ConvKyber: Unleashing the Power of AI Accelerators for Faster Kyber with Novel Iteration-based Approaches

Tian Zhou[1], Fangyu Zheng[2(✉)], Guang Fan[3], Lipeng Wan[2], Wenxu Tang[1], Yixuan Song[3], Yi Bian[4] and Jingqiang Lin[1,5]

[1] School of Cyber Security, University of Science and Technology of China, Heifei, China,
{weekdayzt,wenxutang}@mail.ustc.edu.cn,linjq@ustc.edu.cn
[2] School of Cryptology, University of Chinese Academy of Sciences, Beijing, China,
zhengfangyu@ucas.ac.cn,szxwlp@foxmail.com
[3] Ant Group, Hangzhou, China, {fanguang.fg,songyixuan.syx}@antgroup.com
[4] School of Computer Science and Technology, University of Chinese Academy of Sciences,
Beijing, China, bianyi18@mails.ucas.ac.cn
[5] Beijing Research Institute, University of Science and Technology of China, Beijing, China

Abstract. The remarkable performance capabilities of AI accelerators offer promising opportunities for accelerating cryptographic algorithms, particularly in the context of lattice-based cryptography. However, current approaches to leveraging AI accelerators often remain at a rudimentary level of implementation, overlooking the intricate internal mechanisms of these devices. Consequently, a significant number of computational resources is underutilized.

In this paper, we present a comprehensive exploration of NVIDIA Tensor Cores and introduce a novel framework tailored specifically for Kyber. Firstly, we propose two innovative approaches that efficiently break down Kyber's NTT into iterative matrix multiplications, resulting in approximately a 75% reduction in costs compared to the state-of-the-art scanning-based methods. Secondly, by reversing the internal mechanisms, we precisely manipulate the internal resources of Tensor Cores using assembly-level code instead of inefficient standard interfaces, eliminating memory accesses and redundant function calls. Finally, building upon our highly optimized NTT, we provide a complete implementation for all parameter sets of Kyber. Our implementation surpasses the state-of-the-art Tensor Core based work, achieving remarkable speed-ups of 1.93x, 1.65x, 1.22x and 3.55x for polyvec_ntt, KeyGen, Enc and Dec in Kyber-1024, respectively. Even when considering execution latency, our throughput-oriented full Kyber implementation maintains an acceptable execution latency. For instance, the execution latency ranges from 1.02 to 5.68 milliseconds for Kyber-1024 on R3080 when achieving the peak throughput.

Keywords: Lattice-based Cryptography · GPUs · Tensor Core · Kyber

1 Introduction

The imminent possibility of large-scale quantum algorithm computers capable of practically executing Shor's algorithm [Sho97] in the near future has sparked intensive research in the realm of post-quantum cryptosystems, designed to withstand quantum attacks. In anticipation of the quantum era, NIST proactively issued a call for proposals to replace their current standards for digital signatures, public-key encryption (PKE), and key-encapsulation mechanisms (KEM). Following three rounds of thorough evaluation and scrutiny, NIST announced in July 2022 that they had chosen CRYSTALS-KYBER

(abbreviated as Kyber) as the standard algorithm for Post-Quantum Cryptography (PQC) Public-key Encryption and Key-establishment Algorithms. In contrast to signature algorithms, adopting KEM algorithms appears more pressing due to the threat of "harvest now, decrypt later". In August 2023, the Chromium Project declared its adoption of a hybrid cryptographic algorithm (X25519/Kyber768) for Chrome and Google Servers [Blo].

In the realm of cryptographic schemes based on lattice-related problems, such as Ring-LWE [LPR10], Module-LWE [LS15], and Module-LWR [BPR12], the most time-consuming components typically involve polynomial multiplication (over the ring R_q) and hash functions. Hash functions primarily entail bit operations, which can be expedited through readily available commercial products featuring processor-assisted accelerations, such as the SHA extension found in Intel and ARM CPUs [SKS$^+$21]. Consequently, much of the effort in lattice-based cryptography (LBC) acceleration is centered on optimizing polynomial multiplication. Various techniques exist to expedite polynomial multiplication. In addition to leveraging Karatsuba [Kar63] and Toom-Cook algorithms [Too63], a prevalent approach is to employ Number Theoretic Transform (NTT), especially when the condition $n|(q-1)$ holds, where q represents the modulus and n signifies the dimension. Kyber [BDK$^+$18, Sch], for instance, even incorporates a customized NTT within its algorithms to enhance efficiency.

Up to this point, researchers have put forth numerous optimized implementations of Kyber across various hardware and software platforms, with particular emphasis on leveraging the NTT. Notably, for general use cases, researchers have crafted optimized implementations using SIMD (Single Instruction, Multiple Data) vector instructions on widely adopted x86 and ARM platforms [SKS$^+$21].

1.1   An Opportunity for PQC with AI-accelerators

Many manufacturers have developed high-performance AI (artificial intelligence) accelerators to cater to the demands of AI applications. Notable examples include Google TPU [Clo], Apple M1 [App], and NVIDIA Tensor Core [NVIb].

Compared to general-purpose processors, AI accelerators primarily emphasize low-precision arithmetic, novel data-flow architectures, and often boast significantly higher computational power. For instance, In 2017, Nvidia introduced the Volta architecture, notable for being the first graphics processor to feature dedicated cores, known as Tensor Cores, specifically designed for tensor calculations. These Tensor Cores can perform 64 General Matrix Multiplications (GEMMs) per clock cycle on 4×4 matrices, which contain FP16 (16-bit floating-point) values or a combination of FP16 multiplication and FP32 addition. Despite their small size, these cores effectively break down larger matrices into smaller tiles to compute the final results. Tensor Cores are further enhanced in each new NVIDIA GPU architecture generation. Tensor Cores of the latest Tesla H100 can deliver up to 1979 Tensor TFLOPS for INT8 precision data. In the case of embedded products, the NVIDIA Jetson AGX Orin offers supercomputer-level performance of up to 275 Tensor TOPS while consuming up to 60W of power [NVIc].

The formidable capabilities of AI accelerators open up new possibilities for accelerating Post-Quantum Cryptography (PQC). One pioneering effort in this domain was undertaken by Wan *et al.* [WZL21]. They harnessed the power of Volta Tensor Cores to accelerate the lattice-based scheme LAC [LLZ$^+$18], which utilizes a byte-level modulus. This endeavor yielded significant performance improvements.

1.2   Technical Challenges and Contributions

The substantial performance advantage has inspired researchers to integrate AI accelerators into cryptographic implementations. Since AI accelerators are specialized for machine

learning and neural networks, the primary challenge lies in adapting cryptographic work-loads to their operations while ensuring the accuracy of results and achieving significant performance improvements.

It is noteworthy that most prior work has focused on lattice-based cryptography without the need for Number Theoretic Transform (NTT), specifically schemes like LAC [WZL21, LSZH22] or NTRU [LSZH22]. Their approaches are quite straightforward. For example, Wan *et al.* [WZL21] extend the coefficient vector of a polynomial into a matrix resembling a Toeplitz matrix. Subsequently, it transforms polynomial multiplication $c = ab$ over the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ in LAC into a vector-matrix multiplication, as follows:

$$ab = \begin{bmatrix} a_0 & a_1 & \cdots & a_{n-1} \end{bmatrix}_{n \times 1} \begin{bmatrix} b_0 & b_1 & \cdots & b_{n-1} \\ -b_{n-1} & b_0 & \cdots & b_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ -b_1 & -b_2 & \cdots & b_0 \end{bmatrix}_{n \times n}.$$

Then, they batch multiple such vector-matrix multiplications into matrix-matrix mul-tiplications. These matrix-matrix multiplications can be directly accelerated using the dedicated instructions of Tensor Cores, and the byte-level modulus $q = 251$ used in LAC aligns perfectly with the most efficient INT8 parameter configuration of Tensor Cores.

However, the more commonly employed NTT-based schemes, such as Kyber, are not amenable to Tensor Core acceleration. The prevalent approach for polynomial multipli-cation using NTT relies on the divide and conquer method. Nevertheless, the resulting butterfly operations are incompatible with the Tensor Core's operational mode. In a recent work, Wan *et al.* [WZF+22] further introduced a practical implementation of Kyber that leverages Tensor Cores. In this implementation, they employed the formula method to replace the commonly used butterfly calculations in NTT. Their research into the relationship between accuracy and performance of WMMA (Warp Matrix Multiply and Accumulate) operations provides valuable insights for future endeavors in this domain.
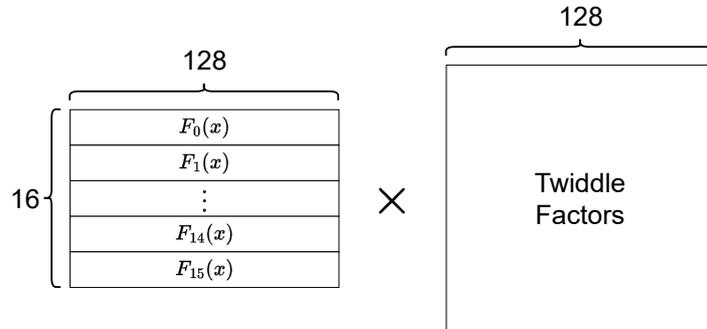

Figure 1: Wan *et al.*'s approach for Kyber's NTT [WZF+22]

The essence of Wan *et al.*'s approach [WZF+22] lies in transforming NTT operations into a "large" matrix multiplication of size 128×128 and processing this large matrix using the native "16×16" matrix multiplication instructions of Tensor Cores. This strategy has delivered highly favorable outcomes, primarily because the dimension $n = 128$ is relatively small. It has achieved a remarkable over six-fold improvement compared to traditional implementations [GJCC20].

However, it is evident that this direct and straightforward approach comes with significant computational complexity. The primary goal of this paper is to harness Tensor Cores to implement a more efficient algorithm, addressing this complexity and enhancing performance.

Based on our explored internal mechanism of Tensor Core and the unique design of the customized NTT in Kyber, this paper proposes an NTT calculation scheme dedicated to Kyber and further implements an entire implementation of Kyber. We name this framework convKyber because we convert the core NTT operations into operations resembling convolutions to leverage Tensor Cores better.

This paper mainly includes the three following contributions.

- Firstly, our work forms an iteration-based framework for an AI accelerator to accelerate module-lattice based cryptography. Two innovative approaches are proposed under this framework that efficiently break down Kyber's NTT into iterative matrix multiplications, resulting in approximately a 75% reduction in costs compared to the state-of-the-art scanning-based methods. Although these approaches are highly tailored for Kyber's NTT, they can be slightly modified for the universal NTT acceleration. (**Section 3**).

- Secondly, we have gained a deep understanding of the internal operational mechanisms of Tensor Cores behind the WMMA API through reverse-engineering. We precisely manipulate the internal resources of Tensor Cores using assembly-level code rather than relying on inefficient standard interfaces, thereby eliminating memory accesses and redundant function calls. (**Section 4**).

- Finally, building upon our highly optimized NTT, we provide a complete implementation for all parameter sets of Kyber, including Kyber-512/768/1024, with non-trivial optimizations for SHA-3, memory access coalescing etc. (**Section 5**).

For the NTT part, empirical results illustrate that our proposed NTT schemes achieve a 12.48x performance improvement compared to conventional butterfly operation based NTT on the same GPU platform. Even when compared to state-of-the-art Tensor Core-based NTT, we achieve a 93% speed-up. For a full Kyber implementation, compared with the state-of-the-art implementation, we have achieved 1.93x, 1.65x, 1.22x, and 3.55x for polyvec_ntt, KeyGen, Enc, Dec in Kyber-1024, as compared to the state-of-the-art Tensor Core-based implementation in the same GPUs. We have also achieved improvements of 1 to 2 orders of magnitude compared to CPU, FPGA, and other embedded platforms. Although achieving such performance advantages requires simultaneous processing of thousands of requests, and the execution latency for these thousands of requests is several hundred times higher compared to other platforms (e.g., CPU, FPGA) handling individual requests, our throughput-oriented full Kyber implementation maintains an acceptable execution latency and is highly suitable for scenarios targeting large-scale users. (**Section 6**).

The rest of this paper is organized as follows. Section 2 introduces some background knowledge, especially the details of Kyber and the features of Tensor Core. Section 3 describes our approach to converting NTT into Tensor core workload. We proposed the Tensor core based NTT approach in Section 4. Section 5 demonstrates the implementation details. In Section 6, We present a series of evaluation and comparison results and discuss the limitations and generalization of the proposed schemes. Then, we make a brief conclusion in Section 7.

2   Preliminary

In this section, we give a basic background of Kyber and AI accelerators.

2.1 Notation and Definition

2.1.1 Notation

For a prime q, $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ is the residue class ring modulo q. $\mathbb{Z}_q^n$ represents n coefficients from $\mathbb{Z}_q$. Define the ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, which means the coefficients are from $\mathbb{Z}_q$. Regular font letters denote elements in R_q (which includes elements in $\mathbb{Z}_q$) and bold lower-case letters represent vectors with coefficients in R_q. By default, all vectors will be column vectors. Bold upper-case letters are matrices. For a vector $\boldsymbol{v}$ (or matrix $\boldsymbol{A}$), $\boldsymbol{v}^T$ (or $\boldsymbol{A}^T$) means its transpose, and $\boldsymbol{v}[i]$ denotes its i-th entry (with indexing starting at zero). For a matrix $\boldsymbol{A}$, $\boldsymbol{A}[i][j]$ denotes the entry in row i, column j (again, with indexing starting from zero). The rank k represents that a polynomial vector contains k polynomials, and a matrix contains $k \times k$ polynomials. For a finite field $F = \mathbb{Z}_q$, the primitive n-th root ω of unity exist whenever $n | (q-1)$, where $\omega^n \equiv 1 (\mathrm{mod}\ q)$.

