology, Nanjing University of Posts and Telecommunications, Nanjing. His research interest includes wireless sensor networks.