2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00091

WarpDrive: GPU-Based Fully Homomorphic Encryption Acceleration Leveraging Tensor and CUDA Cores

Guang Fan*, Mingzhe Zhang*, Fangyu Zheng†, Shengyu Fan‡, Tian Zhou†, Xianglong Deng‡, Wenxu Tang†, Liang Kong*, Yixuan Song*, and Shoumeng Yan*
*Ant Group, Hangzhou, China.
†School of Cryptology, University of Chinese Academy of Sciences, Beijing, China.
‡Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China.
{fanguang.fg, kongliang.kong, songyixuan.syx, shoumeng.ysm}@antgroup.com, smartzmz@gmail.com, zhengfy1028@hotmail.com, {fanshengyu, dengxianglong}@iie.ac.cn, weekdayzt@gmail.com, vx.tang@qq.com

***Abstract*—The application of Fully Homomorphic Encryption (FHE) is rapidly gaining traction as a means to maintain data confidentiality while performing computations on encrypted data. Given the accessibility and computational power, GPUs hold promise for significantly accelerating FHE operations. However, existing GPU-based acceleration solutions face several formidable challenges, notably the extensive occurrence of pipeline stalls induced by memory access and suboptimal harnessing of GPU hardware.**

This paper presents WarpDrive, a comprehensive framework for GPU-based FHE acceleration. Through sophisticated computation decomposition and fine-grained memory access design, WarpDrive significantly reduces the number of instructions by 73% and pipeline stalls by 86% compared to the state-of-the-art solution. Additionally, WarpDrive features a framework that supports the concurrent utilization of CUDA Cores and Tensor Cores within the NTT operation, for the first time, achieving performance that surpasses that of any single type of processing unit. Furthermore, we fully exploit the intra-ciphertext parallelism to elevate both computation and memory utilization, achieving up to $2.12\times$ improvements without the need for ciphertext batching. Experimental results demonstrate that our optimizations highly enhance the performance of homomorphic operations. On an NVIDIA A100 GPU, WarpDrive achieves a throughput of 1218 KOPS for NTT and 305 KOPS for homomorphic multiplication, outperforming the state-of-the-art GPU solution (TensorFHE) by factors of $13.4\times$ and $3.5\times$, respectively. For the specific FHE workload, even under a much smaller batch size, our approach achieves $2.8\times$ the performance of TensorFHE.

I. Introduction

Fully Homomorphic Encryption (FHE) allows computations to be performed directly on encrypted data without the need for decryption, ensuring that the data remains confidential throughout the computation process. Due to this powerful feature, FHE is particularly beneficial for privacy-sensitive applications, such as cloud computing, secure data analysis, and machine learning.

FHE technology has seen significant advancements in the past decade. Second-generation solutions like BGV [13] and BFV [21] have significantly improved computational efficiency and noise management. Third-generation schemes, including FHEW [19] and TFHE [16], further enhance bootstrapping efficiency but do not support batching. Notably, CKKS [15], referred to as a fourth-generation solution, is particularly effective in privacy-preserving applications, such as machine learning and data analysis, due to its remarkable advancements in handling floating-point and approximate numerical computations.

Despite numerous optimization schemes continuously emerging for homomorphic encryption, FHE still faces performance issues, typically being more than four orders of magnitude slower compared to plaintext computations. To address this bottleneck, many researchers have explored hardware acceleration methods for FHE. For example, [6], [50], [56], [60] investigate FPGA-based acceleration techniques, while [33], [34], [51], [52] demonstrate the acceleration effects using ASICs.

As a commonly used high-performance commodity computing device, GPU is an excellent choice for accelerating FHE. GPUs are characterized by their accessibility, ease of deployment, and powerful computational performance due to their abundant computing resources. Previous studies on GPU acceleration have shown promising results. For example, [28] optimizes memory access through kernel fusion in homomorphic operations and reordered bootstrapping. Phantom [55] enhances the hybrid key-switching technique [26], [31] on GPUs and further experiments with kernel fusion. TensorFHE [28] leverages Tensor Cores to accelerate the Number Theoretic Transform (NTT), the most time-consuming part of FHE, and introduces kernel-level batching to maximize data parallelism. Numerous other studies and libraries have also explored GPU acceleration to enhance the efficiency of FHE operations [2], [3], [7], [18], [23], [24], [37], [40], [57]. These works highlight the potential of GPU acceleration and lay the groundwork for further advancements.

However, several unexplored areas remain in GPU-based FHE acceleration.

Mingzhe Zhang and Shoumeng Yan are corresponding authors.

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00091

- Firstly, compared with deep optimizations related to computational units, existing implementations exhibit very limited memory optimization. Through comprehensive analysis (§III-A), we found that despite using advanced computational units and parallel strategies, existing Tensor Core-based methods do not meet performance expectations. Numerous stalls hinder Tensor Cores from realizing their immense computational power and potential, indicating a significant need for specialized deep memory optimization for Tensor Core-based FHE.
- Secondly, current solutions often rely on a single type of computational unit (CUDA Cores or Tensor Cores), even though these units are separate within the GPU and can be used concurrently. Although some studies in other fields, such as Tacker [58], attempt this, the additional inter-thread communication and precision splitting required for FHE significantly increase the complexity of using such techniques, resulting in suboptimal resource utilization (§III-B).
- Thirdly, we found that previous works achieved performance advantages by feeding GPUs with a large number of concurrent ciphertexts to maximize resource utilization (§III-C). However, schemes like CKKS already support single-plaintext SIMD operations, making it impractical to use multiple ciphertexts in real-world scenarios. Under single-ciphertext conditions, these approaches show low hardware utilization. Furthermore, this ciphertext batching method exacerbates the memory resource constraints, which are already a significant issue in FHE applications.

Based on the above observations, this paper aims to optimize memory access issues in existing methods, explore the concurrent use of Tensor and CUDA Cores in FHE schemes, and improve GPU utilization without increasing memory footprint, to better harness GPU hardware resources. The CKKS scheme [15], known for its high throughput among FHE schemes, is the main focus of this study. Our contributions can be summarized as follows:

- We propose an efficient NTT method leveraging Tensor Cores with a sophisticated computation decomposition design and a fine-grained memory access strategy. This approach reduces the number of instructions by 73% and pipeline stalls by 86% compared to the state-of-the-art Tensor Core-based method.
- We propose a unified framework that enables the Tensor Core-based NTT approach and two CUDA Core-based NTT approaches to work in parallel, maximizing the utilization of computational resources from both Tensor and CUDA Cores. Remarkably, WarpDrive demonstrates for the first time that the concurrent usage of Tensor and CUDA Cores in NTT can surpass the performance of using any single type of processing unit.
- We propose a parallelism-enhanced kernel design that fully exploits intra-ciphertext parallelism, allowing the kernel to process all polynomials within a single ciphertext concurrently, achieving high performance without the need for multiple ciphertext inputs. Compared to previous work [28] that utilizes kernel fusion, our approach achieves up to a 1.87× increase in computation throughput utilization and a 2.12× increase in memory throughput utilization.

Taking the CKKS scheme as a case study, we conduct a comprehensive evaluation of WarpDrive and compare it with previous research. In experiments using the NVIDIA A100 platform, our scheme achieves NTT and homomorphic multiplication throughputs of 1218 KOPS and 305 KOPS, reaching $13.4\times$ and $3.5\times$ of the state-of-the-art solution. For specific FHE workloads, even under a much smaller batch size, our approach achieves $2.8\times$ the performance of the state-of-the-art solution.

II. BACKGROUND

In this section, we first discuss the CKKS scheme, followed by an introduction to the Number Theoretic Transform (NTT). Additionally, we provide a brief overview of the GPU architecture. Key symbols and notations used throughout the paper are defined in Table I.

A. CKKS Scheme

CKKS, named after its authors Cheon, Kim, Kim, and Song, is a fully homomorphic encryption scheme designed for approximate arithmetic on encrypted data, suitable for real-number computations in applications like machine learning [15]. We succinctly describe CKKS and its core operations.

In CKKS, messages are represented as vectors containing $N/2$ real or complex numbers. To prepare these messages for encryption, the encoding process involves transforming a vector into a polynomial $m(X) \in \mathcal{R}_{Q_L}$, where $\mathcal{R}_{Q_L}$ denotes the ring of polynomials with coefficients in the ring of integers modulo Q_L, which is the modulus of level L. Subsequently, the encoded polynomial is encrypted, producing a ciphertext at level L, denoted as $ct = (a(X), b(X))$.

Key-switching (`KeySwitch`) is a crucial technique in FHE schemes, including CKKS, facilitating ciphertext operations under different keys. It allows the conversion of ciphertexts from one key to another without decryption, essential for multi-key computations and operations like `HMULT` and `HROTATE`, which require relinearization.

For ciphertexts at level ℓ, $\mathrm{ct}_0 = (a_0(X), b_0(X))$ and $\mathrm{ct}_1 = (a_1(X), b_1(X))$ in $R^2_{Q_\ell}$, the following are additional common homomorphic evaluation operations in CKKS:

• Homomorphic Addition (`HADD`): Adds two ciphertexts as $\texttt{HADD}(\mathrm{ct}_0, \mathrm{ct}_1) = [a_0(X) + a_1(X), b_0(X) + b_1(X)]_{Q_\ell}$.