2.1.2 Module-LWE

A lattice is the set of all integer linear combinations of some linearly independent vectors belonging to the euclidean space. Most lattice-based cryptographic schemes are built upon the assumed hardness of the Short Integer Solution (SIS) [Ajt96] and Learning With Errors (LWE) [Reg05] problems. The LWE problem was popularized by Regev [Reg05] who showed that solving a random LWE instance is as hard as solving certain worst-case instances of certain lattice problems. This assumption states that it is hard to distinguish the uniform distribution from $(\boldsymbol{A}, \boldsymbol{As} + \boldsymbol{e})$, where $\boldsymbol{A}$ is a uniformly-random matrix in $\mathbb{Z}_q^{m \times n}$, $\mathbf{s}$ is a uniformly-random vector in $\mathbb{Z}_q^n$, and $\boldsymbol{e}$ is chosen from some distribution. Later, Lyubashevsky *et al.* [LPR10] introduced a similar adaptation for LWE, called Ring-LWE, which showed that it is also hard to distinguish a variant of the LWE distribution from the uniform one over certain polynomial rings. Combining the security advantages of LWE and the flexibility of Ring-LWE, Langlois *et al.* [LS15] demonstrated the worst-case to average-case reductions for module lattices. Intuitively, the size of matrix $\boldsymbol{A}$ in Module-LWE is $k \times k$, where k is the rank. The elements in the matrix are vectors selected from $\mathbb{Z}_q^n$.

2.2 Description of Kyber

Kyber is an IND-CCA2-secure post-quantum key exchange mechanism. The security of Kyber is based on the hardness of solving the LWE problem in module lattices.

The submission to NIST PQC [SAB$^+$22] lists three different parameter sets, Kyber-512, Kyber-768, and Kyber-1024, aiming at different security levels roughly equivalent to AES-128, AES-192, and AES-256, respectively. The parameters are listed in Table 1, where η_1 and η_2 are the parameters of centered binomial distribution (CBD).

Table 1: Parameter sets for Kyber version 3

	n	k	q	η_1	η_2
Kyber-512	256	2	3329	3	2
Kyber-768	256	3	3329	2	2
Kyber-1024	256	4	3329	2	2

The key generation, encryption, and decryption are described in Algorithm 1, 2, and 3. In the KeyGen phase, d is a random number, ρ and σ are fixed-length intermediate variables generated by d through hash function G. The parameter $\hat{\mathbf{A}}$ is a $k \times k$ polynomial matrix generated by ρ. The parameters $\mathbf{s}$ and $\mathbf{e}$ are polynomial vectors generated through different sample functions but same distribution B_{η_1}. The final parameters need to be

---

Algorithm 1 KYBER.CPAPKE.KeyGen(): key generation

Ensure: Secret key sk, Public key pk.
1: $d \leftarrow \textbf{Random()}$
2: $(\rho, \sigma) := G(d)$
3: $\hat{\mathbf{A}} \leftarrow \text{Gen\_matrix\_}\hat{\mathbf{A}}(\rho)$, $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
4: $\mathbf{s} \leftarrow \text{Sample\_s}(\sigma)$, $\mathbf{s} \in R_q^k$ from B_{η_1}
5: $\mathbf{e} \leftarrow \text{Sample\_e}(\sigma)$, $\mathbf{e} \in R_q^k$ from B_{η_1}
6: $\hat{\mathbf{s}} := \text{NTT}(\mathbf{s})$
7: $\hat{\mathbf{e}} := \text{NTT}(\mathbf{e})$
8: $\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
9: **return** $pk := \text{Encode}(\hat{\mathbf{t}}||\rho)$, $sk := \text{Encode}(\hat{\mathbf{s}})$

---

compressed and encode. In the Enc phase, the public key pk will be decoded first. Here,

---

Algorithm 2 KYBER.CPAPKE.Enc(): encryption

Require: Public key pk, Message m, Random seed r
Ensure: Ciphertext c
1: $(\hat{\mathbf{t}}, \rho) \leftarrow \text{Decode}(pk)$
2: $\hat{\mathbf{A}}^T \leftarrow \text{Gen\_matrix\_}\hat{\mathbf{A}}^T(\rho)$, $\hat{\mathbf{A}}^T \in R_q^{k \times k}$ in NTT domain
3: $\mathbf{r} \leftarrow \text{Sample\_r}(r)$, $\mathbf{r} \in R_q^k$ from B_{η_1}
4: $\mathbf{e}_1 \leftarrow \text{Sample\_e}_1(r)$, $\mathbf{e}_1 \in R_q^k$ from B_{η_2}
5: $e_2 \leftarrow \text{Sample\_e}_2(r)$, $e_2 \in R_q$ from B_{η_2}
6: $\hat{\mathbf{r}} := \text{NTT}(\mathbf{r})$
7: $\mathbf{u} := \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
8: $v := \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}(m)$
9: **return** $c_1 := \text{Encode}_u(\mathbf{u})$, $c_2 := \text{Encode}_v(v)$

---

we need to emphasize that e_2 and v are polynomials rather than vectors. The ciphertext c consists of two parts: c_1 and c_2, which are obtained from $\mathbf{u}$ and v with different encode. Correspondingly, in the Dec phase, these two parts need to be decoded with different functions first. Then the NTT and the subsequent INTT are performed.

Algorithm 3 KYBER.CPAPKE.Dec(): decryption

Require: Secret key sk, Ciphertext c
Ensure: Message m
1: $\mathbf{u} := \text{Decode}_u(c)$
2: $v := \text{Decode}_v(c)$
3: $\hat{\mathbf{s}} := \text{Decode}(sk)$
4: **return** $m := \text{Compress}(v - \text{NTT}^{-1}(\hat{\mathbf{s}} \circ \text{NTT}(\mathbf{u})))$

2.3   AI Accelerator and Tensor Core

Due to the explosive growth of AI applications, general-purpose processors are having difficulty meeting the needs of machine learning. Therefore, a dedicated AI accelerator, an application-specific integrated circuit with a more specific design, may gain far more efficiency. The well-known AI accelerators include Google TPU, Apple M1, M1 MAX, M1 Pro, and ARM NPU. These accelerators mainly focus on optimized memory use and lower precision arithmetic to accelerate calculation and increase the throughput.

Tensor Core. In December 2017, NVIDIA released the 1st generation Tensor Core (on Volta architecture), which is just for tensor calculations. Tensor Cores are designed to carry 64 GEMMs (General Matrix Multiplication) per clock cycle on 4×4 matrices, containing FP16 values (16-bit floating-point numbers) or FP32 (the *float* format). A year later, NVIDIA launched the Turing architecture Tensor Core, which has been updated to support other data formats, such as INT8 (8-bit integer values). In the latest Ampere architecture, NVIDIA has improved the performance (256 GEMMs per cycle, up from 64) and added further data formats, shown in Table 2.

Up to now, Tensor Core can only support operations at the warp level, usually 32 threads. The warp matrix function requires co-operation from all threads in the warp and performs $D = A \times B + C$, where A, B, C, D are matrices with specific size, as shown in Fig. 2.
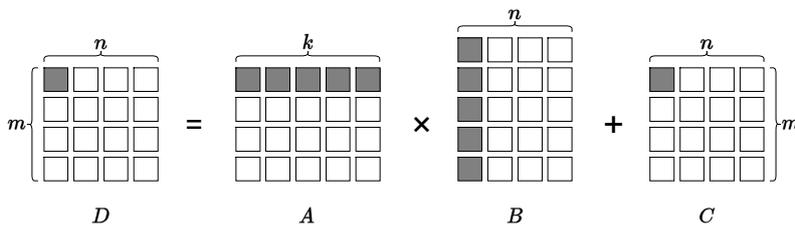

Figure 2: A warp-level m-n-k matrix operation

It is further complicated by threads holding only a fragment (a type of opaque architecture-specific ABI data structure) of the overall matrix, with the developer not allowed to make assumptions on how the individual parameters are mapped to the registers participating in the matrix multiply-accumulate. There are also some restrictions on matrix size. Generally, k is fixed to 16, and m can be 8, 16, or 32 (n corresponds to 32, 16, or 8).

The WMMA instruction exhibits the shortest latency when using INT8 as the computational precision [WZF+22]. We have adopted this technique, decomposing values exceeding 8 bits into multiple INT8 representations.

Given the degree of the polynomial ($n = 256$) in Kyber and the proposed schemes (Section 3.2.2 and Section 3.2.3), we employ m, n, and k values of 16, 16, and 16, respectively.

Table 2: Precision supported by multiple generations of Tensor Core

	Volta	**Turing**	**Ampere**
Precision	*FP16*	*FP16, INT8, INT4, INT1*	*FP64, TF32, bfloat16, FP16, INT8, INT4, INT1*

3   Decomposing NTTs into Matrix Multiplications

This section describes how we decompose NTTs into matrix multiplications.

3.1   The Limitation of Scanning-based Methods

Wan *et al.* [WZF+22] employed a straightforward approach to translate NTT workloads to Tensor Cores. They organized several polynomials that required processing into a matrix

and placed the twiddle factors into another matrix, wherein R is a power of 2, facilitating Montgomery reduction, as follows:

$$
\begin{bmatrix}
\zeta^{0 \times 2\mathbf{br}_7(0)}R & \zeta^{0 \times 2\mathbf{br}_7(1)}R & \cdots & \zeta^{0 \times 2\mathbf{br}_7(127)}R \\
\zeta^{1 \times 2\mathbf{br}_7(0)}R & \zeta^{1 \times 2\mathbf{br}_7(1)}R & \cdots & \zeta^{1 \times 2\mathbf{br}_7(127)}R \\
\vdots & \vdots & \ddots & \vdots \\
\zeta^{127 \times 2\mathbf{br}_7(0)}R & \zeta^{127 \times 2\mathbf{br}_7(1)}R & \cdots & \zeta^{127 \times 2\mathbf{br}_7(127)}R
\end{bmatrix}_{128 \times 128}
$$

On average, computing a single coefficient in the NTT domain required 128 multiplications.

It is important to note that only a fixed-size tile can be loaded into a fragment at a time, while the target matrix is significantly larger. Therefore, Wan *et al.* devised two scanning methods based on the raw precision of the data being processed. For parameters with element values less than 8 bits (e.g., 256 or 128 for signed numbers), such as the secret $\mathbf{s}$ and random noise $\mathbf{r}, \mathbf{e}$ generated from CBD (Centered Binomial Distribution), which have at most 3 significant bits, they employed a basic NTT method.

However, this direct approach is not very efficient. Firstly, it requires 16 concurrent inputs, i.e., batching is necessary. Secondly, its computational workload is relatively high. For Kyber's NTT, it necessitates a total of $2 \times 128 \times 128$ (where 2 accounts for odd and even coefficients) element multiplications. Consequently, the computational complexity of their scheme for a single NTT operation is equivalent to performing $\frac{2 \times 128 \times 128}{16 \times 16 \times 16} = 8$ instances of 16×16 matrix multiplications. While this scheme is functional in its design, it is straightforward and offers ample opportunities for optimization.

3.2   The Proposed Iteration-based Algorithms

In this section, we decompose NTTs into matrix multiplications (Section 3.2.1). Building on this, and varying the handling of twiddle factors, we introduce a 2-phase scheme (Section 3.2.2) and a 3-phase scheme (Section 3.2.3), respectively.

3.2.1   Decomposing Kyber's NTT

Let us revisit Kyber's NTT. Kyber has incorporated NTT into its algorithms, resembling the polynomial form of the Chinese Remainder Theorem (CRT). For the prime modulus $q = 3329$ with $q - 1 = 2^8 \cdot 13$, the base field $\mathbb{Z}_q$ includes 2^8-th roots of unity. Consequently, the defining polynomial $(X^{256} + 1)$ of the ring R can be factored into 128 polynomials of degree 2 modulo q. This polynomial can be expressed as:

$$
X^{256} + 1 = \prod_{i=0}^{127}(X^2 - \zeta^{2i+1}) = \prod_{i=0}^{127}(X^2 - \zeta^{2\mathbf{br}_7(i)+1})
$$

Here, $\mathbf{br_7}(i)$ for $i = 0, 1, \cdots, 127$ represents the bit reversal of the unsigned 7-bit integer i. Consequently, the NTT of a polynomial $f \in R_q$ results in a vector of 128 polynomials of degree 1, and can be represented as:

$$
(f \bmod X^2 - \zeta^{2\mathbf{br}_7(0)+1}, \cdots, f \bmod X^2 - \zeta^{2\mathbf{br}_7(127)+1})
$$

In Kyber, the NTT is defined $\mathrm{NTT} : R_q \to R_q$ to be the bijection that maps $f \in R_q$ to the polynomial with the aforementioned coefficient vector. Hence,

$$
\mathrm{NTT}(f) = \hat{f} = \hat{f}_0 + \hat{f}_1 X + \cdots + \hat{f}_{255} X^{255}
$$

with

$$
\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2\mathbf{br}_7(i)+1)j} \tag{1}
$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1}\zeta^{(2\mathbf{br}_7(i)+1)j} \tag{2}$$

$$f_{2i} = n^{-1} \cdot \sum_{j=0}^{127} \hat{f}_{2j}\zeta^{-(2\mathbf{br}_7(j)+1)i} \tag{3}$$

$$f_{2i+1} = n^{-1} \cdot \sum_{j=0}^{127} \hat{f}_{2j+1}\zeta^{-(2\mathbf{br}_7(j)+1)i} \tag{4}$$

Equation (1) - (4) is evidently far from the 16×16 matrix multiplication we envision for. As a starting point, we can attempt to transform the input vector with 128 elements into a 16×8 matrix. We take the NTT calculation of $\hat{f}_{2i}$ as an example (i.e., Equation (1)). The NTT calculation for $\hat{f}_{2i+1}$ is identical, so we will not derive it separately. While there are some differences in the INTT part (mainly in the order of i and j), the basic approach is similar, and we detail it in Appendix A.

