TensorPolyMul: Accelerating Polynomial Multiplication in NTT-unfriendly Lattice-based Cryptography Using Tensor Cores

Yi Bian†, Fangyu Zheng‡*, Jiwu Jing‡
† School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China
‡ School of Cryptology, University of Chinese Academy of Sciences, Beijing, China
bianyi18@mails.ucas.ac.cn, {zhengfangyu, jwjing}@ucas.ac.cn

***Abstract*—The urgent demand for computing power in Artificial intelligence (AI) technology has driven the rapid development of dedicated accelerators. Meanwhile, the threat posed by quantum computing to traditional public-key cryptography has prompted the emergence of post-quantum algorithms, such as lattice-based cryptography. However, performance issues with these algorithms have raised concerns within the industry about the transition to quantum-safe solutions. In this paper, we propose a novel universal framework for NTT-unfriendly lattice-based post-quantum algorithms, leveraging NVIDIA's AI accelerator Tensor Core to address this challenge. By employing techniques such as polynomial matrixization and multi-precision representation, we effectively transform the primary workload (i.e., polynomial multiplication) into a series of small-coefficient matrix multiplications that can be directly accelerated by Tensor Cores. This approach effectively bridges the gap between typical Tensor Core workloads and the core workloads of lattice-based post-quantum cryptography. As a case study, we implemented a prototype called TensorPolyMul to provide an implementation of Saber, a quantum-safe Key Encapsulation Mechanism (KEM). The experiments showcase that TensorPolyMul surpasses the state-of-the-art Tensor Core-based work, achieving remarkable speed-ups of 1.53×, 1.33×, 1.62×, and 1.22× for Inner Product, MatrixVecMul, Encaps, and Decaps, respectively.**

***Index Terms*—Lattice-based Cryptography, Saber, Tensor Core, Polynomial Multiplication, Post-quantum Cryptography**

I. Introduction

As quantum computing technology advances, traditional public-key cryptosystems such as RSA [1] and ECC (Elliptic Curve Cryptography) [2] face unprecedented security threats. Quantum computers have the potential to leverage Shor's algorithm [3] to efficiently break these conventional encryption algorithms, thus threatening the security of modern communication and data protection. Consequently, Post-Quantum Cryptography (PQC) [4] has emerged as a crucial research area to develop cryptographic algorithms that can efficiently withstand quantum attacks.

Researchers have explored various acceleration techniques to enhance the performance of these post-quantum algorithms, with GPU (Graphics Processing Unit) parallel computing being an effective approach [5]–[10]. GPUs possess substantial parallel computing capabilities that can significantly expedite complex computational tasks. Tensor Core [11], a specialized hardware unit in NVIDIA GPUs, is designed for efficient matrix computations, offering higher computational efficiency and lower power consumption. Although initially designed for deep learning tasks, the efficient matrix multiplication capabilities of Tensor Core can also be leveraged to accelerate polynomial multiplication.

Efficient computation of polynomial multiplication is critical for the practical deployment of post-quantum cryptographic algorithms. A common acceleration method is using the Number Theoretic Transform (NTT), with several studies [8], [9], [12] leveraging Tensor Core for this purpose. However, these accelerations are not optimal in terms of computational complexity when compared to the best butterfly operations, and their advantages rely heavily on Tensor Core's superior computational performance. Therefore, we considered whether it is possible to accelerate implementations that are more suitable for Tensor Core, which could also provide valuable insights for future PQC designs.

Based on the above considerations, we have turned our research focus to Saber. According to NIST IR 8413 [13], Saber is a very efficient scheme whose security is supported by a large body of cryptographic research. Saber's use of the power of 2 moduli and rounding is intended to make implementation easier relative to other designs, but disallow an NTT implementation of polynomial multiplication. This structure provides the potential for Tensor Core-friendly implementation.

However, effectively adapting polynomial multiplication to Tensor Core is a challenging task. It requires redesigning the algorithms and adopting suitable data layouts and memory management strategies to maximize the benefits of the Tensor Core. Therefore, the motivation of this paper is to explore how Tensor Core can be utilized to accelerate polynomial multiplication in post-quantum cryptographic algorithms such as Saber [14], ultimately improving the overall performance of these algorithms.

To apply the powerful computational capabilities of Tensor Core to post-quantum cryptography, we propose TensorPolyMul, a mechanism for polynomial multiplication based on Tensor Core that significantly accelerates polynomial multiplication in lattice-based cryptography that is not NTT-friendly. Specifically, we make the following contributions:

*The corresponding author is Fangyu Zheng.

- Firstly, we analyzed the differences between the typical workloads of Tensor Core and the core workloads of lattice-based post-quantum cryptography. A novel method for adapting polynomial multiplication to Tensor Core is proposed to fully leverage the computational capabilities of Tensor Core.
- Secondly, we implemented a prototype called TensorPolyMul, and optimized it for the NTT-unfriendly lattice-based post-quantum algorithm Saber. We employed various optimization techniques such as polynomial matrization, multi-precision representation, memory optimization, and parallel strategies to enhance performance.
- Finally, we conducted a series of comprehensive experiments on TensorPolyMul, including micro-benchmarks and macro-benchmarks, to evaluate the effectiveness of the proposed method [15]. In addition, we also compared it with existing acceleration methods. The results demonstrate that our proposed method is 1.22× to 1.62× faster than the fastest existing method and 13.72× to 40.46× faster than the AVX2 benchmark code [16].