• Homomorphic Multiplication (`HMULT`): Multiplies two ciphertexts and uses key switching to relinearize the result as $\texttt{HMULT}(\mathrm{ct}_0, \mathrm{ct}_1, \mathbf{evk}) = [(a_0(X) \cdot a_1(X), a_0(X) \cdot b_1(X) + a_1(X) \cdot b_0(X)) + \texttt{KeySwitch}(b_0(X) \cdot b_1(X), \mathbf{evk})]_{Q_\ell}$.

• Plaintext Multiplication (`PMULT`): Multiplies a plaintext with a ciphertext as $\texttt{PMULT}(\mathrm{m}, \mathrm{ct}) = [(a_0(X) \cdot a_1(X), a_0(X) \cdot b_1(X))]_{Q_\ell}$.

• Homomorphic Rotation (`HROTATE`): Rotates the message within a ciphertext by rn as $\texttt{HROTATE}(\mathrm{ct}_0, rn, \mathbf{evk}) = [(0, \mathrm{rotate}(b_0(X), rn) + \texttt{KeySwitch}(\mathrm{rotate}(a_0(X), \mathbf{evk})]_{Q_\ell}$.

TABLE I: Symbols and Notions Used in This Paper.

Symbol	Definition
$L(l)$	Maximum (current) multiplicative level of a ciphertext.
$dnum$	Decomposition number [26].
K	Number of special prime moduli.
N	Degree of a polynomial.
Q_L	(Prime) moduli product $\prod_{i=0}^{L} q_i$.
P_K	Special (prime) moduli product $\prod_{k=0}^{K} p_k$.
q_l	(Prime) moduli, where $0 \leq l < L$.
p_k	Special (prime) moduli, where $0 \leq k \leq K$.
ω, ω^{-1}	Root of unity of twiddle factor for (I)NTT.

• Rescaling (RSCALE): Reduces the modulus level of a ciphertext as $\texttt{RSCALE}(\mathrm{ct}_0) = (a'_0(X), b'_0(X)) \in R^2_{Q_{\ell-1}}$.

B. Number Theoretic Transform

1) NTT Fundamentals: The Number Theoretic Transform (NTT) plays a crucial role in the acceleration of FHE. FHE schemes require a vast number of polynomial multiplications, and NTT is instrumental in optimizing these operations. Essentially, NTT is used to convert polynomial multiplication into efficient point-wise multiplication, thereby significantly enhancing computational efficiency. The forward NTT is defined as:

$$X(k) = \sum_{j=0}^{N-1} x(j) \cdot \omega^{jk} \mod q \tag{1}$$

In this equation, $x(j)$ represents the coefficients of the input polynomial. The symbol ω is a primitive N-th root of unity in the finite field. The modulus q is a prime number that defines the finite field in which computations are performed, ensuring that all operations remain within this field.

The inverse NTT (INTT) is used to transform the data back from the frequency domain to the coefficient domain, allowing the original polynomial to be reconstructed.

2) NTT Decomposition: To enhance performance, the NTT can be decomposed into smaller sub-transforms. One notable method is the 4-step NTT algorithm, which improves memory access patterns and increases parallelism. This algorithm decomposes an NTT of size $N = N_1 \cdot N_2$ into smaller inner NTTs of sizes N_1 and N_2 as following equation:

$$X(k_1 + k_2 \cdot N_1) = \sum_{j_2=0}^{N_2-1} \left(\sum_{j_1=0}^{N_1-1} x(j_1 + j_2 \cdot N_1) \cdot \omega_{N_1}^{j_1 k_1} \right) \cdot \omega_n^{(j_1 + j_2 \cdot N_1) k_2} \tag{2}$$

where $k_1 = 0, 1, \ldots, N_1 - 1$ and $k_2 = 0, 1, \ldots, N_2 - 1$.

The above equation reveals the four steps of the algorithm: (a) proceeding N_1 groups of N_2-point inner NTTs; (b) transposing the matrix; (c) multiplying twiddle factors; and (d) proceeding N_2 groups of N_1-point inner NTTs.

3) Tensor Core-based NTT Method: Previous work by TensorFHE [22] accelerates the NTT on GPUs using Tensor Cores by mapping the NTT to matrix-matrix multiplications (GEMM). In this method, NTT decomposition is used to reduce the size of the twiddle factor matrices. Considering the practical limitations of Tensor Core precision, the Tensor Core-based NTT also needs to incorporate bit splitting and merging. Following this approach, NTT in TensorFHE is broken down into five stages, as discussed in Section III-A.

III. Motivation

In the realm of FHE acceleration, achieving optimal performance on GPUs hinges on addressing several key computational and memory access challenges. This section explores the significant memory-related pipeline stall issues associated with the Tensor Core-based NTT solution, the unexploited potential of concurrently utilizing Tensor and CUDA Cores, and the inefficiencies in current GPU parallel schemes for homomorphic operations. By identifying these challenges and proposing opportunities, we aim to enhance the overall efficiency and performance of FHE schemes on modern GPU architectures.

A. Memory-Related Pipeline Stall Issues in the Tensor Core-Based NTT Method

According to the paper and the open-source implementation of TensorFHE [4], [22], the Tensor Core-based NTT comprises five stages, implemented on the GPU using five different kernels, as shown in Algorithm 1. Among these stages, the second and fourth stages utilize Tensor Cores for GEMM with the UINT8 data type, while the remaining three stages are executed using CUDA Cores.

We use Nsight Compute to test the number and distribution of pipeline stalls in TensorFHE implementations [4], with results shown in Table II. The breakdown of the execution times for the five stages is shown in the upper part of Figure 1. The hardware platform and software we used can be referenced in Section V-B. We selected the parameters $N = 2^{16}$ and $batch\_size = 1024$, which are achievable in typical homomorphic encryption applications. Under this set of parameters, stages 2 and 4, which utilize multiple streams, are executed serially on the GPU due to the large number of SMs used.

Experimental results reveal that pipeline stalls in each stage lead to significant instruction dispatch delays, with memory-related pipeline stalls being the primary contributors. The first stage, involving the bit splitting of input data, encountered severe *Stall LG Throttle*. This kind of stall occurs when congestion in the load/store units bottlenecks warp progress due to an excessive volume of memory instructions, reflecting the kernel's high memory-to-computation ratio at this stage. In addition, *Stall Long Scoreboard* is the second most frequent pipeline stall in the NTT operation, distributed across all five stages. It occurs when a warp is delayed waiting for long-latency memory operations, typically involving global memory (GMEM) access. This reflects the excessive off-chip memory access in the Tensor Core-based NTT.

B. Challenges in Parallel Utilization of Heterogeneous GPU Units for FHE

Previous GPU-based FHE acceleration solutions [8], [9], [17], [20], [29], [54], [55] utilize only CUDA Cores, overlooking the computational power of Tensor Cores. In contrast, TensorFHE [22] leverages Tensor Cores to achieve significant

TABLE II: Stall Cycles per Issued Instructions and the Proportion of Memory Access-Related Stalls for the Tensor-Based NTT Method.

$(N = 2^{16})$ $(batch\_size = 1024)$	Stage 1 U32ToU8	Stage 2 GEMM	Stage 3 Hada&Trans	Stage 4 GEMM	Stage 5 U8ToU32
Stall Cycles Per Issued Instruction	66.5	16×3.0	3.4	16×3.0	5.2
Total Memory-Related Pipeline Stalls *	99.5%	62.4%	54.1%	62.4%	70.2%
Stall LG Throttle	82.7%	0.5%	4.5%	0.5%	3.8%
Stall Long Scoreboard	4.6%	21.1%	43.1%	21.1%	60.7%

*: Memory access-related stalls include: *Stall LG Throttle*, *Stall Long Scoreboard*, *Stall MIO Throttle*, *Stall Short Scoreboard* , *Stall Drain* and *Stall IMC Miss*.

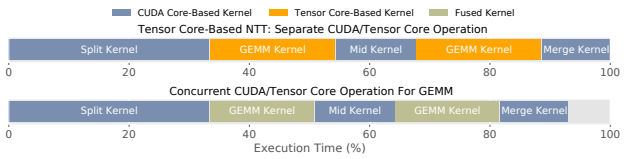

Fig. 1: Kernel Execution Timelines of Tensor Core-Based NTT and Its Naive Adaptation of CUDA/Tensor Core Concurrency

performance gains. However, no existing solution facilitates the parallel utilization of both Tensor and CUDA Cores for FHE. As illustrated in the upper part of Figure 1, TensorFHE employs CUDA Cores and Tensor Cores in separate kernels, with Tensor Cores only utilized in the GEMM kernels of stages 2 and 4. This approach clearly does not exploit the concurrent capabilities of heterogeneous processing units.

A study in another field, Tacker [58], explored the method of simultaneously using Tensor Cores and CUDA Cores in GPU Kernels, proving that both types of cores can indeed be utilized concurrently in machine learning scenarios.

Tacker is designed for matrix operations and can be used in the low-precision (UINT8) GEMM kernels of the Tensor Core-based NTT method, as shown in the lower part of Figure 1. According to the conclusion in the paper, there is an opportunity for a notable performance improvement (about 18.6%) in this part. However, GEMM is not the main load in the Tensor Core-based NTT method, accounting for only about 41% of the overall NTT, which is less than the time overhead of the bit splitting and merging stages. Bit splitting and merging computations are entirely unnecessary for CUDA Cores, as CUDA Cores natively support 32-bit integer calculations. Therefore, directly applying the Tacker solution not only fails to address the numerous pipeline stalls caused by memory accesses in the Tensor Core-based NTT method but also results in many unnecessary computations for the CUDA Cores. For high-precision (e.g., UINT32) NTT computations, it is necessary to redesign the parallel usage scheme of heterogeneous GPU units.