To facilitate subsequent explanations, we introduce the following notations:

$$i = 8i_0 + i_1, j = 8j_0 + j_1, i_0, j_0 \in \mathbb{Z}_{16}, i_1, j_1 \in \mathbb{Z}_8$$

$$f_0(i_0, i_1) = f_{2(8i_0+i_1)} = f_{2i}, f_1(i_0, i_1) = f_{2(8i_0+i_1)+1} = f_{2i+1}$$

$$\hat{f}_0(i_0, i_1) = \hat{f}_{2(8i_0+i_1)} = \hat{f}_{2i}, \hat{f}_1(i_0, i_1) = \hat{f}_{2(8i_0+i_1)+1} = \hat{f}_{2i+1}$$

Using these notations, Equation (1) can be deduced as follows:

$$\begin{aligned}
\hat{f}_0(i_0, i_1) &= \sum_{j=0}^{127} f_0(j_0, j_1)\zeta^{j(2\mathbf{br}_7(i)+1)} \\
&= \sum_{j_0=0}^{15} \sum_{j_1=0}^{7} f_0(j_0, j_1)\zeta^{(8j_0+j_1)(2\mathbf{br}_4(i_0)+32\mathbf{br}_3(i_1)+1)} \\
&= \sum_{j_0=0}^{15} \sum_{j_1=0}^{7} f_0(j_0, j_1)\zeta^{8j_0(2\mathbf{br}_4(i_0)+1)} \cdot \zeta^{j_1(2\mathbf{br}_4(i_0)+1)} \cdot \zeta^{32j_1\mathbf{br}_3(i_1)} \\
&= \sum_{j_0=0}^{15} \sum_{j_1=0}^{7} f_0(j_0, j_1) \cdot \zeta_0(i_0, j_0) \cdot \zeta_1(i_0, j_1) \cdot \zeta_2(i_1, j_1) \tag{5}
\end{aligned}$$

It should be noted that $\zeta^{256k} \equiv 1 \pmod{q}, k \in \mathbb{Z}$ and $\mathbf{br}_7, \mathbf{br}_4, \mathbf{br}_3$ represent the 7-bit, 4-bit and 3-bit bit-reverse function, respectively. The sub-twiddle factors are denoted as follows:

$\zeta_0(i_0, j_0) = \zeta^{8j_0(2\mathbf{br}_4(i_0)+1)}$, $\zeta_1(i_0, j_1) = \zeta^{j_1(2\mathbf{br}_4(i_0)+1)}$ and $\zeta_2(i_1, j_1) = \zeta^{32j_1\mathbf{br}_3(i_1)}$.

Equation (5) will serve as the foundation for our work. While designed for the NTT transform of even-indexed terms, this formula can also be applied to the NTT transform of odd-indexed terms and the INTT transform of both odd and even-indexed terms. We won't delve into those aspects here.

In the subsequent section, we will develop two distinct strategies, a 2-phase scheme and a 3-phase scheme, to efficiently compute NTT based on Equation (5). Please refer to Appendix A for the discussion for INTT.

3.2.2   2-Phase Scheme

It is important to highlight that all the sub-twiddle factors $\zeta_0(i_0, j_0), \zeta_1(i_0, j_1), \zeta_2(i_1, j_1)$ in Equation (5) can be precomputed. A natural idea is to merge as many sub-twiddle factors as possible to avoid unnecessary online computation.

However, it is not possible to merge too many sub-twiddle factors because it would involve too many indices (i.e., i_0, i_1, j_0, j_1), resulting in an extremely large table. In practice, it is acceptable to work with construction methods that involve only three indices, and there are two possible approaches:

1. Offline computing the product of ζ_0 and ζ_1, and denoting it as $\zeta_{01}(i_0, j_0, j_1)$, Equation (5) can be computed in two phases as follows :

$$\hat{f}_0(i_0, i_1) = \sum_{j_1=0}^{7} \underbrace{\left[\sum_{j_0=0}^{15} f_0(j_0, j_1) \cdot \zeta_{01}(i_0, j_0, j_1) \right]}_{\text{Phase 1}} \cdot \left[\zeta_2(i_1, j_1) \right]}_{\text{Phase 2}}$$

2. Offline computing the product of ζ_1 and ζ_2, and denoting it as $\zeta_{12}(i_0, i_1, j_1)$, Equation (5) can be computed in two phases as follows:

$$\hat{f}_0(i_0, i_1) = \sum_{j_1=0}^{7} \underbrace{\left[\sum_{j_0=0}^{15} f_0(j_0, j_1) \cdot \zeta_0(i_0, j_0) \right]}_{\text{Phase 1}} \cdot \left[\zeta_{12}(i_0, i_1, j_1) \right]}_{\text{Phase 2}}$$

The computation method for $\hat{f}_1$ can be derived similarly, and the twiddle factors are the same as those for $\hat{f}_0$. When comparing these two merging approaches, the table for $\zeta_{01}(i_0, j_0, j_1)$ comprises 8 precomputed tables, each of size 16×16, while $\zeta_{12}(i_0, i_1, j_1)$ requires 16 tables. Clearly, the former is more space-efficient; therefore, we have chosen it for our 2-phase scheme.

Considering both $\hat{f}_0$ and $\hat{f}_1$, let us now delve into the implementation of the computational aspects of this scheme using matrix operations as outlined below. Note that $\hat{f}_0$ and $\hat{f}_1$ in the matrix are interwoven in rows, much like the coefficients in a polynomial are arranged in a power-order, alternating between even and odd. Besides, g_0 and g_1 represent intermediate values in the computation, and the elements within the shadow matrix are precomputed.

Phase 1 computes the following terms:

For $i_0 = 0, \ldots, 15$, $j_0 = 0, \ldots, 15$ and $j_1 = 0, \ldots, 7$,

$$g_0(i_0, j_1) = \sum_{j_0=0}^{15} f_0(j_0, j_1) \cdot \zeta_{01}(i_0, j_0, j_1) \tag{6}$$

$$g_1(i_0, j_1) = \sum_{j_0=0}^{15} f_1(j_0, j_1) \cdot \zeta_{01}(i_0, j_0, j_1) \tag{7}$$

$$\Rightarrow \begin{cases} g_0(i_0, 0) = \sum\limits_{j_0=0}^{15} f_0(j_0, 0) \cdot \zeta_{01}(i_0, j_0, 0), & g_1(i_0, 0) = \sum\limits_{j_0=0}^{15} f_1(j_0, 0) \cdot \zeta_{01}(i_0, j_0, 0) \\ \quad\vdots \\ g_0(i_0, 7) = \sum\limits_{j_0=0}^{15} f_0(j_0, 7) \cdot \zeta_{01}(i_0, j_0, 7), & g_1(i_0, 7) = \sum\limits_{j_0=0}^{15} f_1(j_0, 7) \cdot \zeta_{01}(i_0, j_0, 7) \end{cases}$$

The preceding equations will be expanded into a matrix multiplication, as illustrated in Figure 3. In Figure 3, the red box denotes the elements involved in the computation of $g_0(0,0)$ and $g_0(0,7)$, serving as an example. Note that there are eight 16×16 precomputed tables in computation :

$$\boldsymbol{Z}_0 = \{\zeta_{01}(i_0, j_0, 0)\}, i_0 = 0, \ldots, 15, j_0 = 0, \ldots, 15 \tag{8}$$

$$\vdots$$

$$\boldsymbol{Z}_7 = \{\zeta_{01}(i_0, j_0, 7)\}, i_0 = 0, \ldots, 15, j_0 = 0, \ldots, 15 \tag{9}$$

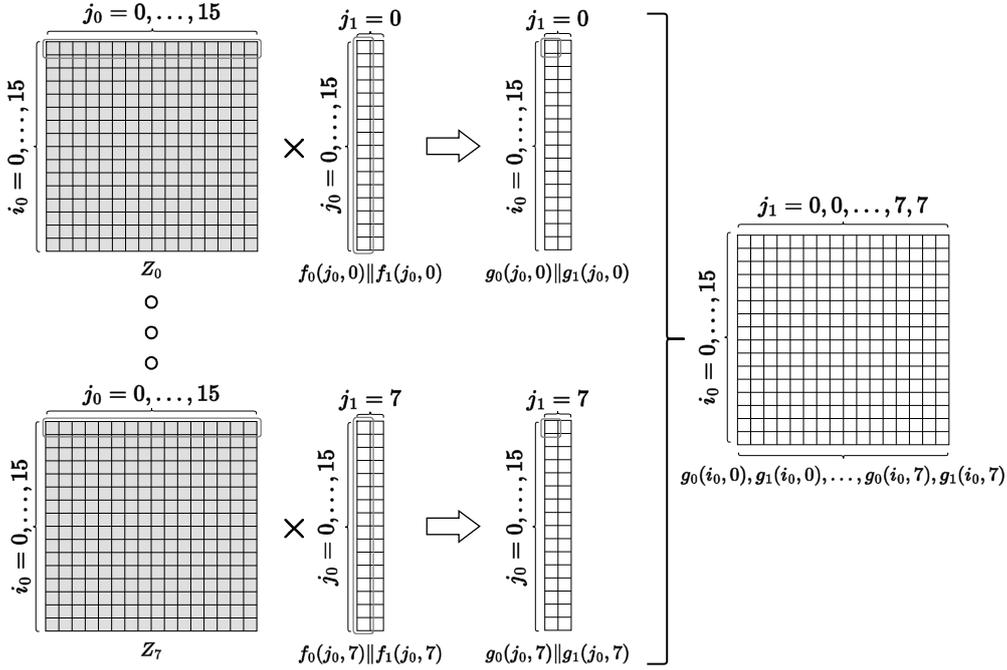**Phase 2** computes the following terms:

Figure 3: The first phase of 2-phase scheme

For $i_0 = 0, \ldots, 15$ and $j_1 = 0, \ldots, 7$,

$$\hat{f}_0(i_0, i_1) = \sum_{j_1=0}^{7} g_0(i_0, j_1) \cdot \zeta_2(i_1, j_1) \tag{10}$$

$$\hat{f}_1(i_0, i_1) = \sum_{j_1=0}^{7} g_1(i_0, j_1) \cdot \zeta_2(i_1, j_1) \tag{11}$$

In Phase 2, there is a significant difference: $\zeta_2(i_1, j_1)$ will be expanded into an 8×8 matrix (notably, $i_1, j_1 \in \mathbb{Z}_8$), while the minimum unit of a Tensor Core is 16×16. Therefore, we further expand this small matrix into a 16×16 sparse matrix, simultaneously completing the computations for $\hat{f}_0$ and $\hat{f}_1$. In Figure 4, the red box highlights the elements involved in the computation of $g_0(0, 0)$, serving as an example.
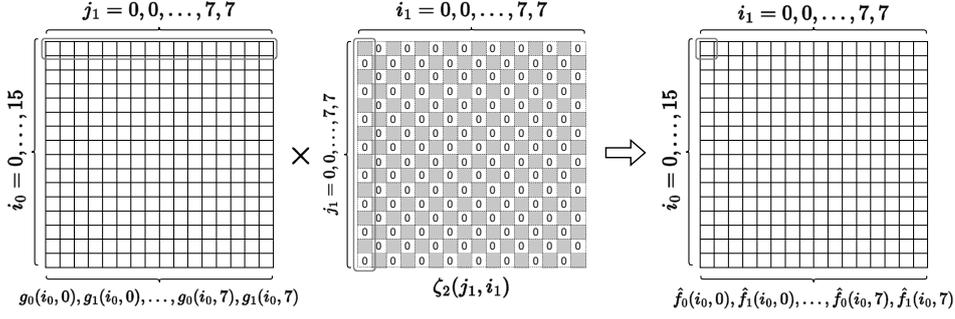
Figure 4: The second phase of 2-phase scheme

3.2.3 3-Phase Scheme

Merging sub-twiddle factors leads to a larger precompute table size, necessitating the introduction of a batching process. Moving forward, let us return to the non-merging approach, which is:

$$
\hat{f}_0(i_0, i_1) = \underbrace{\left\{ \sum_{j_1=0}^{7} \underbrace{\left[\sum_{j_0=0}^{15} f_0(j_0, j_1) \cdot \zeta_0(i_0, j_0) \right]}_{\text{Phase 1}} \cdot \zeta_1(i_0, j_1) \right\}}_{\text{Phase 2}} \cdot \zeta_2(i_1, j_1)
$$
$$
\underbrace{\phantom{\hat{f}_0(i_0, i_1) = \left\{ \sum_{j_1=0}^{7} \left[\sum_{j_0=0}^{15} f_0(j_0, j_1) \cdot \zeta_0(i_0, j_0) \right] \cdot \zeta_1(i_0, j_1) \right\} \cdot \zeta_2(i_1, j_1)}}_{\text{Phase 3}}
$$

The calculation method for $\hat{f}_1$ can be derived analogously.