The paper is organized as follows: Section II presents the necessary background. Section III outlines the design of TensorPolyMul. Section IV details its implementation. Section V presents experiments and comparisons with related works. Finally, Section VI concludes the paper.

II. Preliminaries

This section introduces the necessary background knowledge, including the Saber algorithm, Tensor Core, and the CUDA programming model.

A. Saber

Saber is a lattice-based post-quantum cryptographic primitive whose security relies on the hardness of the Module Learning With Rounding (Mod-LWR) problem [14]. It has advanced to the third round of the NIST PQC standardization competition, featuring simplicity, efficiency, and flexibility. Saber offers an IND-CPA secure encryption scheme, Saber.PKE, as shown in Algorithms 1, 2, and 3. It also provides an IND-CCA secure KEM using the Fujisaki-Okamoto transformation, Saber.KEM. The security level can be configured by the rank: $l = 2$ for LightSaber, $l = 3$ for Saber, and $l = 5$ for FireSaber. More information can be found at TABLE I and [14]. It is important to note that the moduli (p, q) chosen by Saber do not natively support the number theoretic transform (NTT) for polynomial multiplication. This limitation motivates the proposal of Tensor Core-based optimizations for Saber.

TABLE I
Saber Parameter Sets

Category	l	n	q	p	T	μ	Security Level
LightSaber	2				2^3	10	I
Saber	3	256	2^{13}	2^{10}	2^4	8	III
FireSaber	4				2^6	6	V

Algorithm 1 Saber.PKE.KeyGen()

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} = \text{gen}(seed_{\boldsymbol{A}}) \in R_q^{l\times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s} = \beta_\mu(R_q^{l\times 1}; r)$
$\boldsymbol{b} = ((\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
return $(\text{pk} := (seed_{\boldsymbol{A}}, \boldsymbol{b}), \boldsymbol{s})$

Algorithm 2 Saber.PKE.Enc$(pk = (\boldsymbol{b}, seed_{\boldsymbol{A}}), m \in R_2; r)$

$\boldsymbol{A} = \text{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l\times l}$
if r is not specified **then**
    $r = \mathcal{U}(\{0,1\}^{256})$
end if
$\boldsymbol{s}' = \beta_\mu(R_q^{l\times 1}; r)$
$b' = (((\boldsymbol{A}\boldsymbol{s}' + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p)) \in R_p^{l\times 1}$
$v' = \boldsymbol{b}^T(\boldsymbol{s}' \bmod p) \in R_p$
$c_m = (v' + h_1 - 2^{\epsilon_p - 1}m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
return $c := (c_m, \boldsymbol{b}')$

B. Tensor Core

Tensor Core is a hardware accelerator introduced by NVIDIA to enhance deep learning computations, available on Volta and subsequent architectures [11]. Tensor Core multiplies two matrices and then adds a third matrix to the result using fused multiply-add operations, like $D = A \times B + C$. It supports various data types and matrix sizes, as shown in TABLE II. When performing large matrix multiplications using Tensor Core, the large matrix must be divided into smaller sub-matrix blocks (tiles). These sub-matrix blocks are represented using the fragment data structure, and Tensor Core is invoked multiple times to perform the calculations.

TABLE II
Matrix Types Supported by Tensor Core on NVIDIA RTX 3070

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	$16 \times 16 \times 16$
__half	__half	__half	$32 \times 8 \times 16$
unsigned char	unsigned char	int	$8 \times 32 \times 16$
signed char	signed char	int	
__nv_bfloat16	__nv_bfloat16	float	
tf32	tf32	float	$16 \times 16 \times 8$
double	double	double	$8 \times 8 \times 4$

CUDA offers various programming interfaces to leverage Tensor Cores, including the CUTLASS [17] and cuBLAS [18] libraries, and Warp Matrix Multiply-Accumulate (WMMA) API. WMMA is an API designed to perform matrix mul-

Algorithm 3 Saber.PKE.Dec$(s, c = (c_m, b'))$

$v = \boldsymbol{b'}^T(\boldsymbol{s} \bmod p) \in R_p$
$m' = (((v - 2^{\epsilon_p - \epsilon_T}c_m + h_2) \bmod p) \gg (\epsilon_p - 1)) \in R_2$
return m'

tiplications and can directly control Tensor Cores. Typical usage of the WMMA API involves the following steps: (1) define a fragment, (2) load matrix data into the fragment, (3) perform matrix multiplication using `mma_sync`, and (4) store the result from the fragment to global or shared memory.

III. Design

In this section, we introduce the overall design of TensorPolyMul from multiple dimensions, including the analysis of Saber's workload, the adaptation between polynomial multiplication and Tensor Core typical workloads, and considerations for using shared memory.

A. Workload Analysis of Saber

Understanding the workload characteristics of Saber is crucial for optimizing its performance, especially when leveraging advanced hardware like GPUs. We analyze the time consumption of various operations in Saber using the official reference code provided by the NIST Post-Quantum Cryptography standardization. The execution time breakdown of Saber is shown in Fig. 1.