C. Inefficient Parallel Schemes for Homomorphic Operations

In the homomorphic operation layer, TensorFHE [22] utilizes a batching design, whereas other efforts in accelerating FHE with GPUs [28], [55] have employed solutions targeting single ciphertexts. However, none of these approaches have adequately addressed the exploitation of intra-ciphertext parallelism. In homomorphic encryption schemes, a single ciphertext can expand to nearly 1GB of data during critical steps such as homomorphic multiplication. However, existing solutions do not achieve high GPU utilization for processing a single ciphertext, thereby missing out on potential acceleration opportunities.

TABLE III: Memory and Compute Throughput Utilization of Key Kernels in 100x's Keyswitch Operation on A100 GPU.

$(N = 2^{15},\ l = 24,\ K = 1)$	NTT	ModUP	INTT	ModDown	InProd
Memory Throughput Utilization (%)	49.1	43.0	17.6	30.9	83.4
Compute Throughput Utilization (%)	37.4	36.7	19.7	49.9	20.2
$(N = 2^{16},\ l = 34,\ K = 1)$	NTT	ModUP	INTT	ModDown	InProd
Memory Throughput Utilization (%)	58.3	57.4	24.1	37.1	83.5
Compute Throughput Utilization (%)	40	49.2	19.3	60.9	19.9

We utilize Nsight Compute to measure the memory and compute throughput utilization of the critical operation, `Keyswitch` in [28]. The tests are conducted with large polynomial dimensions and depths, with kernel fusion enabled. The results indicate that, even with kernel fusion, the GPU utilization of kernels, except for InnerProduct Kernel, does not exceed 61%. Specifically, the GPU utilization for the INTT operation is below 25%. Moreover, The current utilization rate is expected to further decrease with the introduction of powerful Tensor Cores and additional memory access optimizations. Besides, designing kernels at a lower level leads to a large number of kernels, and the frequent allocation and deallocation of kernel resources incur additional overhead.

D. Our Opportunities

• **Optimizing Tensor Core-based NTT:** Given the significant role of NTT in homomorphic encryption schemes and the challenges posed by memory-related pipeline stalls in the Tensor Core-based NTT design, there lies a compelling opportunity to optimize NTT performance further. By reducing off-chip memory access and maximizing the use of on-chip memory and registers to store intermediate data, we can minimize memory access instructions and mitigate memory-related stalls. This approach will fully harness the computational power of Tensor Cores, leading to higher performance in NTT operations.

• **Bridging Tensor and CUDA Core-based methods:** Based on our efficient Tensor Core-based NTT design, we can also propose CUDA Core-based NTT solutions with a similar structure. By tailoring the design to fit the specific characteristics of NTT in FHE Schemes, we can develop an NTT solution that leverages both Tensor and CUDA Cores. This fusion approach has the potential to outperform implementations based on a single type of processing unit, showcasing better NTT performance.

• **Enhancing Kernel Design for Homomorphic Operations:** Given the limitations observed in current approaches, there is significant potential to enhance GPU utilization by designing kernels at the ciphertext level instead of the polynomial level. By elevating the design abstraction, we can leverage higher degrees of parallelism, which in turn allows for more efficient

Algorithm 1 Kernel-Level Task Assignment for Tensor Core-based NTT

SplitKernel:
  Load X from GMEM
  Split UINT32 x_i in X into 4 UINT8 elements in X_m
  Store X_m to GMEM
GEMMKernel
  Load X_m and W_{1n}/W_{3n} from GMEM
  Perform GEMM of X_m and W_{1n} with Tensor Cores
  Store Y_{mn} to GMEM
MidKernel
  Load Y_{mn} and W_2 from GMEM
  Reassembling 16 elements in Y_{mn}
  Perform ModRedc and hadamard production with W_2
  Split back into four UINT8 elements in X_m
  Store X_m to GMEM
MergeKernel
  Load Y_{mn} from GMEM
  Reassembling 16 elements in Y_{mn} and perform ModRedc
  Store X to GMEM
HostFunction FIVE_STAGE_NTT
  $(X_1, X_2, X_3, X_4) \leftarrow$ SPILTKERNEL(X)
  for $m \leftarrow 1$ to 4; $n \leftarrow 1$ to 4 **do**
    $Y_{mn} \leftarrow$ GEMMKERNEL(X_m, W_{1n})
  $(X_1, X_2, X_3, X_4) \leftarrow$ MIDKERNEL$(\{Y_{mn} \mid 1 \leq m, n \leq 4\}, W_2)$
  for $m \leftarrow 1$ to 4; $n \leftarrow 1$ to 4 **do**
    $Y_{mn} \leftarrow$ GEMMKERNEL(X_m, W_{3n})
  $X \leftarrow$ MERGEKERNEL$(\{Y_{mn} \mid 1 \leq m, n \leq 4\})$

use of GPU compute and memory resources. This shift enables the exploitation of intra-ciphertext parallelism, thereby improving the throughput for homomorphic operations.

IV. WarpDrive Design

Building on the insights from Section III, we introduce WarpDrive, a comprehensive framework designed to efficiently and comprehensively utilize GPU resources to enhance CKKS performance on GPU platforms. WarpDrive introduces the efficient Tensor Core-Based NTT for the frequently occurring and time-consuming FHE polynomial operations, NTT, significantly reducing off-chip and on-chip memory accesses and substantially lowering the number of operations. Additionally, WarpDrive presents a method for the parallel utilization of Tensor and CUDA cores adapted for NTT, enabling the concurrent use of temporarily idle CUDA cores. For homomorphic operations, WarpDrive offers a parallelism-enhanced kernel design, leveraging inherent parallelism within a single ciphertext to improve hardware utilization. In the remainder of this section, we provide a detailed explanation of the optimization techniques employed in WarpDrive.

A. Efficient Tensor Core-Based NTT

1) Warp-Level Method for Tensor Cores: In the previous Tensor Core-based NTT solution [22], kernel-level task assignments are employed, as illustrated in Algorithm 1. This approach allows the use of existing GPU-based linear algebra libraries, such as CUTLASS and CUBLAS. However, as analyzed in Section III-A, this method suffers from severe memory access stalls. To address this issue, we propose the warp-level task assignment for Tensor Core-based NTT, as shown in Algorithm 2. This method invokes Tensor Cores at

Algorithm 2 Warp-Level Task Assignment for Tensor Core-based NTT

OneStageNTTKernel:
  Load X and W_{1n}/W_{3n} from GMEM
  Split UINT32 x_i in X into 4 UINT8 elements in X_m
  Store X_m and W_{1n}/W_{3n} to SMEM
  Load X_m and W_{1n}/W_{3n} from SMEM
  Perform GEMM of X_m and W_{1n} with Tensor Cores
  Store Y_{mn} to SMEM
  Load W_2 from GMEM
  Load Y_{mn} from SMEM
  Reassembling 16 elements in Y_{mn}
  Perform ModRedc and Hadamard Production with W_2
  Split back into four UINT8 elements in X_m
  Store X_m to SMEM
  Load X_m and W_{1n}/W_{3n} from SMEM
  Perform GEMM of X_m and W_{1n} with Tensor Cores
  Store Y_{mn} to SMEM
  Load Y_{mn} from SMEM
  Reassembling 16 elements in Y_{mn} and perform ModRedc
  Store X to GMEM
HostFunction ONE_STAGE_NTT
  ONESTAGENTTKERNEL(X)

the warp level, eliminating the need to allocate separate kernels for each GEMM computation. This can prevent the frequent on-chip/off-chip data exchanges, thereby avoiding prolonged stalls caused by memory access.

Before employing the warp-based method, it is necessary to ensure that sufficient on-chip resources are available to store data previously held in GMEM. To achieve this, a more granular decomposition of the NTT can be employed to reduce on-chip resource usage.

2) NTT-Decomposition: Tensor Cores are designed for matrix computations, requiring that operators be converted into matrix operation forms. However, the specific conversion method and the granularity of the matrices are crucial for memory operations. In the kernel-level approach, NTT-decomposition introduces additional overhead due to memory data transfers, resulting in significant costs for multiple decompositions. However, in the warp-level approach, NTT decomposition does not lead to an increase in the number of kernels, thereby creating the conditions for employing deeper NTT decompositions.

Figure 2 first illustrates the iterative decomposition method for the NTT on the left side. This two-level decomposition comprises three distinct NTT decompositions, each employing Equation 2. We consolidate the steps of the 4-step algorithm by considering steps two and three as one, resulting in each decomposition dividing one or multiple groups of NTTs into three steps. After two layers of decomposition, the original NTT is organized into seven steps. Specifically, steps 1, 3, 5, and 7 execute grouped inner NTTs, while steps 2 and 6 perform grouped matrix transposition and Hadamard products. Step 4 handles the transposition and Hadamard product of large matrices.