Taking both $\hat{f}_0$ and $\hat{f}_1$ into consideration, let us now detail how the computational aspects of this scheme can be implemented using matrix operations as follows. Note that $\hat{f}_0$ and $\hat{f}_1$ in the matrix are interwoven in rows, much like the coefficients in a polynomial are arranged in a power-order, alternating between even and odd. Besides, g_0, g_1, h_0, h_1 merely represent intermediate values in the computation, and the elements within the shadow matrix are precomputed.

Phase 1 computes the following terms:

For $i_0 = 0, \ldots, 15, j_1 = 0, \ldots, 7,$

$$
g_0(i_0, j_1) = \sum_{j_0=0}^{15} f_0(j_0, j_1) \cdot \zeta_0(i_0, j_0) \tag{12}
$$

$$
g_1(i_0, j_1) = \sum_{j_0=0}^{15} f_1(j_0, j_1) \cdot \zeta_0(i_0, j_0) \tag{13}
$$

The preceding equations will be expanded into a matrix multiplication, as illustrated in Figure 5. In Figure 5, the red box denotes the elements involved in the computation of $g_0(0,0)$, serving as an example. Different from the 2-phase approach, phase 1 of the 3-phase approach requires only one 16×16 precompute table.

For $i_0 = 0, \ldots, 15, j_1 = 0, \ldots, 7,$

$$
h_0(i_0, j_1) = g_0(i_0, j_1) \cdot \zeta_1(i_0, j_1) \tag{14}
$$

$$
h_1(i_0, j_1) = g_1(i_0, j_1) \cdot \zeta_1(i_0, j_1) \tag{15}
$$

Phase 2 simply computes the Hadamard product, and in Figure 6, the red box denotes the elements involved in the computation of $h_0(0,0)$.
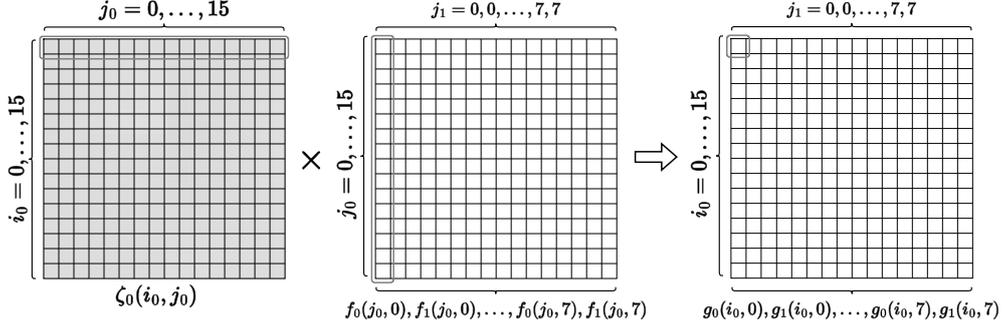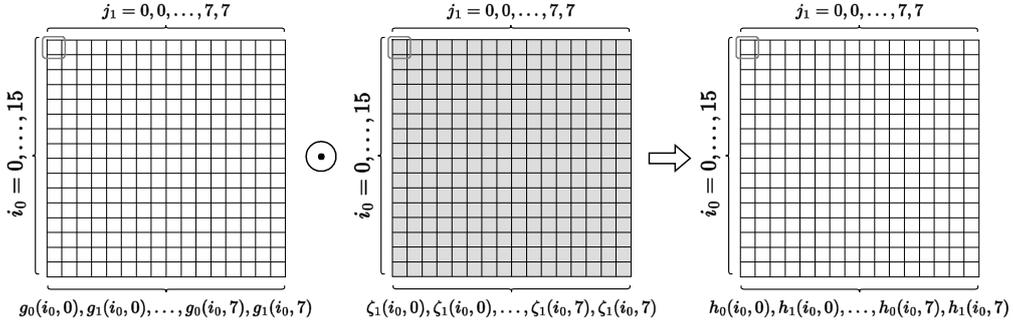
Figure 5: The first phase of 3-phase scheme

Figure 6: The second phase of 3-phase scheme

Phase 3 computes the following terms:
For $i_0 = 0, \ldots, 15$ and $i_1 = 0, \ldots, 7$,

$$\hat{f}_0(i_0, i_1) = \sum_{j_1=0}^{7} h_0(i_0, j_1) \cdot \zeta_2(i_1, j_1) \tag{16}$$

$$\hat{f}_1(i_0, i_1) = \sum_{j_1=0}^{7} h_1(i_0, j_1) \cdot \zeta_2(i_1, j_1) \tag{17}$$

Phase 3 works almost identically with phase 2 of the 2-phase scheme, as shown in Figure 4.

3.2.4 Summary of These 2 Approaches

Decomposing the 256-points NTT schemes into atomic operations (element multiplication and 16×16 matrix multiplication operations), the comparison between 4 NTT schemes are summarized in Table 3. Note that in the first phase of the two-phase scheme, the 16 matrix-vector multiplications fold into 1 matrix multiplication.

As observed in Table 3, the butterfly operation has the lowest computational complexity. However, the scanning-based scheme demonstrates superior performance when implemented with Tensor Cores.

It is crucial to emphasize that while the butterfly operation requires fewer element multiplications, GPU's general instruction set only supports INT32 instructions. Even though each element is relatively small, it still necessitates the use of slower INT32 instructions. In contrast, Tensor Cores can natively support faster INT8 instructions. Therefore, Table 3 provides only a rough comparison, and in GPU implementations, the

Table 3: Comparisons of operations per NTT in different schemes

	NTT		
	# Element multiplications	# Matrix multiplications*	# Precomputed Elements
Butterfly operation used in [GJCC20]	1792	0	128
Scanning-based scheme used in [WZF+22]	0	8	128×128
Proposed 2-phase scheme	0	2	9×256
Proposed 3-phase scheme	256	2	3×256

*Single 16×16 matrix multiplication involves 4096 element multiplications

butterfly operation's advantage in terms of operation count may not be as significant as it appears.

Our proposed iteration-based scheme reduces computational complexity by a quarter (considering only matrix multiplications) and is also highly compatible with Tensor Core acceleration. Furthermore, there is no substantial disparity in computational load between the 2-phase scheme and the 3-phase scheme (the 2-phase scheme involves additional operations to integrate matrix-vector multiplications). Therefore, further investigation is required, and their performance should be empirically compared through implementations.

4   The Proposed NTT Implementation

Section 3 provides the basic principle of the proposed Kyber's NTT. This section will further illustrate how to implement the two schemes with Tensor Cores. This section will highlight three key techniques to overcome the limitations of native Tensor Core instructions:

- **Splitting Kyber's NTT into small Tensor Cores based NTTs**: We will briefly outline how to implement finite field arithmetic for Kyber using the limited precision of Tensor Cores. A standard Kyber's NTT will be separated into several Tensor Cores based NTT computations.

- **A batch implementation for 2-phase NTT**: Based on 2-phase NTT schemes, we will implement a batch NTT that simultaneously processes 8 NTT operations. Additional pre-processing and post-processing steps will be used to transform the requests into matrices that can be accelerated by Tensor Cores.

- **A memory-free 3-phase NTT implementation**: Based on the 3-phase NTT scheme and a thorough exploration of the internal mechanisms of Tensor Cores, we will implement a novel memory-free NTT implementation.

4.1   Multi-precision Presentation and NTT Splitting

As Section 2 describes, Tensor Core performs FMA mixed-precision operation, with low-precision input and high-precision output. For example, on the Ampere architecture, the input can be INT8 (*char*), and the output can be INT32 (*int*).

Wan *et al.* [WZF+22] experimented on the performance of different precision combinations and pointed out that the WMMA matrix multiplication instruction with INT8 precision delivers the highest performance, even though it offers very limited precision. Consequently, depending on the modulus, we can categorize matrix multiplication using the WMMA instruction into two groups: *basic-Mul* for smaller moduli and *split-Mul* for larger moduli.

Additionally, the computation process of *split-Mul* can be integrated with the Karatsuba multiplication [Kar63]. By incurring additional additions, the four multiplications in the schoolbook multiplication can be reduced to three by Karatsuba multiplication. The detailed computation process of *split-Mul* combined with Karatsuba multiplication is illustrated in Figure 7.
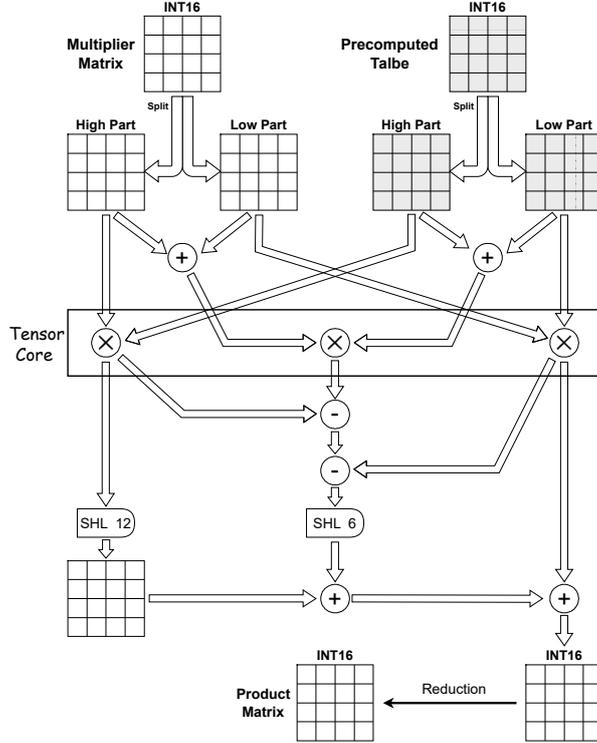

Figure 7: *split-Mul* combined with Karatsuba multiplication

We implement both *split-Mul* combined with schoolbook multiplication and *split-Mul* combined with Karatsuba multiplication, with their performance comparison detailed in Section 6.

4.2   2-phase NTT Implementation with Batch of 8

4.2.1   Pre- and Post- Processing

In the 2-phase NTT scheme, the first step involves splitting a matrix containing 256 polynomial coefficients into 16 vectors and performing 16 matrix-vector multiplications. These 16 vectors cannot be simply merged because the matrices they multiply with are not entirely identical. A straightforward approach would be to calculate all 16 NTT operations simultaneously and then merge the vectors at corresponding positions for computation. However, it's worth noting that vectors at odd positions and their corresponding vectors at even positions use the same twiddle factor matrix. Hence, we only need to compute 8 NTT operations concurrently to fill a 16×16 matrix.

The input for these 8 NTT operations consists of 8 vectors, each containing 256 elements. These matrices are recombined into 8 16×16 matrices, and they are multiplied individually with 8 different precomputed twiddle factor matrices. The resulting product matrices are then decomposed once again to restore the original arrangement.