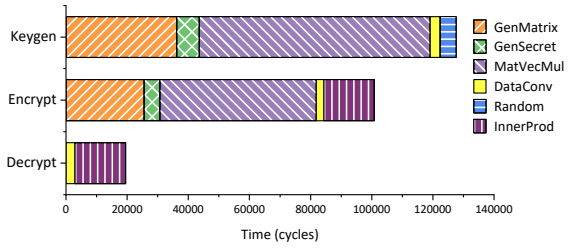

Fig. 1. Breakdown of Saber Execution Time (The execution time primarily consists of inner products and matrix-vector multiplications.)

The most time-consuming operations in Saber's various algorithms are polynomial matrix-vector multiplication and polynomial vector-vector multiplication. Notably, polynomial matrix-vector multiplication can be seen as an extension of polynomial vector multiplication, and polynomial vector multiplication can be viewed as an extension of polynomial multiplication. Therefore, the core computational workload of Saber is polynomial multiplication.

The operations of Saber are performed in the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$. where $x^n \equiv -1 \bmod (x^n+1)$. Assuming two polynomials $\boldsymbol{a}$ and $\boldsymbol{b}$ of degree n, represented as (1).

$$\begin{aligned} \boldsymbol{a} &= \sum_{i=0}^{n-1} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} \\ \boldsymbol{b} &= \sum_{i=0}^{n-1} b_i x^i = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1} \end{aligned} \tag{1}$$

And polynomial multiplication is defined as (2),

$$\boldsymbol{c} = ab = \sum_{i=0}^{n-1} c_i x^i \tag{2}$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \tag{3}$$

Tensor Core natively supports matrix multiplication, which differs significantly from the polynomial multiplication shown in (2). Therefore, it is not feasible to directly exploit Tensor Core to accelerate polynomial multiplications of Saber. We need to further explore methods to bridge the gap between these two operations.

B. Adapting Polynomial Multiplication for Tensor Core

To optimize polynomial multiplication using Tensor Core, it is necessary to convert it into matrix form. While coefficient $\boldsymbol{a}$ in (2) remains fixed, coefficient $\boldsymbol{b}$ can be transformed into an $n \times n$ matrix as shown in (4). Following the usage requirements of Tensor Core, the tile size is 16×16. A practical approach is to simultaneously process 16 instances in each computation to obtain 16 intermediate results, as illustrated in Fig. 2.

$$\begin{aligned} \boldsymbol{c} &= \begin{bmatrix} c_0 & c_1 & \cdots & c_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} a_0 & a_1 & \cdots & a_{n-1} \end{bmatrix} \begin{bmatrix} b_0 & b_1 & \cdots & b_{n-1} \\ -b_{n-1} & b_0 & \cdots & b_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ -b_1 & -b_2 & \cdots & b_0 \end{bmatrix} \end{aligned} \tag{4}$$

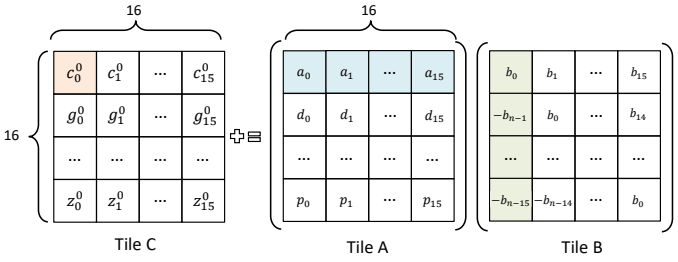

Fig. 2. Data Layout within Fragments using Tensor Cores for Parallel Polynomial Multiplications

However, this method expands in size of $\boldsymbol{b}$ from n to n^2, which is unacceptable for our purposes. A closer inspection of (3) reveals that the expanded coefficient matrix can potentially be compressed. The coefficient $\boldsymbol{b}$ is composed of b_{i-j} (where $0 \le j \le i$) and $-b_{n+i-j}$ (where $i+1 \le j \le n-1$). For $i \in [0, n]$, these coefficients include $[b_n, \ldots, b_0, -b_{n-1}, \ldots, -b_1]$, totaling $2n-1$ coefficients. In addition to the tile size of Tensor Core being 16×16, the WMMA API requires alignment of the pointer to the first element in memory to 256 bits. Considering the prerequisites for utilizing Tensor Core, we zero-padded the vector to length $2n$ in the form of $[b_n, \ldots, b_0, -b_{n-1}, \ldots, -b_1, 0]$ and expanded it into a $2n \times 16$ matrix following the matrix pattern on the right side of 4, where each column's elements are derived from logically left-shifting vector to the left by one position.

C. Multi-precision Pepresentation

Tensor Core supports various data types, as shown in TABLE III. The performance of general matrix multiplication in the table was obtained through testing on NVIDIA RTX 3070 using official code [19]. Generally speaking, lower precision often means higher computing speed but results in a smaller range of accurate representation. Since the modulus of Saber is $q = 2^{13}$, only single-precision FP32 and double-precision FP64 types can precisely represent the modulus. However, compared to low-precision data types like INT8, these high-precision data types suffer from much lower matrix computation throughput.

TABLE III
FEATURES OF DIFFERENT PRECISION SUPPORTED BY TENSOR CORE.