The right side of Figure 2 shows the step-by-step execution process from the warp perspective. After iterative decomposition, the original extensive NTT transforms into grouped smaller NTTs, which can be executed using either matrix

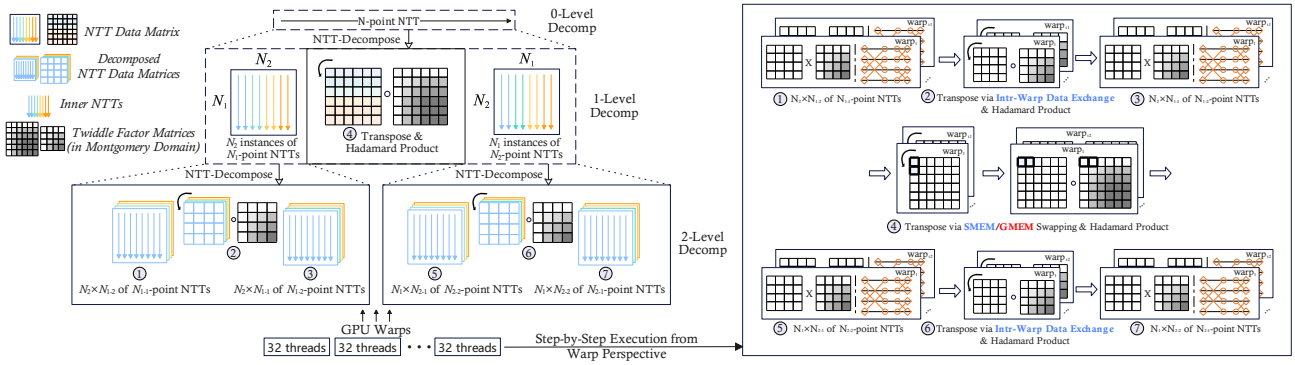

Fig. 2: NTT Decomposition in WarpDrive-NTT

multiplication or butterfly operations in steps 1, 3, 5, and 7. Regarding memory operations, steps 2 and 6 only require data exchange within the warp, with a GMEM operation potentially needed only during transposition of the large matrix in step 4.

We measure the computational cost and the size of twiddle factor matrices when using matrix multiplication to process the NTT. Table IV illustrates the trends of these parameters across different decomposition levels. Taking the 2-level decomposition in Figure 2 as an example, the entries Element-Wise Multiplication (EW-Mul), Modular Reduction (ModRed), and Bit Decomposition and Merging (Bit-Dec&Mer) in the table correspond to the computational cost required for steps 1, 3, 5, and 7. Meanwhile, Modular Multiplication (ModMul) corresponds to the cost for steps 2, 4, and 6. The Matrix Size refers to the size of the twiddle factor matrices used for matrix multiplication in steps 1, 3, 5, and 7. An exponential decrease is observed in both matrix size and EW-MUL operations with more decomposition levels, particularly evident in the substantial reductions not just from level 0 to 1, but also from level 1 to 2.

In TensorFHE, only a single level of decomposition is used to avoid introducing additional kernels and on-chip/off-chip data exchanges. In this case, the twiddle factor matrices are in the range of hundreds of KB, making them difficult to fit into on-chip shared memory (SMEM), and the overall number of computational operations remains relatively high. In contrast, WarpDrive employs a sophisticated decomposition, which not only reduces the size of the twiddle factor matrices to fit within the capacity of SMEM but also decreases the number of multiplications in the matrix multiplication operations, the core computation load in Tensor Core-based NTT method, to one-eighth of the original amount.

For larger values of N, such as $N = 65536$, using two levels of decomposition allows us to control the dimensions of the inner NTT to be within 16. For smaller values of N, we can reduce the number of decompositions at the second level, performing only one decomposition at that level. For instance, for $N = 4096$, we decompose it into the form $(16 \times 16) \times 16$. Due to the fact that deeper levels of decomposition result in matrix multiplication dimensions becoming too small, leading to underutilization of Tensor Core performance and an increased computational load on the CUDA Cores, we did not further increase the number of decomposition levels.

3) Further reduction of memory access: The Warp-level Tensor Core-based NTT solution replaces GMEM accesses with SMEM accesses. Building on this foundation, we further explore methods to reduce SMEM accesses by utilizing registers.

Tensor Cores differ from ordinary CUDA Cores in that an entire warp is scheduled onto a Tensor Core and executes uniformly. Although NVIDIA has not disclosed the specific tasks assigned to each thread within GEMM operations, reverse engineering results from [59] indicate that each thread is allocated registers to handle fixed positions within the matrix.

Utilizing this information, we can allocate data for specific positions in the matrix to each thread within a warp. While Tensor Cores still execute at the granularity of a warp, it is no longer necessary to read or write data to memory. Instead, the allocated registers for each corresponding thread can be used to store intermediate results, reducing the stalls caused by on-chip memory accesses.

Furthermore, for unavoidable off-chip memory accesses, we employ coalesced memory accesses to reduce the number of memory transactions.

4) The selection of optimization algorithms: **Modular reduction (ModRedc)** is a crucial operation in FHE schemes. In WarpDrive, we employ Montgomery ModRedc [36] and Barrett ModRedc [11] to optimize modular arithmetic. In the NTT, the twiddle factors can be pre-converted to the Montgomery domain, addressing the need for pre-computation or post-computation in the Montgomery algorithm. Moreover, Montgomery ModRedc offers approximately a 10% performance improvement over Barrett ModRedc. Therefore, we uniformly use Montgomery ModRedc in the NTT, while employing Barrett ModRedc in other computations.

Karatsuba multiplication [30] can be utilized to reduce the number of operations required for multi-precision multiplication and can be applied in computing multi-precision matrix multiplication using Tensor Cores. However, the Karatsuba algorithm presents a trade-off. We test a 4-term Karatsuba algorithm within a Tensor Core-based NTT framework. This approach decreases the number of multiplications from 16 to

TABLE IV: The Number of Different Operations in the NTT Methods with Varying Decomposition Layers.

	Matrix Size	EW-Mul	ModRed	ModMul	Bit-Dec&Mer
l-Level Decomp	$N^{\frac{1}{2^{l-1}}}$	$N \cdot N^{\frac{1}{2^l}} \cdot 2^l$	$N \cdot 2^l$	$(2^l-1)N$	$(2^{l+1}-2)N$
0-Level Decomp (N = 65536)	2^{32}	2^{32}	2^{17}	2^{16}	2^{17}
1-Level Decomp (N = 65536)	2^{16}	2^{25}	2^{17}	2^{16}	2^{17}
2-Level Decomp (N = 65536)	2^{8}	2^{22}	2^{18}	3×2^{16}	3×2^{17}
3-Level Decomp (N = 65536)	2^{4}	2^{21}	2^{19}	7×2^{16}	7×2^{17}

9, but introduces 5 additional additions during preprocessing, and reduces the effective word length by 2 bits. Despite these modifications, we observe no significant performance improvement in our NTT implementation. Consequently, we ultimately choose not to adopt Karatsuba multiplication.

B. Bridging Tensor and CUDA Core-based Methods

In the warp-level method for NTT, we integrate computations utilizing Tensor Cores and CUDA Cores into a single kernel. However, this integration alone is not sufficient to achieve concurrent execution of Tensor and CUDA Cores.

1) Framework Setup: To enable concurrent execution of Tensor and CUDA Cores, it is necessary for each to handle a segment of computationally parallel tasks simultaneously. In Tacker [58], this is achieved by allowing Tensor Cores and CUDA Cores to concurrently perform GEMM operations. In WarpDrive, we integrate Tensor and CUDA-based methods into a unified NTT framework, WarpDrive-NTT, that employs Warp-level Task Assignment and sophisticated NTT Decomposition. The grouped inner NTTs, which are inherently parallelizable, are allocated to Tensor and CUDA Cores for concurrent computation.

2) CUDA Core-Based NTT Design: First, WarpDrive supports a method similar to that in Tacker, allowing Tensor and CUDA Cores to perform the same GEMM operations. However, we do not require CUDA Cores to perform low-precision GEMM as in Tacker. Instead, we fully leverage the word length of CUDA Cores to directly compute high-precision (32-bit) GEMM. By leveraging Warp-level Task Assignment, both bit splitting and GEMM computations are executed within the same kernel. This allows CUDA Cores to bypass bit operations and directly perform GEMM computations.

Besides, for 32-bit GEMM, the use of Tensor Cores necessitates bit splitting and merging performed by CUDA Cores. This increases the computational load on CUDA Cores. To reduce CUDA Cores' workload, WarpDrive also supports using butterfly operations to perform inner NTTs with CUDA cores, instead of using GEMM. The differentiated computation for CUDA and Tensor Cores also ensures that the time windows for reading twiddle factors by them do not completely overlap, thereby enlarging the overall read-write time window and reducing memory access pressure under time-sharing conditions. In the computational segment of the butterfly operations, we utilize high-radix NTTs with registers to store intermediate variables, minimizing the memory stalls caused by Read-After-Write (RAW) dependencies mentioned in [22]. The radix value is ideally set to 16, the same as that of Tensor Cores, while also supporting radix-8 and radix-4 to accommodate different values of N.

3) Warp Allocation: In the CUDA model, assigning different blocks within the same kernel to a fixed Streaming Multiprocessor (SM), on the GPU is subject to various constraints related to hardware architecture, resources, and software compilation, making it nearly impossible to achieve stably. This becomes even more difficult when considering blocks from different kernels. However, within a single block, all warps are allocated to the processing units of the same SM. Therefore, to concurrently utilize Tensor and CUDA Cores, it is necessary to ensure that the warps using Tensor Cores and warps using CUDA Cores within a block cover all the Streaming Processors (SPs) within SM. This ensures that each SP has both Tensor Core-based tasks and CUDA Core-based tasks, making it possible for these two types of computations to overlap on SPs. Meanwhile, due to the inter-thread communication requirements of NTT, the data processed by a warp needs to be placed in SMEM. Consequently, owing to the capacity limitations of SMEM, we cannot employ an excessive number of warps within a single block.