Before delving into the details of this scheme, let us establish a foundation by assuming that we are about to perform NTT on eight polynomials. For the sake of reference, we denote their respective coefficient matrices as $\boldsymbol{P_0}$ through $\boldsymbol{P_7}$.
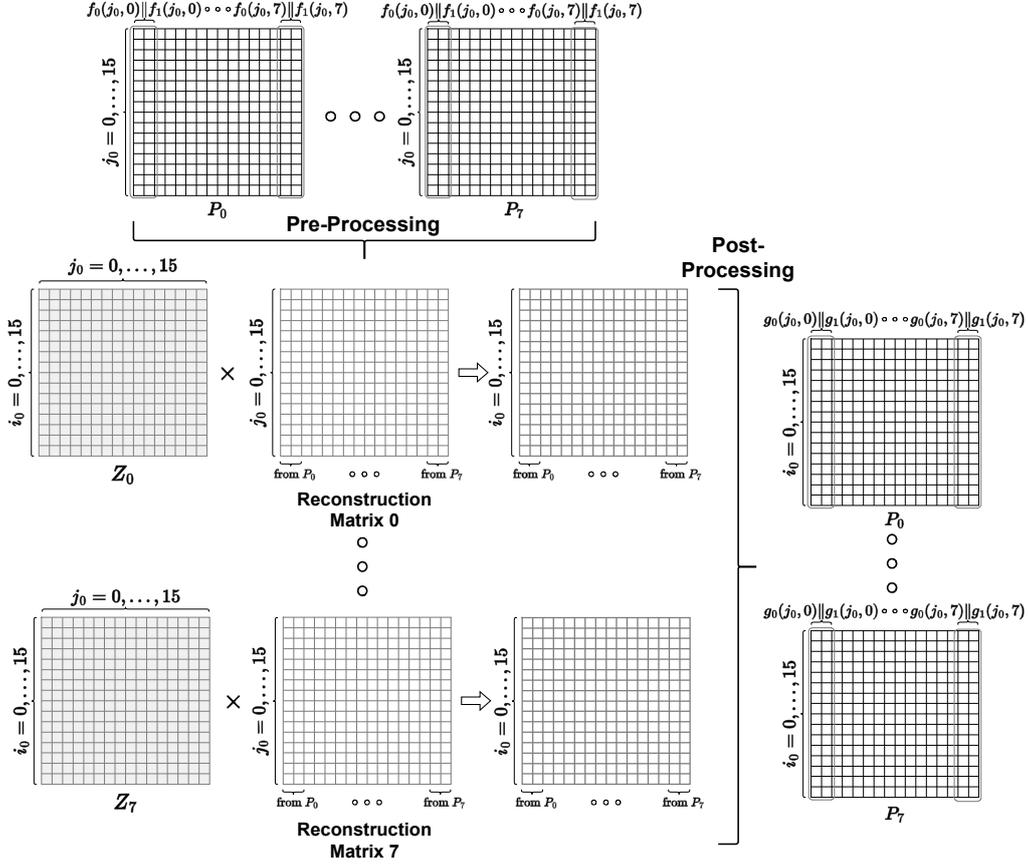

Figure 8: Pre- and post-processing of the 2-phase scheme

The initial step of the batch 2-phase scheme is elegantly depicted in Figure 8. To given context, let's consider the precomputed matrix $\boldsymbol{Z_0}$ (defined by Equation (8)). It corresponds to the first and second columns (highlighted by the red box) spanning from $\boldsymbol{P_0}$ to $\boldsymbol{P_7}$. These columns are thoughtfully consolidated into a restructured matrix labeled as 0, which is subsequently multiplied by $\boldsymbol{Z_0}$ and then reverted to their original positions. The computation relating to $\boldsymbol{Z_7}$ (defined by Equation (9)) can be derived in a similar fashion, illustrated by the blue box.

In the actual implementation, to facilitate the merging of memory accesses, the restructured matrix should be in row-major order rather than column-major order (with elements in the same row arranged consecutively in memory). Therefore, all matrices involved in the first phase need to be transposed, and the left matrix multiplication operations should be changed to right multiplication.

In the second phase of the batch 2-phase scheme, the coefficient matrices associated with polynomials 0 through 7 undergo multiplication with a precomputed matrix formed by ζ_2 (defined in Section 3.2.1) following the computations carried out in the preceding stage. As a result, we obtain the NTT domain representations for all eight polynomials simultaneously.

It is worth noting that the first matrix multiplication in the scheme requires pre-

processing and post-processing. However, the pre-processing can be omitted when the NTT object is a polynomial obtained by sampling.

4.2.2 Implementation Summary

The resulting implementation method is illustrated in Figure 9.
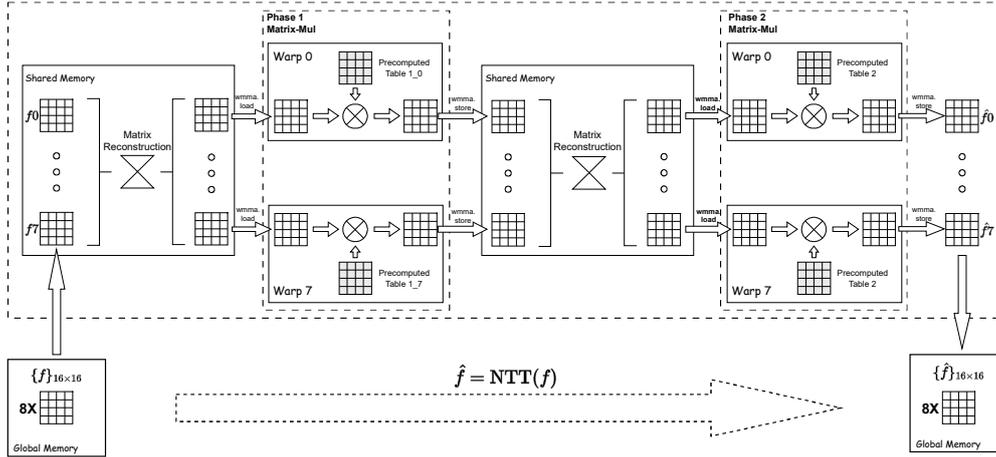

Figure 9: Details of the entire process of 2-phase NTT

In summary, in phase 1, 8 warps are each responsible for multiplying by 8 different precomputed matrices, and in Phase 2, each warp multiplies by the same precomputed matrix.

Batch processing is constrained by phase 1, requiring each block to simultaneously contain 8 warps, which means a block size with a multiple of 256. In terms of memory, shared memory is used for reorganizing matrices and facilitating data exchange among 8 warps after multiplication. Since the precomputed tables are fixed, we can load them all at once when loading the binary, and there is no need for additional memory writes when using the precomputed tables subsequently.

4.3 A Memory-free 3-phase NTT Implementation

4.3.1 Limitation of the Standard Interfaces of Tensor Cores

Whether it is the 2-phase or 3-phase approach, we must execute two WMMA instructions sequentially, with the output of the first instruction serving as the input for the second.

Listing 1 provides a detailed explanation of the standard method for invoking the WMMA API. After computation, the results need to be stored in (global or shared) memory using `wmma::store_matrix_sync` before proceeding with the next operation. However, this step appears to be unnecessary in theory if we recursively use the output as input, as it should be possible to fully reuse the `wmma::fragment` variable. The challenge lies in the fact that `wmma::fragment` is type-specific, being one of `wmma::matrix_a`, `wmma::matrix_b`, or `wmma::accumulator`. Unfortunately, `wmma::accumulator` cannot be used as input, and its type cannot be converted. The only way out is to "store" it in memory and then "load" it into a new `wmma::fragment`.

We illustrate the limitation of the standard WMMA interfaces in Figure 10. It is evident that 4 load/store operations are entirely unnecessary. While these surplus memory transfers do not significantly impact the 2-phase approach due to the necessity of pre- and

post-processing in memory, they become entirely redundant in the case of the 3-phase approach.