	INT8	BF16	FP16	TF32	FP64
Total Bits	8	16	16	19	64
Data Type	*int8_t* or *uint8_t*	*__nv_bfloat16*	*__half*	*tf32*	*double*
Max Modulus	2^7 or 2^8	2^8-1	$2^{11}-1$	$2^{11}-1$	$2^{53}-1$
Perf.1*	**65.47**×	28.47×	30.43×	8.97×	1.00×
Perf.2*	**221.68**×	95.39×	108.97×	5.00×	1.00×

* These values show the performance differences of Tensor Core at various data types on RTX 3070. Perf.1 represents the general matrix multiplication without shared memory, while Perf.2 means the version with shared memory. The performance of various data types is benchmarked against FP64.

To fully utilize the computing power of Tensor Core, we adopt mixed-precision representation to make a trade-off, specifically using multiple low-precision elements to represent a polynomial coefficient. For Saber, the coefficient is represented in multi-precision form as $a = (a_{hi} \cdot 2^{base} + a_{lo})$. Given Saber's modulus is 2^{13}, the base can be 6 or 7. Either the high or the low part can be represented using the INT8 type. Therefore, the multiplication of coefficients using multi-precision representation is depicted in (5).

$$\begin{aligned} as &= (a_{hi} \cdot 2^{base} + a_{lo}) \cdot (s_{hi} \cdot 2^{base} + s_{lo}) \\ &= a_{hi} \cdot s_{hi} \cdot 4^{base} + (a_{hi} \cdot s_{lo} + a_{lo} \cdot s_{hi}) \cdot 2^{base} \\ &\quad + a_{lo} \cdot s_{lo} \end{aligned} \tag{5}$$

D. Analysis and Usage of Shared Memory

Compared to global memory, shared memory exhibits significantly lower access latency due to it being on the GPU chip, and facilitates fast data sharing among threads within the same thread block. We considered the usage of shared memory from two aspects to improve the performance of CUDA programs, especially in tasks that require thread cooperation.

Consolidating Global Memory Access. Since global memory access is a relatively slow operation, the overall performance can be greatly enhanced by loading frequently accessed data into shared memory. For instance, in Saber matrix-vector multiplication, vectors are frequently accessed. Loading these vectors in shared memory in advance can significantly reduce the data loading overhead for the Tensor Core.

Intermediate Storage and Data Type Conversion. As shown in TABLE II, performing INT8 matrix multiplication using Tensor Core produces results of type `int`, posing a significant issue for Saber because the result type for Saber polynomial multiplication is `uint16_t`. The data loading mode of Tensor Core makes it impossible to directly perform type conversion, which means that we cannot directly store the results from the fragment in global memory. Therefore, we chose to utilize shared memory for temporary data storage, facilitating type conversion and batch data write-back to global memory through multi-thread cooperation. This approach not only addresses the aforementioned challenges but also reduces the latency of global memory access.

IV. IMPLEMENTATION

In this section, we present the detailed implementation of TensorPolyMul from multiple perspectives, including the architecture, multi-precision representation, memory optimization, and parallelism model.

A. Overview

Fig. 3 illustrates the overall architecture of our proposed Tensor Core-based polynomial multiplication scheme TensorPolyMul. Since keys can be generated in offline mode, the performance requirement for key generation is not as high as for other operations. Therefore, we disregard the optimization for key generation.

- **Pre-processing**: This stage includes various operations such as random number generation, data conversion, matrix generation, and sampling. As we employ multi-precision representation, we need to split the polynomial coefficient into two `int8_t` types and expand the polynomial that can remain fixed into a matrix, as described in III. These functionalities are implemented on the GPU and executed by CUDA Cores.
- **Matrix Multiplication based on Tensor Core**: The preprocessed data is processed by Tensor Core for matrix multiplication. Specifically, each warp first loads the high and low parts of the coefficients into fragment *matrix_a*, and then loads the coefficients from the expanded matrix into fragment *matrix_b* to perform the MMA operation. Later, the final result is obtained through left shifting and addition. Finally, the result from the *accumulator* fragment is stored in global memory.
- **Post-processing**: This stage includes reduction, right shifting, and data type conversion. The order of these processes varies depending on the specific operation. Similarly, these functions are also implemented by CUDA Cores, which can efficiently improve overall performance.

B. Optimization of Multi-Precision Representation

As indicated in section III, we employ multi-precision to represent the polynomial coefficients, which is significantly faster than FP32 and FP64 types. However, a closer inspection of (5) reveals that it requires 4 multiplications and 3 additions, which could potentially diminish the performance improvements provided by Tensor Core. To address this issue, we conducted a detailed analysis of the algorithm and found that both