Figure 3 illustrates the warp allocation in WarpDrive-NTT. In each block of the NTT kernel, we use 4 warps to compute the CUDA Core-based NTT and 4 warps to support the Tensor Core-based NTT, which exactly covers all the SPs within an SM in the latest GPU architectures [41], [42], [44].

C. Parallelism-Enhanced Kernel Design for FHE Operations

As discussed in Sec. III-C, previous GPU-accelerated solutions design GPU kernels at the polynomial level, leading to low hardware utilization or high batching requirements. Therefore, we propose a Parallelism-Enhanced Kernel (PE Kernel) design to fully exploit the parallelism within ciphertexts.

1) Intra-ciphertext parallelism: In FHE, ciphertexts inherently possess a multi-dimensional structure. First, the ciphertext contains multiple polynomials. Operations such as homomorphic addition and plaintext multiplication involve two polynomials, $a(X)$ and $b(X)$; while `Keyswitch` expands the polynomials into $dnum$ new polynomials. Besides, each polynomial has a dimension N, and using Residue Number System (RNS) decomposition introduces an additional dimension L.

Existing solutions [28], [55] use polynomials as the inputs for GPU kernels, thereby leveraging the parallelism in the N and L dimensions. Additionally, TensorFHE [22] introduces an extra dimension of parallelism through batching. However,

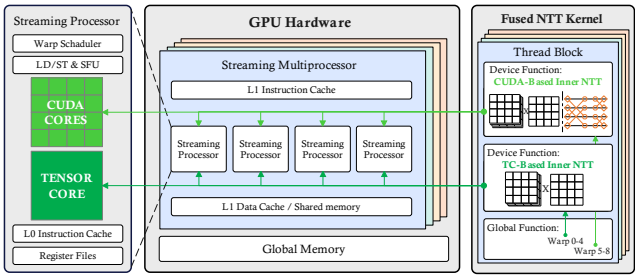

Fig. 3: Concurrent Utilization of CUDA and Tensor Cores

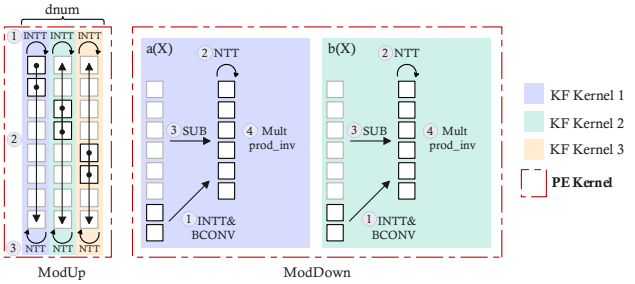

Fig. 4: The Enhanced Parallelism Effect of The PE Kernel Compared to The KF Kernel, Exemplified by ModUp and ModDown in Keyswitch.

the logic for processing multiple polynomials within the ciphertext is handled on the host side, which fails to exploit the parallelism of this dimension on the GPU.

2) Expanding Input Dimensions: WarpDrive initially supports the use of multiple streams to achieve logical parallel processing of multiple polynomials. However, multiple streams utilize multiple GPU contexts, and this approach to hardware-level parallelism is subject to many limitations. Significant parallel effects are only observed when the number of blocks used is very low, leading to poor actual parallel performance in homomorphic encryption acceleration

To fully leverage GPU resources, WarpDrive introduces an additional input dimension in the kernels for homomorphic operations. This enables a single kernel to process an entire ciphertext containing multiple polynomials. WarpDrive incorporates logic for handling multiple polynomials within both the kernel configuration and the kernel itself.

Figure 4 illustrates the parallelism-enhanced effect in WarpDrive, taking ModUp and ModDown in `Keyswitch` as examples. In an ideal kernel fusion scenario, the processing for a single polynomial in ModUp and ModDown can each be executed using a single kernel. However, the kernel-fused kernel (KF Kernel) still cannot handle multiple polynomials within a single kernel. In contrast, our parallelism-enhanced kernel (PE Kernel) can fully exploit the parallelism of multiple polynomials within the GPU.

Compared to directly processing ciphertexts in batches, this approach reduces the pressure on memory capacity and the limitations in application scenarios. Furthermore, this method can be combined with ciphertext batching to reduce the demand for batch sizes.

D. WarpDrive Framework

The WarpDrive framework is designed for the efficient execution of FHE operations, specifically CKKS, on GPUs. It automates parameter configuration based on encryption parameters and GPU characteristics, managing the GPU computation lifecycle efficiently.

1) Initialization Phase: During the Initialization Phase, WarpDrive prepares for computation by selecting and generating moduli and precomputed values such as twiddle factors used in the Number Theoretic Transform (NTT). These precomputed data are transferred to the GPU in advance to reduce runtime memory copy overhead. Furthermore, WarpDrive initializes a memory pool to maximize GPU resource efficiency. The maximum ciphertext size S_{max} during computation is determined by: $S_{max} = l \times N \times dnum \times (l+k) \times BS \times w$, where BS is the batch size, and w is the word size. The framework allocates the memory pool by taking the smaller value between S_{max} and the currently available memory.

2) Kernel and Thread Configuration: Kernel selection for the NTT phase depends on N and shared memory size (S_{shared}). If a single block can accommodate the entire polynomial data ($N \times w \leq S_{shared}$), a single-kernel approach is used; otherwise, a dual-kernel implementation is employed. Threads per block (T) are set to the optimal block size, determined as $T = C \times W \times 32$, where C is the SP count per SM, W is the number of warps allocated per SP, typically set to 2 by default, and 32 is the thread number of a warp. The number of blocks B is calculated as: $B = \left\lceil \frac{N_c}{T \times N_t} \right\rceil$, where N_c represents the number of coefficients in a ciphertext. The number of coefficients allocated to each thread, denoted as N_t, is set to 8 for the NTT due to the processing scale of Tensor Cores, and 1 for other element-wise operations to maximize parallelism. This configuration ensures efficient workload distribution across warps.

3) Core Utilization Optimization: WarpDrive optimizes the warp number balance between Tensor Cores and CUDA Cores. When both Tensor Cores and CUDA Cores are present within an SP, WarpDrive dynamically allocates the number of warps to each type of cores. The ratio of warps assigned to Tensor Cores versus CUDA Cores is configured based on their respective computational power. This strategy ensures that computational throughput is maximized by effectively leveraging the strengths of each core type.

V. Evaluation

In this section, we present the implementation and evaluation of the proposed WarpDrive framework. We assess its performance across three levels: NTT, homomorphic operations, and FHE workloads, and compare it against existing solutions on various hardware platforms and parameter sets.

A. Implementation of WarpDrive

To evaluate the proposed methods, we implement NTT and the CKKS scheme from [12], [26] using the WarpDrive framework. Our implementation employs a 32-bit word size

TABLE V: Compared Schemes and Corresponding Hardware Platforms.

Scheme	Type	Platform	Clock
Baseline [49]	CPU	Xeon(R) Silver 4108	1.80 GHz
GME-base [53]	GPU	AMD MI100	1.50 GHz
TensorFHE [22]	GPU	NVIDIA A100-SMX-40G	1.41 GHz
Cheddar [32]	GPU	NVIDIA A100-PCIE-40G	1.41 GHz
Liberate.FHE [18]	GPU	NVIDIA A100-PCIE-80G	1.41 GHz
100x [1] *	GPU	NVIDIA A100-PCIE-80G	1.41 GHz
WarpDrive	GPU	NVIDIA A100-PCIE-80G	1.41 GHz

*: We use the open-source implementation [1] to reproduce the homomorphic operations in 100x [28] on an NVIDIA A100-PCIE-80G GPU.

TABLE VI: HE Parameter Sets Used in the Evaluation of NTT and Homomorphic Operations.

	Set-A	Set-B	Set-C	Set-D	Set-E
N	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
l	2	6	14	24	34
$\log qp$	108	217	437	704	974

for computation, which is natively supported by INT32 CUDA Cores, but requires bit-splitting for Tensor Cores. We also incorporate the double-prime rescaling [5], [33].

For NTT, we first implement the efficient Tensor Core-based NTT (WD-Tensor) by leveraging WMMA (Warp Matrix Multiply-Accumulate) instructions [45] to perform fine-grained Tensor Core operations. Furthermore, we develop four additional NTT variants for comparative analysis: WD-CUDA, which employs CUDA Cores instead of Tensor Cores for matrix multiplication; WD-FTC, which fuses WD-Tensor and WD-CUDA kernels; WD-BO, which utilizes butterfly operations for inner NTTs; and WD-FUSE, which fuses WD-Tensor and WD-BO kernels.

For homomorphic operations, we adopt the kernel fusion approach from [28] and further integrate PE Kernel Design, with `Keyswitch` supporting different dnum parameter settings. For bootstrapping, we adopt the method in [26] and utilize slim bootstrapping in [14].

B. Experiment Setup

The evaluations are conducted on a platform comprising an NVIDIA A100 GPU (1.41 GHz) and a Hygon C86 7265 CPU (2.20 GHz). The software environment included CUDA 11.3 [43] and GCC 10.2.1 [48]. The default number of threads per block for the kernel was set to 256. Performance metrics such as executed instructions, execution cycles, various stalls, and computation and memory throughput utilization are collected using NVIDIA NSIGHT Compute [46].