Listing 1: Standard invocation methods of WMMA API

```
__global__ void wmma_ker(int8_t *a, int8_t *b, int *c) {
  // Declare the fragments
  wmma::fragment<wmma::matrix_a, 16, 16, 16,
                 int8_t, wmma::row_major> a_frag;
  wmma::fragment<wmma::matrix_b, 16, 16, 16,
                 int8_t, wmma::row_major> b_frag;
  wmma::fragment<wmma::accumulator, 16, 16, 16, int> c_frag;
  // Initialize the output to zero
  wmma::fill_fragment(c_frag, 0);
  // Load the inputs
  wmma::load_matrix_sync(a_frag, a, 16);
  wmma::load_matrix_sync(b_frag, b, 16);
  // Perform the matrix multiplication
  wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
  // Store the output
  wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

This issue has piqued our interest. Next, we will delve deeper into the actual structure of `wmma::fragment` to see if we can achieve functionalities beyond what standard interfaces offer.
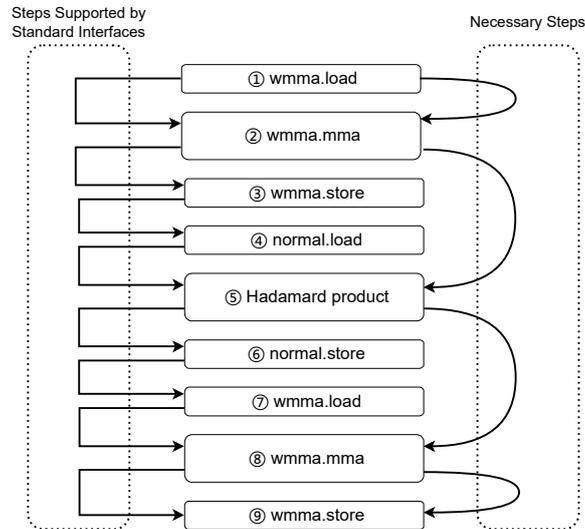
Figure 10: The limitation of the standard WMMA interfaces

4.3.2   Reversing the Fragment Structure

`wmma::fragment` is essentially a data structure that stores the elements of a 16×16 matrix separately in a warp's 32 threads. In reality, each thread will preserve $\frac{16 \times 16}{32} = 8$ registers. If we can manually manipulate these registers, it becomes possible to perform two consecutive matrix multiplications without the need to store them back in memory. However, this idea faces two challenges:

- First, `wmma::fragment` itself is an abstract representation, and its internal registers are transparent to developers. We do not have a direct way to access these register resources via standard WMMA interfaces.

- Second, the storage arrangement within the warp for `wmma::fragment` is unknown. Regarding this point, the CUDA C++ Programming Guide (Release 12.2) states, "the mapping of matrix elements into fragment internal storage is unspecified and subject to change in future architectures" (§10.24.1 in [NVIa]).

Fortunately, while the first issue cannot be resolved through standard interfaces, we can achieve this through the PTX ISA assembly code. Through examination of the original code using `cuobjdump`, we discovered that WMMA instructions, when actually executed, directly use registers rather than `wmma::fragment` as parameters.
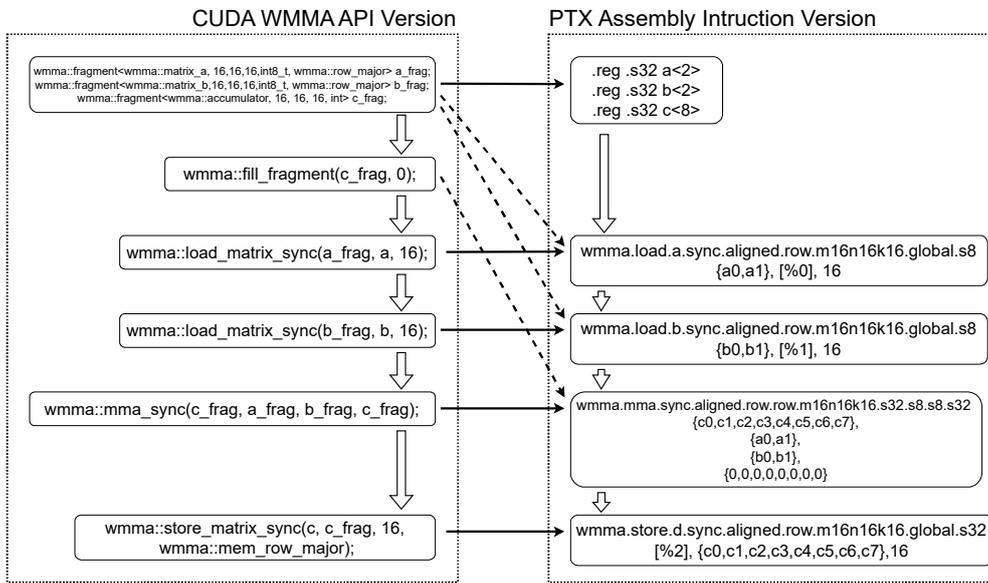
Figure 11: From CUDA WMMA APIs to PTX assembly instructions

The relationship between `wmma::fragment` and registers is shown in Figure 11. As a result, we can delve into the data arrangement within the warp for `wmma::fragment`. The following observation is primarily based on the Jetson AGX Orin platform, with extended verification conducted on the RTX 3080 platform.

Data layout caused by `wmma.load`: A continuous memory address space containing 256 polynomial coefficients can be interpreted as a 16×16 matrix, denoted as $\boldsymbol{M}$, with coefficients arranged sequentially, starting from row 0. When using `wmma.load`, it loads 256 data points into 32 threads within a warp, with each thread holding 8 data points. This data arrangement within these 32 threads can also be considered as the 16×16 matrix $\boldsymbol{W}$, where the first row comprises 16 data points from thread 0 and thread 1, and so forth. We subsequently refer to the transformation that converts the matrix $\boldsymbol{M}$ to the matrix $\boldsymbol{W}$ as the `wmma.load` transformation ($\boldsymbol{W}$=`wmma.load`($\boldsymbol{M}$)), as depicted in Figure 12. Please be aware that this transformation is only correct when loading as matrix $\boldsymbol{A}$; a different transformation occurs when loading as matrix $\boldsymbol{B}$ (refer to Figure 2 for the definitions of the matrix $\boldsymbol{A}$ and $\boldsymbol{B}$).

Data layout caused by `wmma.store`: Similarly, we consider the data arrangement within the warp before `wmma.store` as the matrix $\boldsymbol{W}$ and the data arrangement in memory after `wmma.store` as the matrix $\boldsymbol{M}$. We refer to the layout transformation from matrix $\boldsymbol{W}$
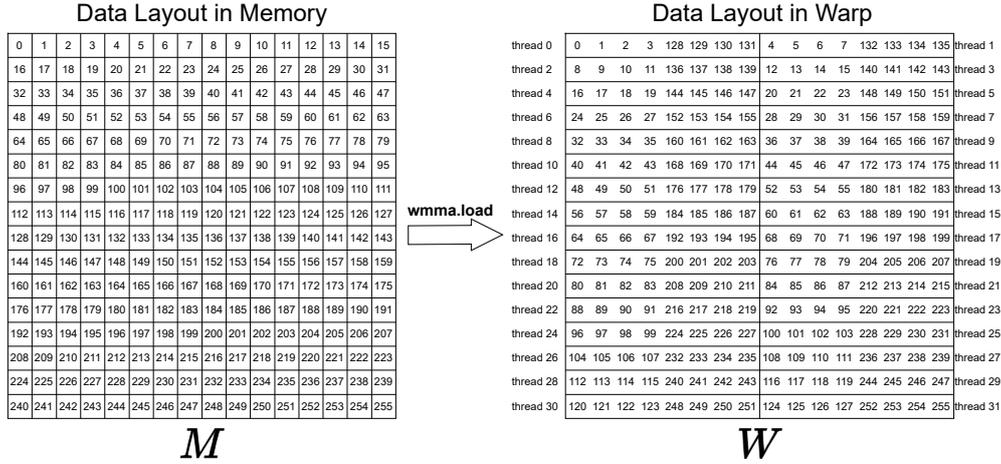
Figure 12: Fragment layout across 32 thread within a warp after `wmma.load`

to matrix M as the `wmma.store` transformation (M= `wmma.store`(W)), as illustrated in Figure 13.
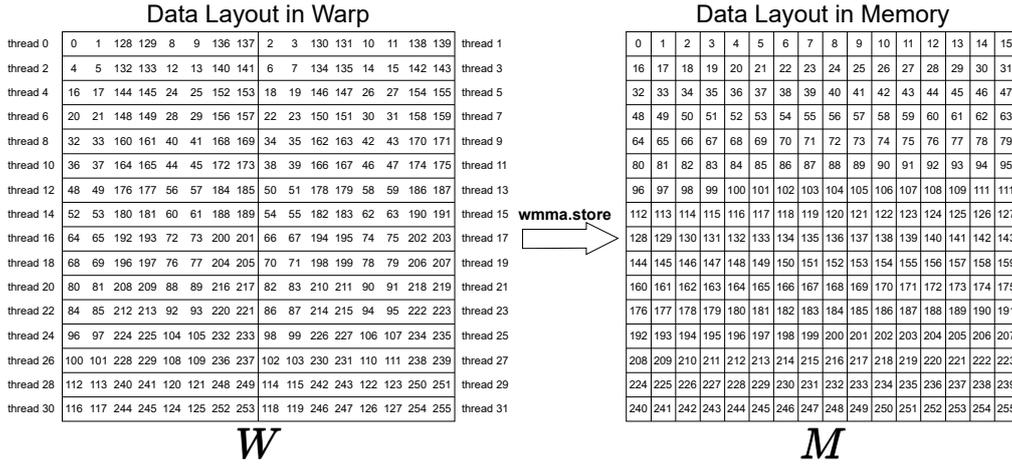
Figure 13: Fragment layout across 32 thread within a warp after `wmma.store`

4.3.3   Processing the Data Layout Displacement

Based on the exploration in Section 4.3.2, a rather challenging issue arises due to data layout displacements between memory and registers across the warp. What complicates matters further is our plan to perform two consecutive matrix multiplications, where the result of the first multiplication serves as the input for the second. However, the data layout displacements caused by `wmma.store` and `wmma.load` are entirely different.

For instance, when used as input, after `wmma.load`, thread 0 stores elements with indices 0, 1, 2, 3, 128, 129, 130, and 131. However, when used as output, before `wmma.store`, thread 0 stores elements with indices 0, 1, 128, 129, 8, 9, 136, and 137. Therefore, before performing the next matrix multiplication, we must rearrange the elements stored by the threads from the layout at the output to the layout at the input.

As a result, the core issue lies in finding a cost-effective method to adjust the internal data layout of the warp, ensuring the correct calculation of the following matrix multiplication. One seemingly straightforward approach is to use CUDA `shuffle` instruction or shared memory to rearrange the output data back into the required input layout. However, this approach contradicts our original intention, as `shuffle` instructions have efficiency comparable to INT32 multiplication, and shared memory introduces unnecessarily complicated matters.

Upon careful analysis of the data arrangement, a crucial observation is that we only need to ensure that the data included in each row of the matrix, as it is actually represented, remains consistent with the original representation, and the order within each row is irrelevant. The reason for this simplicity is that we are performing matrix operations, and in the context of matrices used for left multiplication, the matrices participate in the operations on a row-by-row basis. Changing the order within the rows only requires us to alter the column order of the corresponding right multiplication matrix (i.e., the precomputed table of twiddle factors).

Data Layout in Warp after Shuffle — W

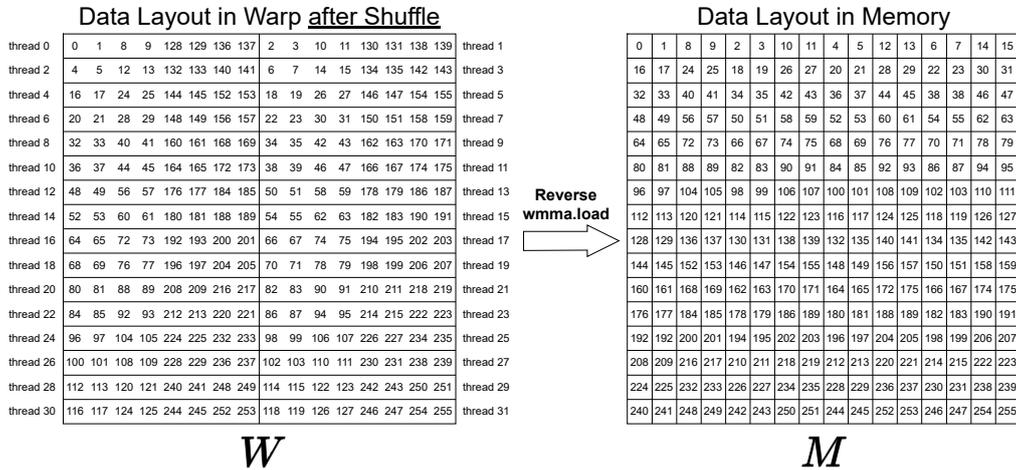| thread									thread								
thread 0	0	1	8	9	128	129	136	137	thread 1	2	3	10	11	130	131	138	139
thread 2	4	5	12	13	132	133	140	141	thread 3	6	7	14	15	134	135	142	143
thread 4	16	17	24	25	144	145	152	153	thread 5	18	19	26	27	146	147	154	155
thread 6	20	21	28	29	148	149	156	157	thread 7	22	23	30	31	150	151	158	159
thread 8	32	33	40	41	160	161	168	169	thread 9	34	35	42	43	162	163	170	171
thread 10	36	37	44	45	164	165	172	173	thread 11	38	39	46	47	166	167	174	175
thread 12	48	49	56	57	176	177	184	185	thread 13	50	51	58	59	178	179	186	187
thread 14	52	53	60	61	180	181	188	189	thread 15	54	55	62	63	182	183	190	191
thread 16	64	65	72	73	192	193	200	201	thread 17	66	67	74	75	194	195	202	203
thread 18	68	69	76	77	196	197	204	205	thread 19	70	71	78	79	198	199	206	207
thread 20	80	81	88	89	208	209	216	217	thread 21	82	83	90	91	210	211	218	219
thread 22	84	85	92	93	212	213	220	221	thread 23	86	87	94	95	214	215	222	223
thread 24	96	97	104	105	224	225	232	233	thread 25	98	99	106	107	226	227	234	235
thread 26	100	101	108	109	228	229	236	237	thread 27	102	103	110	111	230	231	238	239
thread 28	112	113	120	121	240	241	248	249	thread 29	114	115	122	123	242	243	250	251
thread 30	116	117	124	125	244	245	252	253	thread 31	118	119	126	127	246	247	254	255

Reverse wmma.load →

Data Layout in Memory — M

0	1	8	9	2	3	10	11	4	5	12	13	6	7	14	15
16	17	24	25	18	19	26	27	20	21	28	29	22	23	30	31
32	33	40	41	34	35	42	43	36	37	44	45	38	39	46	47
48	49	56	57	50	51	58	59	52	53	60	61	54	55	62	63
64	65	72	73	66	67	74	75	68	69	76	77	70	71	78	79