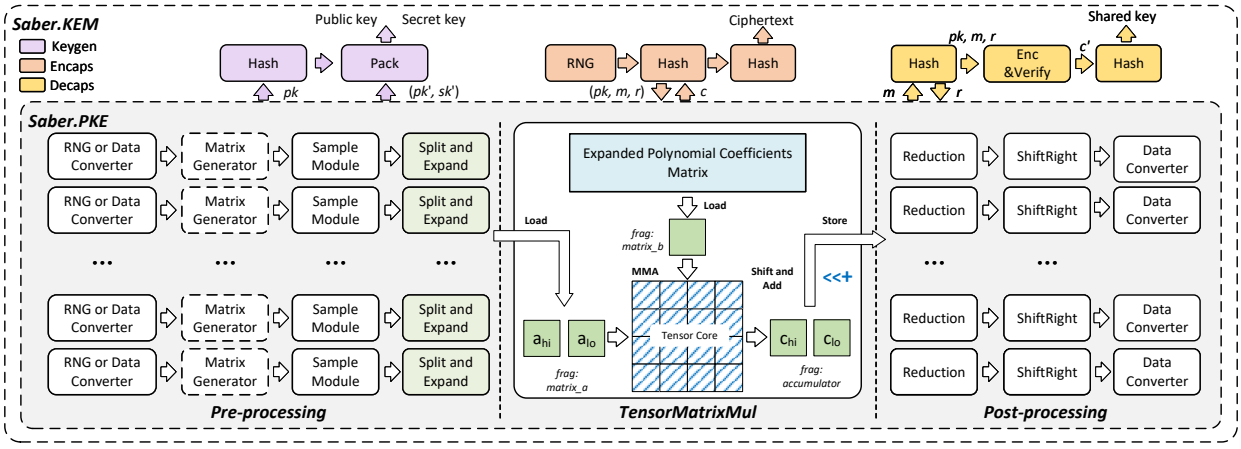

Fig. 3. The Overall Architecture of Saber based on Tensor Core

polynomial matrix-vector multiplication and polynomial vector multiplication involve a polynomial vector whose elements are sampled from a centered binomial distribution (CBD) within the range $[-\mu/2, \mu/2]$. As demonstrated in TABLE I, regardless of the security level of Saber, these elements can be represented using a single `int8_t` type. Therefore, (5) can be optimized to (6), which only requires only 2 multiplications and 1 addition. This optimization significantly reduces the computational complexity and enhances the performance of the Tensor Core for polynomial multiplications.

$$\begin{aligned} as &= (a_{hi} \ll base + a_{lo}) \cdot s = (a_{hi} \cdot 2^{base} + a_{lo}) \cdot s \\ &= a_{hi} \cdot s \cdot 2^{base} + a_{lo} \cdot s \end{aligned} \tag{6}$$

When the input of Tensor Core is INT8 type, the type of accumulator and the final result is `int`, as presented in TABLE II. For Saber, with vector dimension $n = 256$, modulus $q = 2^{13}$, μ is less than 2^4. Therefore, the range of the result is within $[-2^4nq, 2^4nq]$. Given that 2^4nq can reach a maximum of 2^{25}, there is no concern about data overflow.

C. Memory Optimization

To enhance computational efficiency, we optimize memory usage, focusing on global and shared memory utilization.

Optimization for Global Memory. We employ block-level parallelism, where each block computes 1 or 16 instances depending on specific operations (encrypt, or decrypt). This design optimizes not only for Tensor Core computation patterns but also for global memory access. If each thread handles a Saber instance, the memory access stride may be large, increasing memory access requests. To reduce the number of memory transactions, we consolidate the memory of multiple instances into one block. This ensures that threads within a block access memory with the smallest possible stride, thereby minimizing memory transactions and improving memory throughput.

Optimization for Shared Memory. Due to limited shared memory size, it is necessary to identify which operations can utilize shared memory to accelerate polynomial multiplication. Our method involves two types of inputs: fixed inputs (such as public and private keys) and instance-specific inputs (such as random numbers, plaintexts, and ciphertexts). Fixed inputs are expanded into matrix form, as shown in the bottom matrix of Fig 4. Instance-specific inputs are processed in blocks of 16 instances, as presented in the top matrix of Fig 4. The analysis of data reusability in the polynomial multiplication of Saber is shown in TABLE IV.

D. Parallelism Model

Tensor Core processes only 16×16 matrix at a time, so it is necessary to divide the larger matrix into smaller sub-matrices for processing. The following example illustrates this process using the inner product (InnerProd) of Saber.PKE.Enc, as shown in Fig. 4. The inner product is defined as $v' = \text{InnerProd}(\boldsymbol{b'}, \boldsymbol{s} \bmod p, p)$. The multiplication of polynomial vectors with dimensions $L \times N$ can be seen as the accumulation of L instances of N-dimensional polynomial multiplication. Therefore, our focus is on polynomial multiplication. On the top of Fig. 4, 16 instances of $\boldsymbol{s}$ are sequentially stored in memory with a length of N. On the bottom, the polynomial vector $\boldsymbol{b'}$ is expanded to accommodate matrix multiplication, with dimensions $L \times (16 \times 2 \times N)$.

The WMMA API `load_matrix_sync` provided by CUDA loads 16×16 elements into the fragment each time. The loading pattern is influenced by `ldm`, which refers to the stride between consecutive rows in a row-major layout or columns in a column-major layout. To fully utilize the accumulator-type fragment, as shown in section III-B, we set the `ldm` of the upper matrix to N. This configuration enables the simultaneous loading of 16 instances, each containing 16 polynomial coefficients.