The schemes and hardware involved in comparisons are shown in Table V. Note that the NVIDIA A100-SMX-40G used in TensorFHE has the same architecture and clock frequency as the A100-PCIE-80G used in our experiments. The evaluation of 100x [28] is conducted using an NVIDIA Tesla V100 GPU. We replicate the homomorphic operations based on the open-source code [1] and descriptions from the paper [28] and test the results on the A100-PCIE-80G GPU.

Table VI presents a subset of the parameters utilized in our evaluations. These parameters use $k = 1$. The total number of primes is $l + 2$, which includes one base prime and one special prime. Notably, for the SET-D and SET-E parameter sets, we do not use the maximum level allowed by the security modulus bit limit. Instead, we employ a level that lies between the maximum and half the maximum. This configuration better reflects the average performance of operations at different levels in practical applications.

TABLE VII: Throughput Comparison of NTT and INTT Operations for CPU, TensorFHE and WarpDrive (KOPS).

Operation	Scheme	SET-A	SET-B	SET-C	SET-D	SET-E
NTT	CPU Baseline [49]	7.2	3.4	1.6	-	-
	TensorFHE [22] *	910	450	209	98.9	48.3
	WarpDrive	12181	4675	2088	1009	468
	Speedup Over CPU	1692×	1375×	1305×	-	-
	Speedup Over [22]	**13.4**×	**10.4**×	**10.0**×	**10.2**×	**9.7**×
INTT	CPU Baseline [49]	7.6	3.5	1.7	-	-
	TensorFHE [22] *	913	449	209 W	98.8	48.1
	WarpDrive	12097	4643	2073	1009	465
	Speedup Over CPU	1592×	1327×	1223×	-	-
	Speedup Over [22]	**13.2**×	**9.9**×	**10.0**×	**10.1**×	**9.7**×

*: TensorFHE performance data corresponding to SET-A, SET-B, and SET-C are from [22]. SET-D and SET-E are tested on NVIDIA-PCIE-A100 using its open-source NTT implementation [4].

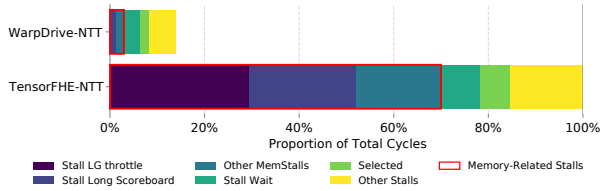

Fig. 5: Scheduler Cycles Breakdown in TensorFHE-NTT and WarpDrive-NTT (WD-Tensor Variant) ($N = 2^{16}, batch\_size = 1024$).

C. Evaluation of WarpDrive-NTT

Table VII presents a comparison of NTT throughput in KOPS (Kilo Operations Per Second) between WarpDrive and TensorFHE [22]. WarpDrive-NTT demonstrates a significant advantage, achieving throughput improvements ranging from 13.4× for SET-A to 9.7× for SET-D compared to TensorFHE. Moreover, these values are 1692×, 1375×, and 1305× higher than the CPU Baseline in [49] for the same parameter sets. Note that the acceleration of WarpDrive-NTT is effective across all parameters, including large ones.

To further illustrate the optimization effects of NTT in WarpDrive, we conducted a detailed comparison of pipeline stalls between the WD-Tensor variant in WarpDrive and TensorFHE-NTT for a polynomial degree of 2^{16}. The experiments employed a batch size of 1024, ensuring uniform task distribution across all schedulers.

The results are shown in Figure 5. Overall, WarpDrive-NTT reduces the cycle count by 86.0% compared to TensorFHE-NTT. In WarpDrive-NTT, memory-related stall cycles comprise only 21.2% of the total cycles, a substantial reduction from the 70% observed in TensorFHE-NTT. Specifically, WarpDrive-NTT almost entirely eliminates the highest proportion stall in TensorFHE-NTT, *Stall LG throttle*, by removing segments with extreme compute-storage imbalance, such as the bit splitting dedicated kernel in TensorFHE-NTT. Secondly, WarpDrive-NTT reduces the number of *Stall*

TABLE VIII: Latency Comparison of Key Operations between WarpDrive and Previous Acceleration Studies (μs).

Operation	Scheme	SET-C	SET-D	SET-E
HMULT	Liberate.FHE [18]	6185	9543	25673
	TensorFHE_repl ‡	847	2893	10986
	100x_fused [1]	595	1734	5971
	100x_opt *	504	1642	5571
	WarpDrive	277	1089	4284
	Speedup †	**1.82**×	**1.51**×	**1.30**×
HROTATE	Liberate.FHE [18]	5832	9164	25263
	TensorFHE_repl ‡	838	2876	11030
	100x_fused [1]	579	1693	5871
	100x_opt *	512	1667	5659
	WarpDrive	273	1095	4341
	Speedup †	**1.88**×	**1.52**×	**1.30**×
RESCALE	Liberate.FHE [18]	572	625	790
	TensorFHE_repl ‡	149	355	759
	100x_fused [1]	107	185	406
	100x_opt *	87	181	396
	WarpDrive	45	100	241
	Speedup †	**1.93**×	**1.81**×	**1.64**×
HADD	Liberate.FHE [18]	62	64	66
	TensorFHE_repl ‡	5.2	11	61
	100x_fused [1]	13	22	82
	100x_opt *	12	21	81.5
	WarpDrive	5.2	11	61
	Speedup †	**2.31**×	**1.91**×	**1.34**×

*: 100x_opt refers to the implementation that replaces the relevant functions in 100x (fused) [1] with the same NTT and 32-bit modular arithmetic as in WarpDrive.
‡: TensorFHE_repl refers to the implementation we replicated using TensorFHE's NTT [4] and WarpDrive's homomorphic operation implementation.
†: Speedup indicates the performance improvement of WarpDrive compared to 100x_opt.

Long Scoreboard cycles by 98% through minimizing off-chip memory access at both ends of the stages in TensorFHE-NTT.

Additionally, the *Selected* metric reflects the number of instructions in the NTT Kernels. Through sophisticated computational decomposition and fine-grained memory access optimizations, WarpDrive-NTT reduces the overall number of instructions by 73%, which in turn leads to varying degrees of reduction in other stall events.

In summary, WarpDrive-NTT significantly improves the computational efficiency of Tensor Core-based NTT by reducing both memory-related stall cycles and the number of instructions, resulting in substantial performance gains.

D. Evaluation of Concurrent for Tensor and CUDA Cores

In this subsection, we evaluate five different NTT implementation variants introduced in section V-A.

The throughput comparison results are illustrated in Figure 6. The Tensor Core-based approach demonstrates a performance advantage of 12% to 28% and 4% to 10% compared to WD-CUDA and WD-BO, respectively, highlighting the computational power of Tensor Cores. Both fusion approaches, WD-FTC and WD-FUSE, achieve higher throughput than their respective CUDA Core-based implementations, demonstrating the performance benefits of combining both types of units. Specifically, in the WD-FTC approach, the CUDA Core component executes GEMM computations, which impose a high computational load on the CUDA Cores, resulting in an overall performance that is inferior to the WD-Tensor variant. Notably, the WD-FUSE approach surpasses any unfused approach, further enhancing throughput by 4% to 7% over the efficient Tensor Core-based NTT method. Consequently, WD-FUSE is the default scheme for WarpDrive-NTT.

TABLE IX: Comparison of Kernel Number and Compute and Memory Throughput Utilization between 100x and WarpDrive's Keyswitch Implementations.

Metric	Scheme	SET-C	SET-D	SET-E
Kernel Num	100x_opt	59	90	109
	WarpDrive	11	11	11
	Reduction	**81.4%**	**87.8%**	**90.0%**
Compute Throughput Utilization (%)	100x_opt	14.2	24.5	31.6
	WarpDrive	26.6	34.8	35.6
	Improvement	**1.87**×	**1.42**×	**1.13**×
Memory Throughput Utilization (%)	100x_opt	25.3	47.0	65.9
	WarpDrive	53.6	70.6	79.4
	Improvement	**2.12**×	**1.50**×	**1.20**×

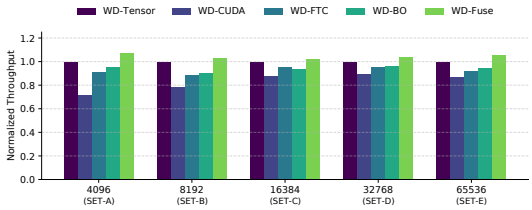

Fig. 6: Comparison of NTT Throughput with Tensor Cores, CUDA Cores, and Concurrent Utilization of Both Cores.

E. Evaluation of Homomorphic Operations with Enhanced Kernel Design

To verify the effectiveness of our parallelism-enhanced kernel design, we evaluate WarpDrive's homomorphic operations and compare them with previous studies. The results are presented in Table VIII. Compared to the 100x implementation [1], WarpDrive demonstrates performance improvements of at least 115%, 59%, and 39% for `HMULT` under the SET-C, SET-D, and SET-E parameters, respectively. Due to the absence of certain functions in the 100x implementation, we utilize functions from WarpDrive as substitutes. For a more fair comparison, we replace the NTT and 64-bit modular operations in 100x with WarpDrive-NTT and 32-bit word-length modular operations, referred to as the 100x_opt. When compared to this implementation, WarpDrive still demonstrates performance improvements of no less than 82%, 51%, and 30% under the same parameters. When compared to Liberate.FHE [18], WarpDrive offers remarkable latency reductions across various homomorphic operations, achieving speedups ranging from 1.7× to 22×. Compared to TensorFHE_repl, WarpDrive achieves speedups up to 3.3×.