80	81	88	89	82	83	90	91	84	85	92	93	86	87	94	95
96	97	104	105	98	99	106	107	100	101	108	109	102	103	110	111
112	113	120	121	114	115	122	123	116	117	124	125	118	119	126	127
128	129	136	137	130	131	138	139	132	133	140	141	134	135	142	143
144	145	152	153	146	147	154	155	148	149	156	157	150	151	158	159
160	161	168	169	162	163	170	171	164	165	172	173	166	167	174	175
176	177	184	185	178	179	186	189	180	181	188	189	182	183	190	191
192	192	200	201	194	195	202	203	196	197	204	205	198	199	206	207
208	209	216	217	210	211	218	219	212	213	220	221	214	215	222	223
224	225	232	233	226	227	234	235	228	229	236	237	230	231	238	239
240	241	248	249	242	243	250	251	244	245	252	253	246	247	254	255

Figure 14: Reordered data layout for memory-free implementation

Building on this insightful observation, we have devised an innovative approach to tackle the data displacement challenge. We reconfigure the data layout within the adjusted warp, as illustrated in Figure 14. The resulting matrix can be obtained by performing matrix multiplication between the permutation matrix P (as depicted in Figure 15) and the order matrix (found in the rightmost matrix of Figure 13).

Remarkably, after this adjustment, although the order may seem shuffled, each row i of the matrix contains elements spanning from $16i$ to $16i + 15$. There remains just one task to complete: the corresponding precomputed table requires a right multiplication by P^{-1}.

The surprising thing is that the implementation of the above method is very neat. Notably, after `wmma.mma`, 256 products are scattered across 32 threads, with 8 `b32` registers in each thread. Let us label them sequentially as $r0$, $r1$, $r2$, $r3$, $r4$, $r5$, $r6$, and $r7$. To address the data layout confusion resulting from omitting `wmma.store` and `wmma.load`, we simply adjust the register order in all threads of a warp from ($r0$, $r1$, $r2$, $r3$, $r4$, $r5$, $r6$, $r7$) to ($r0$, $r1$, $r4$, $r5$, $r2$, $r3$, $r6$, $r7$) (swap registers $r2$, $r3$ with $r4$, $r5$). This way, we effectively perform the operation of multiplying the permutation matrix P. Regarding the corresponding precomputed table, since it is precomputed, we only need to make adjustments in the code to accommodate this change.

Note that the matrix obtained through shuffling the registers among threads can only

Figure 15: Permutation matrix P

serve as matrix $\boldsymbol{A}$ in subsequent matrix multiplications. This is because the combined transformation of storing and loading matrix $\boldsymbol{B}$ cannot be implemented by shuffling registers (refer to Figure 2 for the definitions of the matrix $\boldsymbol{A}$ and $\boldsymbol{B}$).

4.3.4   Implementation Summary

The final implementation consists of three phases and does not require additional memory; it is entirely register-based, as shown in Figure 16.

Because it does not impose any batch requirements, a block can include any number of warps, as opposed to the 2-phase scheme that requires a multiple of 8. Moreover, there is no need for pre-processing, post-processing, or data reorganization, and the data input and output fully comply with Kyber's requirements.

Figure 16: Details of the entire process of 3-phase NTT

5   Extending Proposed NTT Schemes to a Full Kyber Implementation

In this section, we will demonstrate how to extend our proposed NTT schemes to a complete Kyber implementation and elucidate some non-trivial optimization techniques.

5.1   Overall Architecture

Our full Kyber implementation follows the SIMT mode, where each thread handles one Kyber KeyGen/Enc/Dec instance. While the specific procedures may vary slightly for different phases, the high-level overview is depicted in Figure 17.
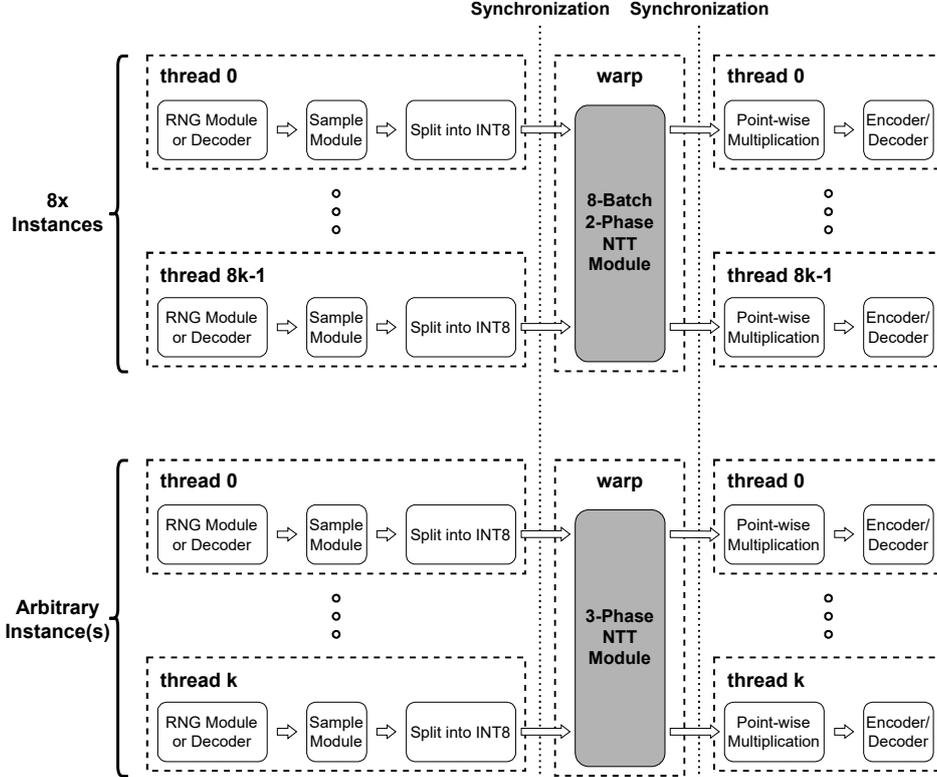
Figure 17: Our overall implementation of Kyber

This architecture is throughput-oriented. In fact, previous works either implemented NTT with a single thread [GJCC20] or required a high degree of batching (e.g., a batch size of 16 was necessary for [WZF$^+$22]). This have to be throughput-oriented because a single NTT operation could not fully leverage the parallel capabilities of the GPU. In contrast, our proposed 3-phase NTT scheme can complete a single NTT with a warp (i.e., 32 threads), and 2-phase NTT scheme requires a batch size of 8.

By lowering the concurrency requirements, we have the flexibility to choose whether our solution is throughput-oriented or latency-oriented. Since our primary focus is on NTT acceleration and for direct comparison with existing literature, we have implemented full Kyber in a more straightforward throughput-oriented manner. In future work, we plan to explore a latency-oriented implementation.

It is worth noting that the KeyGen step, as it does not require input requests, can continuously generate key pairs into a pool in an offline manner and thus is well-suited for the throughput-oriented architecture. Additionally, some works [PZZ$^+$17, WZG$^+$21, BZW$^+$23] have introduced an intermediate layer for caching requests to fully harness the potential of throughput-oriented implementations.

The specific workings of the 8-batch 2-phase NTT module and 3-phase NTT module are shown respectively in Figure 9 and Figure 16 and the precomputed tables can be found in appendix B. Note that the inputs and outputs of these two modules are sequential

coefficients; there is no need for reordering (the specific order required by the 2-phase NTT scheme is completed within the module itself).

Furthermore, since the 8-batch 2-phase NTT module operates in batches of 8, the number of instances input to this module must be a multiple of 8. Finally, since the modules before and after the NTT module operate independently per thread, and the NTT module is executed collaboratively by 32 threads (1 warp). When the program needs to perform NTT, it will synchronize between threads in the same thread block, and then input the data into the NTT module.

5.2 Other Implementation Details

This section details some worth-noting techniques in the full Kyber implementation.

5.2.1 Coalescing Memory Access

Ensuring coalesced memory access is vital for optimizing performance in applications developed on GPU architectures. It involves efficiently using the memory bus to minimize the number of memory accesses, which can significantly reduce the time spent on memory read and write operations. For example, Orin's memory bus width is 128 bits. This means that for individual read and write operations on INT8, INT16, and INT data types on Orin, there would be a wastage of 93.75%, 87.5%, and 75% of the data carrying capacity, respectively. In other words, a substantial portion of the memory bus's capacity remains unused when dealing with these data types. However, in practice, GPU compilers are designed to allocate individual instructions for load operations and store operations tailored to various data precisions. This optimization helps mitigate the wastage of memory bus capacity by aligning data transfers more closely with the actual data type requirements, thereby improving memory access efficiency.

For instance, consider the code in Listing 2, which represents the *poly_tobytes* function provided by the Kyber designers in the reference implementation for serializing polynomials. In this context, the code that reads the variable *coeffs*[i] may be compiled into a form like `ld.global.s16 in0, [%0]`, and the code that stores the variable $r[i]$ might be compiled into a form like `st.global.u8 [%1], out0`.

To optimize memory access, we complete the read and write operations using constructs like:

`ld.global.v4.s32 {in0,in1,in2,in3}, [%0]` and
`st.global.v4.s32 [%1],{out0,out1,out2,out3}`.

This approach retrieves data in chunks, which is then split based on the required precision. In this specific case, reading 32 `coeffs[i]` and writing 48 `r[i]` ensures that each read/write operation aligns with multiples of 128 bits. This alignment maximizes memory access efficiency, making better use of the available memory bus capacity.

Listing 2: Function *poly_tobytes* in reference implementation of Kyber