For the fixed polynomial vector $\boldsymbol{b'}$, the ldm of the bottom matrix is set to $2N$. Assume the matrix $\boldsymbol{s}$ is fixed as the initial fragment $(s_0, s_1, \ldots, s_{15})$, matrix $\boldsymbol{b'}$ loads fragments from the row containing $(b_{15}, b_{14}, \ldots, b_0)$, resulting in $(v^0_{15}, v^0_{14}, \ldots, v^0_0)$ from their matrix multiplication. Shifting the starting index by 16 columns retrieves data

TABLE IV
THE DATA USABILITY ANALYSIS OF POLYNOMIAL MULTIPLICATION IN SABER

Operation	Function	Fixed and Expanded	16 Instances	The size of Expanded (LightSaber / Saber / FireSaber)	The Size of 16 Instances (LightSaber / Saber / FireSaber)
Saber.PKE. Enc	MatrixVectorMul($\boldsymbol{A}, \boldsymbol{s}', q$)	$\boldsymbol{A}$, uint16_t	$\boldsymbol{s}'$, int8_t	$L \times L \times (N \times 2 \times 16) \times$ sizeof(uint16_t) (64 kB / 144 kB / 256 kB)	$16 \times (L \times N) \times$ sizeof(int8_t) (8 kB / 12 kB / 16 kB)
	InnerProd($\boldsymbol{b}, \boldsymbol{s}' \bmod p, p$)	$\boldsymbol{b}$, uint16_t	$\boldsymbol{s}'$, int8_t	$L \times (N \times 2 \times 16) \times$ sizeof(uint16_t) (32 kB / 48 kB / 64 kB)	$16 \times (L \times N) \times$ sizeof(int8_t) (8 kB / 12 kB / 16 kB)
Saber.PKE. Dec	InnerProd($\boldsymbol{b}', \boldsymbol{s} \bmod p, p$)	$\boldsymbol{s}$, int8_t	$\boldsymbol{b}'$, uint16_t	$L \times (N \times 2 \times 16) \times$ sizeof(int8_t) (16 kB / 24 kB / 32 kB)	$16 \times (L \times N) \times$ sizeof(uint16_t) (8 kB / 12 kB / 16 kB)

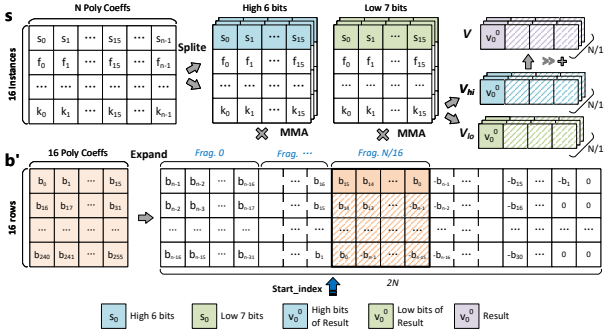

Fig. 4. Parallel Mode for Polynomial Multiplication based on Tensor Core

starting from $(b_{31}, b_{30}, \ldots, b_{16})$, resulting in matrix multiplication with $\boldsymbol{s}$ to compute $(v^0_{31}, v^0_{30}, \ldots, v^0_{16})$. Similarly, this iterative process continues until we obtain the results for $(v^0_{n-1}, v^0_{n-2}, \ldots v^0_{n-16})$. When the matrix $\boldsymbol{s}$ becomes $(s_{16}, s_{17}, \ldots, s_{31})$, the starting index shifts 16 columns to the right from the previous round's starting position, i.e., moving to the row containing $(-b_{n-1}, -b_{n-2}, \ldots, -b_{n-16})$. The process then begins a new iteration and continues until the $\boldsymbol{s}$ matrix loads $(s_{n-16}, s_{n-15}, \ldots, s_{n-1})$.

To fully utilize Tensor Core's computing power, we further decompose the tasks and assign them to different warps for parallel processing. For polynomial multiplication of degree N, there are $N/16$ fragments, requiring $(N/16) \times (N/16)$ matrix multiplications. Therefore, each warp needs to compute $\frac{(N/16)\times(N/16)}{WARPS}$ matrix multiplications. In principle, increasing the number of warps reduces the workload for each warp, thereby enhancing parallelism and improving performance. However, our method is constrained by the degree N, so the upper limit for the number of warps is $N/32$.

V. EVALUATION

This section introduces a series of experiments to evaluate the performance of TensorPolyMul. Additionally, we compare the proposed scheme with related work. The specific configurations of the experimental platform are shown in TABLE V.

TABLE V
EXPERIMENTAL PLATFORM CONFIGURATION

CPU	Intel Xeon Gold 5218R @2.10 GHz	RAM	32 GB
GPU	NVIDIA RTX 3070 (1.5 GHz, 46 SMs, 5888 CUDA Cores, 184 Tensor Cores)	OS	Ubuntu 20.04
Driver	NVIDIA Driver 510.108.03	Software	CUDA 11.6

A. Performance Evaluation

This section evaluates the performance of Saber through micro-benchmark and macro-benchmark tests. All test samples are randomly generated.

1) Micro benchmark: Inner Product and Matrix-Vector Multiplication account for a significant portion of the computation time during the execution of Saber as shown in Fig. 1, optimizing them directly enhances overall performance. TABLE VI shows the micro-benchmark results of Saber under different parameter sets. As analyzed in Section IV-D, our proposed method limits the block size to the maximum value of $SABER\_N$. The results show that performance improves and exhibits linear growth as the batch size increases.