To further analyze the effect of WarpDrive on enhanced parallelism, we conduct a comparison of kernel numbers, compute throughput, and memory throughput for the core operation, `Keyswitch`, between 100x_opt and WarpDrive. WarpDrive requires significantly fewer kernels across all three configurations (SET-C, SET-D, SET-E) compared to 100x_opt baseline, with only 11 kernels needed, greatly reducing the

TABLE X: Comparison of Kernel Number and Compute and Memory Throughput Utilization between TensorFHE and WarpDrive's NTT Implementations.

Metric	Scheme	SET-C	SET-D	SET-E
Compute TP Utilization (%)	TensorFHE [22]	27.0	30.0	31.8
	WarpDrive	49.6	56.8	49.1
	WarpDrive/TensorFHE	1.84×	1.89×	1.54×
Memory TP Utilization (%)	TensorFHE [22]	65.5	73.1	78.7
	WarpDrive	59.0	65.9	80.1
	WarpDrive/TensorFHE	0.90×	0.90×	1.02×

TABLE XI: Latency Comparison of Key Operations for Cheddar and WarpDrive (μs) with Parameters $N = 2^{16}$ and $\alpha = 7$.

Operation	Scheme	Full Level ($l = 27$)	Half Level ($l = 13$)
HADD	Cheddar [32]	78	32
	WarpDrive	52.1	26.3
	Speedup	**1.50**×	**1.22**×
PMULT	Cheddar [32]	62	26
	WarpDrive	45.3	19.9
	Speedup	**1.37**×	**1.31**×
HMULT	Cheddar [32]	890	395
	WarpDrive	917	386
	Speedup	**0.97**×	**1.02**×

overhead associated with kernel resource allocation and deallocation. Additionally, WarpDrive demonstrates a noticeable improvement in both compute and memory throughput utilization compared to the 100x_opt. Specifically, WarpDrive exhibits higher throughput utilization, with compute throughput exceeding that of 100x_opt by $12.7\% \sim 87.3\%$ across the three configurations, and memory throughput improving by $20.5\% \sim 111.5\%$ over 100x_opt.

We compare WarpDrive with a recent work, Cheddar [32], and find that WarpDrive shows up to a 50% improvement in `HADD` and a 37% improvement in `PMULT`, due to its emphasis on hardware utilization through kernel design. While we focus on computational units utilization, Cheddar employs a more compact 32-bit data structure. These different optimization paths lead to similar `HMULT` performance. Note that, the techniques proposed in WarpDrive are orthogonal with Cheddar, which can be integrated with Cheddar to achieve further performance improvement.

We also compare the throughput of homomorphic operations against TensorFHE [22] and a CPU baseline [49], as shown in Table XII. Under the SET-A, SET-B, and SET-C parameters used by TensorFHE, WarpDrive's homomorphic multiplication throughput shows improvements of 3.5×, 1.7×, and 1.4×, respectively, over TensorFHE, and 726×, 596×, and 260×, respectively, over the CPU benchmark. Notably, TensorFHE is optimized for operation-level batching, which enhances GPU utilization but requires the application to gather more data for parallel processing and necessitates larger GPU memory. In contrast, WarpDrive's enhanced parallelism design makes more extensive use of parallelism within a single ciphertext, without additional application and hardware requirements.

In Table X, we present a detailed comparison of the memory and compute utilization between TensorFHE and WarpDrive. WarpDrive demonstrates a notable enhancement in compute throughput utilization, achieving an increase of 1.54× to 1.89×. Simultaneously, it maintains memory utilization levels comparable to TensorFHE, with a range of 0.90× to 1.02×. This advancement can be attributed to WarpDrive's refined memory access architecture, which effectively reduces both on-chip and off-chip memory accesses. Consequently, this design minimizes memory access-induced stalls during instruction execution.

TABLE XII: Throughput Comparison of HMULT Operation for CPU, TensorFHE and WarpDrive (KOPS).

Operation	Scheme	SET-A	SET-B	SET-C
HMULT	CPU Baseline [49]	0.42	0.08	0.02
	TensorFHE [22]	88.0	27.6	3.8
	WarpDrive	304.9	47.7	5.2
	Speedup Over CPU	726×	596×	260×
	Speedup Over [22]	**3.46**×	**1.73**×	**1.37**×

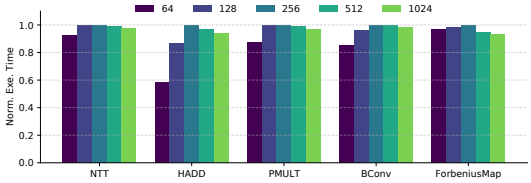

Fig. 7: Sensitivity Analysis to Threads Per Block.

F. Sensitivity Study to the Threads per Block

Figure 7 illustrates the impact of threads per block (T) on various operations. Each operation's performance is measured in terms of normalized execution time, with different colors representing varying thread counts: 64, 128, 256, 512, and 1024. The effect of threads per block differs across operations, but optimal performance is consistently achieved at $T = 256$. The experiments were conducted on SET-D, and similar results were observed with other parameters. Therefore, a T value of 256 is adopted in this study.

G. Evaluation of FHE Workloads

We evaluate three typical FHE workloads: Packed Bootstrapping (Boot) [38], Logistic Regression (HELR) [25], and ResNet-20 [35], using the parameters in Table XIII. The results are presented in Table XIV. The evaluation compares the performance of our implementation, WarpDrive, against TensorFHE [22], 100x [28], and other recent works [47], [53].

Compared to TensorFHE [22], our workloads achieved improved performance even when using a maximum level that is not lower than TensorFHE's and a much smaller batch size. Notably, our HELR implementation shows a performance improvement of 2.82× over TensorFHE. In comparisons with 100x [28] and [47], we do not use ciphertext batching for a fair assessment. Our workload performance significantly outperformed 100x [28]. When compared with [47] on the same platform, our Boot and ResNet implementations demonstrate performance advantages of 1.41× and 1.46×, respectively. Compared to GME [53], which significantly modifies the hardware architecture of MI100 GPU to create an FHE accelerator, WarpDrive is slower than this implementation. However, it is $1.70\times \sim 5.82\times$ faster than GME's baseline (software-only optimization) implementation on the MI100.

TABLE XIII: CKKS Parameters Used in the Evaluation of FHE Workloads.

Workload	N	L	K	$logPQ$
ResNet	2^{16}	37	13	1456
HELR	2^{16}	37	13	1456
Boot	2^{16}	34	12	1344
AES	2^{16}	46	10	1532

TABLE XIV: Performance Comparison of FHE Workloads. Taking Amortized Execution Time as the Metric (BS Denotes Batch Size).

Scheme (Hardware)	Boot (ms)	HELR (ms/it)	ResNet (s)
TensorFHE [22] (A100-SMX-40G)	250 (L=34, BS = 128)	220 (L=37, BS = 64)	4.94 (L=28, BS = 64)
WarpDrive (A100-PCIE-80G)	97 (L=34, BS = 16)	78 (L=37, BS = 16)	4.77 (L=37, BS = 16)
100x [28] (V100)	328 (L=34, BS = 1)	775 (L=35, BS = 1)	-
[47] (A100-PCIE-80G)	171 (L=34, BS = 1)	-	8.58 (L=35, BS = 1)
GME-Baseline [53] (MI100)	413 (L=17)	658	9.99
GME [53] (Modified Hardware)	33.6 (L=17)	54.5	0.98
WarpDrive (A100-PCIE-80G)	121 (L=34, BS = 1)	113 (L=37, BS = 1)	5.88 (L=37, BS = 1)

We also evaluate transciphering using AES-CRT with 128-bit security over CKKS, as shown in Table XV. We use a max level of 46. Our GPU implementation achieves transciphering for 512KB of data in 3.5 minutes, offering a $31.6\times$ performance improvement over the multi-threaded baseline on a 48-core CPU with Hyper-Threading.

VI. Discussion

A. Design Tradeoffs

WarpDrive employs integer cores and Tensor Cores, without relying on floating-point cores. This choice is primarily driven by concerns over the precision issues of floating-point cores and the additional overhead associated with format conversion. For the primary floating-point cores in GPUs, specifically FP32 cores, their significand precision is only 24 bits (including a hidden bit), which is insufficient to meet precision requirements. Moreover, FP64 units are present in very low quantities in most GPUs, and even in flagship models, their count is merely half that of INT32 or FP32 units. Utilizing FP64 units also incurs additional type conversion overhead. Although Tensor Cores do not offer high precision either, they provide a significant performance advantage.

B. Generality

• **Hardware:** The WarpDrive framework utilizes Tensor Cores with INT8 support and INT32 cores, providing a high degree of generality as these types of cores are supported across various types of modern GPUs. Although the design of the Tensor Core-based NTT and WarpDrive-Fuse is related to the data layout of tensor operations within the registers of threads, the number of SPs within an SM, and the computational power ratio between Tensor Cores and CUDA Cores, these can be addressed on GPUs with different architectures.

TABLE XV: Evaluation of Transciphering using AES-CTR-128 over CKKS.

Scheme (Hardware)	Latency (min)	Block Size (bits)	Blocks (#)	Data Size (KB)
Baseline (Hygon C86 7265 CPU)	110.8	128	2^{15}	512
WarpDrive (A100-PCIE-80G)	3.5			

Firstly, the data layout of tensor operations has remained unchanged across the last three NVIDIA GPU generations [41], [42], [44]. For architectures beyond NVIDIA GPUs, adaptation is also achievable. While WarpDrive utilizes NVIDIA's Tensor Cores, many modern GPUs, including those from AMD and Intel, are beginning to incorporate similar matrix processing capabilities and support for low-precision arithmetic like INT8 [10], [27]. The data layout of tensor operations can be deduced using the methods outlined in [59]. As a result, it can be expected that only minor adjustments are needed to achieve the same optimization levels of Tensor Core-based NTT on different architectures or newer GPUs.

Secondly, as long as a GPU has both tensor processing units and integer processing units, we can easily adjust the WarpDrive-Fuse strategy based on the number of processing units and the ratio of different computational powers. This includes selecting units and allocating proportions to ensure optimal performance across various GPU architectures.

• **Schemes:** In this paper, WarpDrive employs CKKS in its implementation and evaluation. However, by leveraging our existing design and implementations, incorporating additional logic for homomorphic operations, and integrating a few supplementary kernels, WarpDrive can be easily adapted to homomorphic encryption schemes that utilize RLWE ciphertexts, such as BGV [13] and BFV [21]. Moreover, since NTT is generally a performance bottleneck in homomorphic encryption schemes, and ciphertexts often have parallelism that can be further exploited, our NTT and kernel designs can be adapted to schemes using RGSW, MLWE, and other ciphertext formats, such as TFHE [16] and ModHE [39].

VII. Conclusion

In this paper, we propose WarpDrive, a comprehensive framework that significantly improves NTT processing efficiency by minimizing operations and pipeline stalls due to memory access. WarpDrive enables the concurrent utilization of Tensor and CUDA Cores and enhances GPU efficiency through full exploitation of intra-ciphertext parallelism.

Our detailed evaluation on the NVIDIA A100 platform demonstrates that WarpDrive achieves substantial performance improvements. The NTT and homomorphic multiplication throughput are $13.4\times$ and $3.5\times$ higher than state-of-the-art solutions, respectively. For specific FHE workloads, our approach delivers $2.8\times$ better performance even with a much lower batch size.

VIII. Acknowledgements

This work is supported in part by National Natural Science Foundation of China No.61902392.

REFERENCES

[1] "CKKS-GPU-CORE," https://github.com/scale-snu/ckks-gpu-core.

[2] "cuFHE," https://github.com/vernamlab/cuFHE.

[3] "NuFHE," https://github.com/nucypher/nufhe.

[4] "TensorFHE," https://hub.docker.com/r/suen0/tensorfhe.

[5] R. Agrawal, J. H. Ahn, F. Bergamaschi, R. Cammarota, J. H. Cheon, F. DM de Souza, H. Gong, M. Kang, D. Kim, J. Kim *et al.*, "High-precision RNS-CKKS on fixed but smaller word-size architectures: theory and application," in *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2023, pp. 23–34.

[6] R. Agrawal, L. de Castro, G. Yang, C. Juvekar, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, and A. Joshi, "FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 882–895.

[7] E. Aharoni, N. Drucker, G. Ezov, E. Kushnir, H. Shaul, and O. Soceanu, "E2E near-standard and practical authenticated transciphering," *Cryptology ePrint Archive*, 2023.

[8] A. Al Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "Privft: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, pp. 226 544–226 556, 2020.

[9] A. Al Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on GPUs: An implementation using CUDA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 2, pp. 70–95, 2018.

[10] AMD, *AMD Instinct MI100 Instruction Set Architecture: Reference Guide*, https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/instinct-mi100-cdna1-shader-instruction-set-architecture.pdf, 2020.

[11] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.

[12] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 587–617.

[13] 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.

[14] H. Chen and K. Han, "Homomorphic lower digits removal and improved fhe bootstrapping," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 315–337.

[15] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

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

[17] W. Dai and B. Sunar, "cuHE: A homomorphic encryption accelerator library," in *International Conference on Cryptography and Information Security in the Balkans*. Springer, 2015, pp. 169–186.

[18] DESILO, "Liberate.FHE: A New FHE Library for Bridging the Gap between Theory and Practice with a Focus on Performance and Accuracy," 2023, https://github.com/Desilo/liberate-fhe.

[19] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.

[20] 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.

[21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[22] 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.

[23] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running Encrypted Discretized Neural Networks in Seconds," in *Network and Distributed System Security Symposium (NDSS)*, 2023, pp. 1–17.

[24] C. Gouert, V. Joseph, S. Dalton, C. Augonnet, M. Garland, and N. G. Tsoutsos, "ArctyrEX: Accelerated encrypted execution of general-purpose applications," *arXiv preprint arXiv:2306.11006*, 2023.

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

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

[27] Intel, "Explore intel® artificial intelligence solutions," https://www.intel.com/content/www/us/en/artificial-intelligence/overview.html, (Accessed on 10/12/2024).

[28] 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.

[29] W. Jung, E. Lee, S. Kim, J. Kim, N. Kim, K. Lee, C. Min, J. H. Cheon, and J. H. Ahn, "Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," *IEEE Access*, vol. 9, pp. 98 772–98 789, 2021.

[30] A. Karatsuba, "Multiplication of multidigit numbers on automata," in *Soviet physics doklady*, vol. 7, 1963, pp. 595–596.

[31] A. Kim, Y. Polyakov, and V. Zucca, "Revisiting homomorphic encryption schemes for finite fields," in *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*. Springer, 2021, pp. 608–639.

[32] J. Kim, W. Choi, and J. H. Ahn, "Cheddar: A swift fully homomorphic encryption library for cuda gpus," *arXiv preprint arXiv:2407.13055*, 2024.

[33] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[34] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, "BTS: An accelerator for bootstrappable fully homomorphic encryption," in *Proceedings of the 49th annual international symposium on computer architecture*, 2022, pp. 711–725.

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

[36] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[37] T. Morshed, M. M. Al Aziz, and N. Mohammed, "CPU and GPU accelerated fully homomorphic encryption," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 142–153.

[38] C. V. Mouchet, J.-P. Bossuat, J. R. Troncoso-Pastoriza, and J.-P. Hubaux, "Lattigo: A multiparty homomorphic encryption library in go," in *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, 2020, pp. 64–70.

[39] A. Mukherjee, A. C. M. Aikata Aikata, Y. Lee, S. Kwon, M. Deryabin, and S. S. Roy, "ModHE: Modular homomorphic encryption using module lattices," 2023.

[40] S. Narisada, H. Okada, K. Fukushima, S. Kiyomoto, and T. Nishide, "GPU acceleration of high-precision homomorphic computation utilizing redundant representation," in *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2023, pp. 1–9.

[41] NVIDIA, *NVIDIA Tesla V100 GPU Architecture*, https://www.nvidia.com/en-us/data-center/v100/, 2017.

[42] NVIDIA, *NVIDIA A100 Tensor Core GPU Architecture*, https://www.nvidia.com/en-us/data-center/a100/, 2020.

[43] NVIDIA, *CUDA Toolkit Documentation*, https://docs.nvidia.com/cuda/archive/11.3.0/, 2021.

[44] NVIDIA, *NVIDIA H100 Tensor Core GPU Architecture*, https://www.nvidia.com/en-us/data-center/h100/, 2022.

[45] NVIDIA, *CUDA Parallel Thread Execution*, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-multiply-accumulate-instructions, 2024.
[46] NVIDIA, *NVIDIA NSIGHT Compute Documentation*, https://developer.nvidia.com/nsight-compute, 2024.
[47] J. Park, D. Kim, J. Kim, S. Kim, W. Jung, J. H. Cheon, and J. H. Ahn, "Toward practical privacy-preserving convolutional neural networks exploiting fully homomorphic encryption," *arXiv preprint arXiv:2310.16530*, 2023.
[48] G. Project, *GCC 10.2.1 Release Notes*, https://gcc.gnu.org/gcc-10/, 2020.
[49] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.
[50] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *2019 IEEE International symposium on high performance computer architecture (HPCA)*. IEEE, 2019, pp. 387–398.
[51] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.
[52] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.
[53] K. Shivdikar, Y. Bao, R. Agrawal, M. Shen, G. Jonatan, E. Mora, A. Ingare, N. Livesay, J. L. Abellán, J. Kim *et al.*, "GME: GPU-based microarchitectural extensions to accelerate homomorphic encryption," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 670–684.
[54] Z. Wang, P. Li, R. Hou, Z. Li, J. Cao, X. Wang, and D. Meng, "HE-Booster: An efficient polynomial arithmetic acceleration on GPUs for fully homomorphic encryption," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 4, pp. 1067–1081, 2023.
[55] 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.
[56] Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang, and X. Li, "Poseidon: Practical homomorphic encryption accelerator," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 870–881.
[57] Zama, "TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data," 2022, https://github.com/zama-ai/tfhe-rs.
[58] H. Zhao, W. Cui, Q. Chen, Y. Zhang, Y. Lu, C. Li, J. Leng, and M. Guo, "Tacker: Tensor-Cuda core kernel fusion for improving the GPU utilization while ensuring QoS," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 800–813.
[59] T. Zhou, F. Zheng, G. Fan, L. Wan, W. Tang, Y. Song, Y. Bian, and J. Lin, "ConvKyber: Unleashing the power of AI accelerators for faster Kyber with novel iteration-based approaches," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 2, pp. 25–63, 2024.
[60] Y. Zhu, X. Wang, L. Ju, and S. Guo, "FxHENN: FPGA-based acceleration framework for homomorphic encrypted CNN inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 896–907.