TABLE VI
MICRO-BENCHMARKS FOR SABER

Batch Size	Inner Product (kops)			Matrix-Vec Multiplication (kops)		
	LightSaber	Saber	FireSaber	LightSaber	Saber	FireSaber
32	1339.83	1053.64	830.16	1019.27	488.28	297.80
64	2769.78	2099.83	1689.19	2034.93	971.55	593.40
128	5624.43	4214.56	3409.66	4032.26	1953.13	1190.48
256	11904.76	8636.24	6819.32	8064.52	3886.22	2358.49
512	23448.77	17006.78	13888.89	16156.27	7614.61	4608.34

2) Macro benchmark: The macro benchmarks evaluate the overall performance of the Saber in different parameter sets, including the Public Key Encryption (PKE) and the Key Encapsulation Mechanism (KEM).

TABLE VII
MACRO-BENCHMARKS FOR LIGHTSABER.PKE AND LIGHTSABER.KEM

Batch Size	LightSaber.PKE		LightSaber.KEM	
	Enc (kops/ms)	Dec (kops/ms)	Encaps (kops/ms)	Decaps (kops/ms)
32	171.08/0.19	919.12/0.03	96.77/0.33	95.71/0.33
64	304.98/0.21	1658.87/0.04	177.40/0.36	163.30/0.39
128	496.70/0.26	3318.45/0.04	296.76/0.43	263.31/0.49
256	721.85/0.35	6826.80/0.04	403.53/0.63	343.09/0.75
512	849.86/0.60	10970.16/0.05	469.10/1.09	380.75/1.34

TABLE VII, TABLE VIII, and TABLE IX present the PKE and KEM performance of three different algorithms: LightSaber, Saber, and FireSaber, respectively. It is evident that performance improves significantly with increasing batch size, but the rate of improvement gradually slows down, accompanied by increasing latency. This is attributed to heightened competition for computational resources and increased cache pressure. As batch size increases, more computational

TABLE VIII
MACRO-BENCHMARKS FOR SABER.PKE AND SABER.KEM

Batch Size	Saber.PKE		Saber.KEM	
	Enc (kops/ms)	Dec (kops/ms)	Encaps (kops/ms)	Decaps (kops/ms)
32	105.81/0.30	738.75/0.04	66.97/0.48	66.06/0.48
64	204.76/0.31	1594.95/0.04	126.31/0.51	124.14/0.52
128	367.31/0.35	3072.48/0.05	231.94/0.55	215.22/0.59
256	587.67/0.44	6010.87/0.05	381.49/0.67	328.17/0.78
512	752.97/0.68	9870.56/0.06	486.07/1.05	395.20/1.30

TABLE IX
MACRO-BENCHMARKS FOR FIRESABER.PKE AND FIRESABER.KEM

Batch Size	FireSaber.PKE		FireSaber.KEM	
	Enc (kops/ms)	Dec (kops/ms)	Encaps (kops/ms)	Decaps (kops/ms)
32	67.13/0.49	574.17/0.07	45.12/0.72	46.550.69
64	133.36/0.48	1418.87/0.05	88.70/0.72	84.61/0.76
128	212.51/0.61	1995.96/0.08	143.08/0.90	130.31/0.98
256	329.38/0.78	5239.08/0.05	211.80/1.21	177.68/1.44
512	389.21/1.32	6788.93/0.09	245.11/2.09	197.50/2.59

resources are required to process additional data, intensifying competition among threads. Additionally, there is a greater demand for data caching, resulting in decreased cache hit rates and consequently increased latency.

B. Comparison with Related Works

To comprehensively evaluate the performance of Saber based on Tensor Core, we compare our work with related work, including both CPU and GPU implementations. The result is presented in TABLE X.

1) Performance Comparison with CPU Implementation: We conduct performance tests on our experimental platform using the official Saber source code, including both C and AVX2 implementations. To ensure accuracy, we performed 1000 tests and averaged the results to reduce measurement errors. In micro-benchmark, TensorPolyMul outperforms the official AVX2 implementation by 28.42× and 40.46×, respectively. Compared to the C implementation, we achieve a performance improvement of two orders of magnitude. In macro-benchmark, TensorPolyMul achieves 752.97 kops for encryption and 9870.56 kops for decryption, marking improvements of 18.38× to 37.00× over AVX2 implementations. Moreover, encapsulation and decapsulation performance still reach 13.72× to 17.30× that of AVX2 implementations.

2) Performance Comparison with GPU Implementation: TABLE X compares TensorPolyMul with the latest Saber optimization based on Tensor Core [15]. To maintain consistency with the GPU used in reference [15], we executed our solution on an RTX 3060 Ti. In micro-benchmark, our scheme achieved 1.53× and 1.33× the optimal performance, respectively. This advantage extends to the overall implementation of Saber. Our Saber.PKE.Enc and Saber.PKE.Dec are 1.55× and 1.34× faster than [15], respectively. Similarly, Saber.KEM.Enc and Saber.KEM.Dec also achieved 1.62× and 1.22× the performance of [15]. These results demonstrate the effectiveness of our proposed scheme, highlighting the significant performance improvements brought by methods such as multi-precision representation and memory optimization.

VI. CONCLUSION

In this paper, we propose TensorPolyMul, a method to accelerate NTT-unfriendly post-quantum cryptographic algorithms using Tensor Core, with the Saber algorithm as an example for our prototype. We effectively bridge the gap between the core workload of Saber and the working mode of Tensor Core through a series of methods, including polynomial matrixization, multiple polynomial parallelism, multi-precision representation, and memory optimization. The experimental results demonstrate that the proposed method significantly improves performance. Compared to the current optimal implementation, our method accelerates inner product and matrix-vector multiplication by 1.53× and 1.33×, respectively. Our work demonstrates the significant potential of Tensor Core in NTT-unfriendly post-quantum cryptographic algorithms. In the future, we will support more post-quantum cryptographic algorithms, including lattice-based and hash-based ones.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No.2022YFB3103303.

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] V. S. Miller, "Use of elliptic curves in cryptography," in *Conference on the theory and application of cryptographic techniques*. Springer, 1985, pp. 417–426.

[3] Shor, Peter W, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[4] Bernstein, Daniel J and Lange, Tanja, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 2017.

[5] Gupta, Naina and Jati, Arpan and Chauhan, Amit Kumar and Chattopadhyay, Anupam, "Pqc acceleration using gpus: Frodokem, newhope, and kyber," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 575–586, 2020.

[6] Wan, Lipeng and Zheng, Fangyu and Lin, Jingqiang, "TESLAC: accelerating lattice-based cryptography with AI accelerator," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2021, pp. 249–269.

[7] Gao, Yiwen and Xu, Jia and Wang, Hongbing, "cuNH: Efficient GPU implementations of post-quantum KEM NewHope," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 551–568, 2021.

[8] Wan, Lipeng and Zheng, Fangyu and Fan, Guang and Wei, Rong and Gao, Lili and Wang, Yuewu and Lin, Jingqiang and Dong, Jiankuo, "A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator," in *European Symposium on Research in Computer Security*. Springer, 2022, pp. 514–534.

[9] Zhou, Tian and Zheng, Fangyu and Fan, Guang and Wan, Lipeng and Tang, Wenxu and Song, Yixuan and Bian, Yi and Lin, Jingqiang, "ConvKyber: Unleashing the Power of AI Accelerators for Faster Kyber with Novel Iteration-based Approaches," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 25–63, 2024.

[10] Lee, Wai-Kong and Zhao, Raymond K and Steinfeld, Ron and Sakzad, Amin and Hwang, Seong Oun, "High throughput lattice-based signatures on gpus: Comparing falcon and mitaka," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 4, pp. 675–692, 2024.

TABLE X
PERFORMANCE COMPARISON WITH RELATED WORKS

	Schemes	Supporting Functions		Saber.PKE		Saber.KEM	
		Inner Product (kops)	MatrixVecMul (kops)	Enc (kops)	Dec (kops)	Encaps (kops)	Decaps (kops)
CPU	Ref.C [16]	63.46	21.15	21.03	89.91	16.51	15.07
	Speedup	267.99×	360.03×	35.80×	109.78×	29.44×	26.22×
	Ref.AVX2 [16]	598.46	188.19	40.96	266.77	28.10	28.81
	Speedup	28.42×	40.46×	18.38×	37.00×	17.30×	13.72×
GPU	TMVP-TC [15]	10861	4643	424.44	6259.78	267.72	294.02
	Speedup†	1.57×/1.53×	1.64×/1.33×	1.77×/1.55×	1.58×/1.34×	1.82×/1.62×	1.34×/1.22×
	Ours (RTX 3060 Ti)	16666.67	6172.84	657.08	8407.37	433.65	359.70
	Ours (RTX 3070)	**17006.78**	**7614.61**	**752.97**	**9870.56**	**486.07**	**395.20**

† Each cell shows two metrics: the performance improvement of our proposed scheme on RTX 3070 and RTX 3060 Ti, separated by a slash (/).

[11] Choquette, Jack and Gandhi, Wishwesh and Giroux, Olivier and Stam, Nick and Krashinsky, Ronny, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," *IEEE Micro*, pp. 29–35, 2021.

[12] Hafeez, Muhammad Asfand and Lee, Wai-Kong and Karmakar, Angshuman and Hwang, Seong Oun, "High Throughput Acceleration of Scabbard Key Exchange and Key Encapsulation Mechanism Using Tensor Core on GPU for IoT Applications," *IEEE Internet of Things Journal*, vol. 10, no. 22, pp. 19 765–19 781, 2023.

[13] National Institute of Standards and Technology, "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process," Accessed on July 5, 2024. [Online]. Available: https://doi.org/10.6028/NIST.IR.8413-upd1

[14] D'Anvers, Jan-Pieter and Karmakar, Angshuman and Sinha Roy, Sujoy and Vercauteren, Frederik, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," in *Progress in Cryptology–AFRICACRYPT*. Springer, 2018, pp. 282–305.

[15] Hafeez, Muhammad Asfand and Lee, Wai-Kong and Karmakar, Angshuman and Hwang, Seong Oun, "Efficient TMVP-Based Polynomial Convolution on GPU for Post-Quantum Cryptography Targeting IoT Applications," *IEEE Internet of Things Journal*, 2024.

[16] KULeuven-COSIC, "SABER." [Online]. Available: https://github.com/KULeuven-COSIC/SABER

[17] NVIDIA, "CUTLASS." [Online]. Available: https://nvidia.github.io/cutlass/

[18] NVIDIA Corporation, "cuBLAS," Accessed on July 5, 2024. [Online]. Available: https://docs.nvidia.com/cuda/cublas/

[19] NVIDIA, "NVIDIA cuda-samples." [Online]. Available: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features