```
typedef struct{
  int16_t coeffs[KYBER_N];
} poly;
void poly_tobytes(uint8_t r[KYBER_POLYBYTES], poly *a){
  unsigned int i;
  uint16_t t0, t1;
  ...
  for(i=0;i<KYBER_N/2;i++){
    t0 = a->coeffs[2*i];
    t1 = a->coeffs[2*i+1];
    r[3*i+0] = (t0 >> 0);
```

```
    r[3*i+1] = (t0 >> 8) | (t1 << 4);
    r[3*i+2] = (t1 >> 4);
  }
}
```

5.2.2 SHA-3

The symmetric primitives of Kyber are also a performance bottleneck for Kyber, with the specific computations belonging to the SHA-3 family, at the heart of which is the StatePermute function f. The f function consists of a total of five processes (θ,ρ,π,ι and χ), of which ρ,π,ι require table look-ups.

We meticulously translated the computation process of the f function into assembly code, unrolling each loop. Steps requiring table lookups were embedded directly into the assembly instructions using immediate values, minimizing memory access requirements during SHA-3 computation.

5.2.3 Lazy Reduction

When performing NTT on polynomials sampled from the CBD distribution, note that the coefficients are expressed in less than 4 bits. The data in all precalculated tables is less than or equal to 12 bits (modulus $q = 3329$), which means that the `b32` register of the result after matrix multiplication and Hadamard product can still be contained. Therefore, instead of reducing after matrix multiplication immediately, we can reduce after the Hadamard product.

6   Performance Evaluation & Discussion

In this section, we present our evaluation results, including the performance of the NTT module and Kyber-512, Kyber-768, Kyber-1024, and perform a comparative analysis with related works. All codes are written in CUDA C++, which uses PTX inline assembly to optimize key operations and intensive memory access operations.

Table 4 details the specifications of the targeted platforms. To simplify matters, we will refer to these two platforms as "Orin" and "R3080" respectively. Notably, the Jetson AGX Orin is an embedded platform which offers adjustable power modes at 15W, 30W, 50W, and 60W, allowing for tailored performance optimization. And we will evaluate Kyber across these modes.

It should be noted that our optimizations are primarily tailored for the Orin platform, and evaluations based on the R3080 are principally aimed at facilitating comparisons with other works within the same platform. Therefore, the parameters used, such as block size, while being optimal for Orin, have been adopted as-is for the R3080 and may not represent the most optimal configuration for the latter.

6.1   Results of NTT/INTT

Our evaluation starts with the testing for the peak throughput of the 2-phase phase and 3-phase NTT/INTT for input types of both INT8 and INT16 on Orin and R3080. Given that the 2-phase scheme operates in batches of 8, a multiple of 8 warps is required for collaborative computation, making the block size a minimum of 256. Since the INTT always deals with data with full precision, we only evaluate implementations where the input type is INT16. The experimental results are listed in Table 5.

As evident from Table 5 and consistent with the design expectations, the performance of NTT and INTT is essentially comparable. When comparing the 2-phase NTT/INTT

Table 4: Experimental Platform Configuration

	Orin	R3080
GPU	NVIDIA Jetson AGX Orin	NVIDIA GeForce RTX 3080
Clock Frequency	408MHz (15W)/612MHz (30W) 816MHz (50W)/1.3GHz (60W)	1.8GHz
# Streaming Multiprocessor (SM)	6 (15W)/8 (30W)/16 (50&60W)	68
Shared Memory per SM	48KB	100KB
# CUDA Core per SM	128	128
# Tensor Core per SM	4	4
OS	Ubuntu 20.04.6 LTS	Ubuntu 22.04 LTS
CUDA Toolkit	v11.4	v11.8

Table 5: The peak throughput of NTT/INTT on Orin (the data inside the parentheses represents the throughput on R3080), $n = 256$; For Orin (60W), **Total_case**=16384, **Grid_size**=32; For R3080, **Total_case**=69632, **Grid_size**=136

Operations	NTT				INTT	
Input precision	INT8		INT16		INT16	
Block size	128	256	128	256	128	256
2-Phase (Mops)	/	77.1 (395.4)	/	54.6 (316.3)	/	54.2 (314.9)
3-Phase (Mops)	43.3 (439.8)	116.3 (485.6)	40.1 (381.9)	101.7 (414.0)	38.9 (381.8)	100.4 (410.2)

scheme and the 3-phase NTT/INTT scheme, an advantage of the former is that it is universally applicable to platforms that support efficient matrix operations. However, this scheme is slower than the 3-phase scheme in performance, especially in Orin (which we particularly optimized for), due to the data exchange between warps in the post-processing of the first phase, specifically during the matrices reconstruction, as illustrated in Figure 8. In contrast, the latter offers higher performance, involves fewer precomputed tables, and eliminates the need to allocate temporary memory for storing intermediate data. Since the 3-phase NTT/INTT scheme has better performance, we will adopt this scheme in the subsequent comparisons with other works targeting NTT and Kyber.

Notably, we made an unusual discovery during our experiments when testing NTT/INTT: utilizing column-major WMMA load/store is approximately 10% faster than using row-major access. Consequently, we adjusted our implementation to prioritize reading and writing polynomial coefficients in a column-major fashion wherever possible.

6.1.1 Comparison of Throughput under Peak Concurrency Conditions.

When comparing with other GPU-based research results, we consider the maximum level of parallelism and compute the average time cost for each instance, as shown in Table 6. In addition, we implement *split-Mul* (Section 4.1) utilizing two methods: schoolbook multiplication and Karatsuba multiplication.

When performing schoolbook multiplication, computing the product of double-precision numbers (e.g., $a_1\beta + a_0$ and $b_1\beta + b_0$) requires 4 multiplication operations (a_0b_0, a_0b_1, a_1b_0, and a_1b_1) and 1 addition. As illustrated in Figure 7, in the case of Karatsuba multiplication, with the cost of 3 additional additions/subtractions, it only requires 3 multiplication operations (a_0b_0, a_1b_1, and $(a_0 + a_1)(b_0 + b_1)$). In our implementation, the main reason for the comparable performance between schoolbook and Karatsuba multiplication lies in the fact that multiplication utilizes fast Tensor Cores while addition and subtraction operations uses general CUDA cores, which may offset the advantage in the number of multiplications. But overall, Karatsuba multiplication still demonstrates slightly better performance than the schoolbook approach.

Table 6: Comparison average time of `polyvec_ntt` in Kyber-1024, $n = 256, k = 4$

	Device	Architecture	Average cost (ns)
Gupta *et al.* [GJCC20]	R3080	Ampere	107.81 *
Wan *et al.* [WZF$^+$22]	R3080	Ampere	16.65
Ours	R3080	Ampere	9.36° (Schoolbook) 8.61° (Karatsuba)

  * The code in [GJCC20] is downloaded from https://github.com/nainag/PQC and tested on R3080.

  ° This performance is obtained by continuously testing 100 times with the (grid size, block size) set to (136, 512) and averaging the results.

Table 6 demonstrates our approach achieving nearly double the speed compared to the current state-of-the-art implementations. In fact, the performance data of our work lists in Table 6 is not the maximum achievable, and with increasing (grid size, block size), the performance can be further improved.

However, our method boasts just one-fourth of the computational complexity of Wan *et al.*'s [WZF$^+$22], as evident in Table 3, suggesting the potential for a fourfold performance boost. This discrepancy is primarily due to precision decomposition and reduction constraints between the two sequential matrix multiplications, limiting it from reaching its theoretical potential. Moreover, upon consultation with the authors of [WZF$^+$22], when measuring, their input and output are pre-arranged with even and odd terms separated, incurring additional time consumption when extending to a complete Kyber implementation. In contrast, our 3-phase scheme does not necessitate data reordering. While the time cost is minimal, it is one of the potential reasons why our performance falls short of expectations.

Another factor is that the implementation by Wan *et al.* [WZF$^+$22] considers maximum concurrency. Their scanning-based method, which continuously computes a substantial number of matrix multiplications, optimally utilizes Tensor Core's pipeline, ultimately achieving peak performance when amortized.

6.1.2 Comparison of Execution Latency

In reality, there may not always be such a high level of concurrency. Therefore, we also consider a challenging scenario for GPU-based implementation where only one `polyvec_ntt` operation is executed at a time. In this context, we evaluate the latency of executing a single `polyvec_ntt` operation to make comparisons across different types of devices, as illustrated in Table 7.

For `polyvec_ntt`, we leverage the strength of the 3-phase scheme, where each warp computes the NTT of an individual polynomial independently. By setting the block size to 128, we make sure there are 4 warps in a grid. In this manner, one grid computes one set of `polyvec_ntt`, allowing us to evaluate the delay of computing a single `polyvec_ntt` when $k = 4$. The enhancement in performance, as evidenced by our evaluation, can be attributed to

Table 7: Comparison of execution latency of single `polyvec_ntt` of Kyber-1024, $n = 256, k = 4$

	Device	Latency (μs)
Alkim *et al.* [AEL$^+$20]§	Xilinx Artix-35T @59.2 MHz	464$^{\circ\star}$
Ma *et al.* [MWB21]§	FPGA @227.27 MHz	2.389*
Becker *et al.* [BHK$^+$22]§	ARM Cortex-A72 @1.5GHz	3.2$^{\#\star}$
Ours	Orin @1.3GHz	0.35†
	R3080 @1.8GHz	0.259†

§ The implementation result of this work is obtained using one core of processor.
$^\circ$ Derived from the "custom" implementation [AEL$^+$20], where a NTT requires 6868 cycles for a single
   polynomial.
* The authors in [MWB21] reported that a NTT requires 543 cycles for Kyber-1024.
$^\#$ The authors in [BHK$^+$22] indicated that an NTT necessitates 1200 cycles for a single polynomial.
   Moreover, it is imperative to note that their NTT is 32-bit.
* The comparison pertains to `polyvec_ntt` for Kyber-1024, thus the latency for NTT of one polynomial
   should be multiplied by 4.
† In fact, we measure the execution latency for 16 instances of `polyvec_ntt` in Orin and 68 instances
   in R3080. Because executing this many instances of `polyvec_ntt` and executing a single `polyvec_ntt`
   yield nearly the same overall execution time. Grid_size=16, Block_size=128 is set for Orin and
   Grid_size=68, Block_size=128 is set for R3080

several pivotal aspects as follows: 1) The remarkable technique of Tensor Core in executing matrix computations bestows our scheme with a computationally efficient backbone; 2) Our approach smartly decomposes NTTs into matrix multiplications, shifting the computational load of the NTTs onto the Tensor Core to leverage its strong computational capabilities effectively; 3) Our 3-phase scheme enables simultaneously computing of multiple NTTs by utilizing parallel computing platforms.

In fact, it must be emphasized that the implementations [AEL$^+$20, MWB21, BHK$^+$22] based on FPGA and embedded devices in Table 7 are single-core, and their design goals prioritize low latency or low cost. The cross-platform comparison we conducted here is solely to illustrate that our proposed approach not only achieves high throughput but also demonstrates the potential for low-latency implementations on parallel platforms that do not inherently excel in terms of latency.

6.2 Results of Kyber

We conduct continuous testing of Kyber security strengths, namely Kyber-512 ($k = 2$) on the embedded platform Orin and Kyber-768 ($k = 3$), Kyber-1024 ($k = 4$) on the desktop platform R3080, performing 500 iterations to calculate average performance, as illustrated in Figures 18.

For Kyber1024, on Orin, the peak KeyGen, Enc, and Dec performance is achieved when the (grid size, block size) is set to (32, 128). On R3080, the best KeyGen and Enc performance is attained when the (grid size, block size) is set to (136, 128), while, for Dec, the highest performance is achieved with (grid size, block size) set to (68, 128). Additionally, from a latency perspective, the lowest latency is observed when the (grid size, block size) is set to (16/68, 128). The latencies for (16/68, 256) and (32/136, 128) are approximately twice that of (32/68, 128), while (32/136, 256) is approximately four times that of (16/68, 128) on Orin or R3080 respectively.

From Figure 18, the computational performance of R3080 is approximately five times that of Orin and it is evident that the throughput for KeyGen and Enc is relatively low, whereas the throughput for Dec is significantly higher. This is because both KeyGen (shown in Algorithm 1) and Enc (shown in Algorithm 2) involve the computationally intensive XOF and PRF functions, which are based on hashing, to generate matrix A and to sample multiple polynomials, respectively. On the other hand, Dec (shown in Algorithm 3) only involves decoding and polynomial computations, which are memory access-intensive
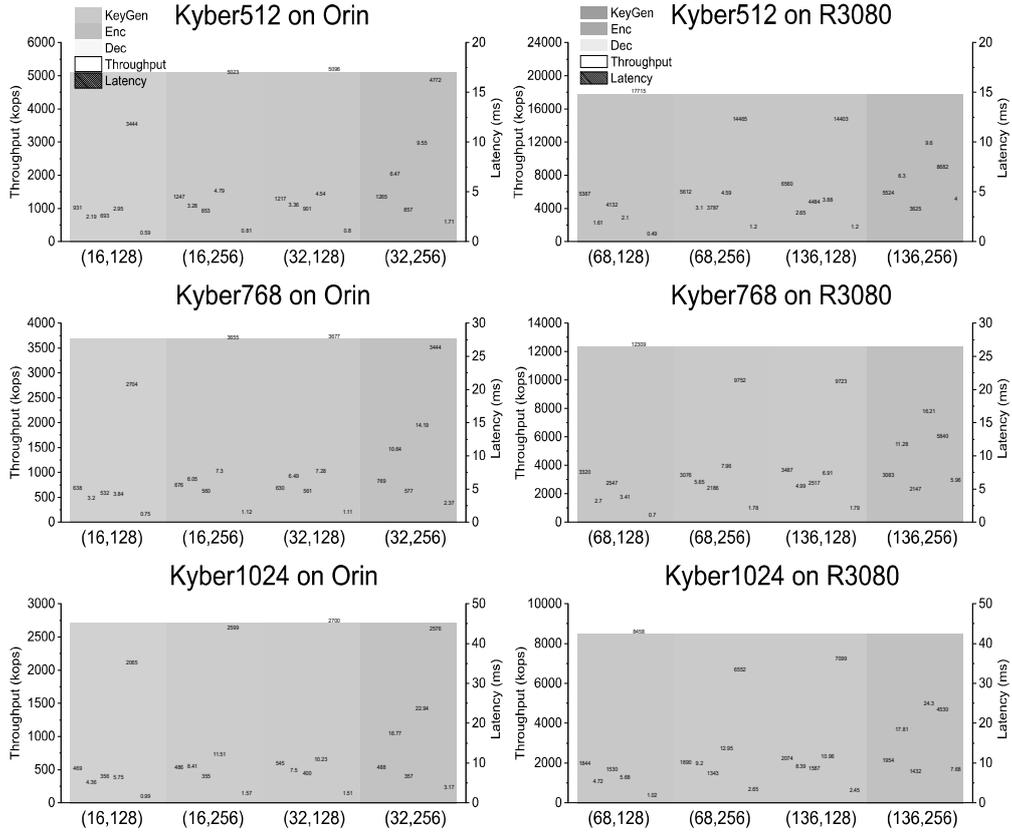
Figure 18: The performance of Kyber-512, Kyber-768, Kyber-1024 on Orin and R3080

operations. Effectively leveraging the data transmission capability of the memory bus yields notable results.

6.2.1 Performance Comparison with Other Implementations on the Same GPU

We first compare our results with other GPU-based works, as shown in Table 8.

Table 8: Comparison of Kyber-1024 throughput with GPU-based works

	Platform	KeyGen (kops)	Enc (kops)	Dec (kops)	KX° (kops)
Gupta *et al.* [GJCC20]	R3080	/	/	/	473
Wan *et al.* [WZF+22]		1,250	1,299	2,381	820
This work		2074	1587	8458	**1336**

° The throughput of KX is computed by $\frac{ab}{a+b}$, where a, b are the throughput of KeyGen and Dec.

As seen in Table 8, in terms of Kyber's implementation performance, our implementation's KX (Key Exchange) is 2.82x faster than Gupta *et al.*'s, which uses butterfly operations, and 1.62x faster than Wan *et al.*'s, which is based on Tensor Core as well. Specifically, compared to Wan *et al.*'s implementation, our Dec is 3.55x faster. Beyond adopting a more efficient NTT approach, as described in Section 5.2.1, we meticulously merged nearly every memory access, ensuring that each access transfers data close to 128

bits (the memory bus width of Orin). This substantially reduce memory access latency, leading to a significant performance improvement.

Notably, the Dec operation showcases a more significant performance advantage. The root of this lies in our utilization of Tensor Core-accelerated NTT/INTT implementations, and the computational load for KeyGen and Dec is primarily hash-based, which are challenging to accelerate on parallel computing platforms.

6.2.2   Performance Comparison with Other Implementations on CPU and FPGA

The previous implementations of Kyber are based on various platforms, targeting different scenarios and following different design ideas. Table 9 lists the throughput of Kyber-1024 for the related works.

Table 9: Comparison of throughput on Kyber-1024 with related works

	Platform	KeyGen (kops)	Enc/ Encaps* (kops)	Dec (kops)	Decaps (kops)
PQClean [MJP+]§	ARM Cortex-A75 @2.8GHz†	7.26	5.88	/	5.13
Xing *et al.* [XL21]§	Xilinx Artix-7 @161MHz	17.18	14.72	/	11.60
Aikata *et al.* [AMI+23]§	US+Z @270MHz	29.67	23.79	/	19.42
Sanal *et al.* [SKS+21]§	Apple A12 @2.49GHz	26.15	26.77	/	26.94
Becker *et al.* [BHK+22]§	ARM Cortex-A72 @1.5GHz	9.57	7.80	/	8.14
C-Ref [Sch]§	Intel Core i7-4770K @3492MHz	11.38	10.10	/	8.82
AVX2-Ref [Sch]§	Intel Core i7-4770K @3492MHz	47.6	36.0	/	44.2
This work	Orin(15W) @408MHz	94	69	775	63°
	Orin(30W) @612MHz	204	144	1641	132°
	Orin(50W) @816MHz	342	249	2272	224°
	Orin(60W) @1.3GHz	545	400	2700	348°
	R3080 @1.8GHz	2074	1587	8458	1336°

* Enc and Encaps have comparable computational costs.
° The throughput of decapsulation is computed by $\frac{ab}{a+b}$, where a, b are the throughput of Enc and Dec.
§ The implementation results of this work are obtained using one core of processor.
† The authors did not explicitly state the frequency of this platform in the paper. We consulted technical documentation and provided the common frequency of this platform for reference purposes only.

From Table 9, it can be observed that performance and power consumption under

different power modes of Orin exhibit an approximately proportional relationship. This implies that we can choose power settings according to specific requirements, making it easy to strike a balance between power consumption and performance.

Compared to resource-constrained devices, our implementation based on Orin exhibits performance approximately one order of magnitude higher, while the implementation based on R3080 demonstrates performance approximately two orders of magnitude higher. Even in the 15W power mode, our proposed method outperforms platforms like Apple A12 CPU, ARM Cortex-A72 CPU, and Xilinx Artix-7 FPGA, with performance gains 2.5-8.8x. Considering platform power consumption, this performance advantage remains substantial.

Then, we compare implementations based on different platforms in Table 9 for Kyber1024 in terms of throughput and latency.

When measured against the state-of-the-art ASIC implementation detailed in [AMI$^+$23], they achieved KeyGen, Encaps, and Decaps performance of 33.7, 42.04, and 51.5 μs, respectively, based on ASIC platform. Our implementation's throughput on Orin (60W) is approximately 17x higher than theirs. However, achieving such throughput is achieved when concurrently processing 4096 (32×128) requests . The execution latency for computing these 4096 requests is 240x greater than that of processing a single request as indicated in [AMI$^+$23].

In comparison to the leading FPGA-based implementation, Xing *et al.* [XL21] achieved decent performance by carefully scheduling sampling and NTT-related calculations within limited hardware resources. They achieved KeyGen, Encaps, and Decaps performances of 58.2, 67.9, and 86.2 μs, respectively, based on a single processor. In terms of throughput, our implementation on Orin (60W) is approximately 30x higher than theirs, but our latency of 4096 requests is around 136x larger.

In the context of efficient ARMv8-based software implementation discussed in [BHK$^+$22], they achieved KeyGen, Encaps, and Decaps performance of 104.5, 128.2, and 122.8 μs, respectively, based on the Cortex-A72. Our implementation on Orin (60W) achieves 42-56x higher throughput than theirs, while their latency is 73-97x faster, considering that our implementation simultaneously processes 4096 instances.

The above comparisons clearly illustrate the significant differences in performance aspects between implementations based on embedded platforms and parallel platforms.

Objectively speaking, their implementations are all single-core or single-threaded, focusing on low latency. Comparisons above are not conventional, and the purpose here is to provide readers with a rough understanding of the current state-of-the-art performance of different platforms. While their implementations may be more suitable for resource-constrained platforms, as a throughput-oriented approach, our implementation is most pronounced under high-scale concurrency, making our implementation well-suited for edge devices connecting a multitude of IoT devices.

For desktop platforms, we compare the results of desktop-grade CPUs with our performance on the R3080 platform. For the optimized AVX2 version of Kyber-1024 [Sch], our implementation on R3080 achieves speedup factors of approximately 43x, 44x, and 30x for KeyGen, Enc, and Decaps, respectively. Regarding latency, as we concurrently process 17408 (136×128) requests, we incur respective slowdowns of 95x, 93x, and 136x. It is worth noting that even in the lowest-power mode (15W) of Orin, the implementation exhibits performance superiority over the AVX version of Kyber-1024. Furthermore, given that the primary focus of this paper pertains to enhancements in NTT, the present implementation of Kyber, albeit effective, has not been subjected to fine-grained optimizations, thereby suggesting significant scope for additional improvements.

6.3 Limitation and Discussion

In this section, we will delve into the limitations and scalability of our proposed solution, identify its applicable scenarios, and explore generalizations to other PQC schemes. Finally,

we will discuss the impact of various modular multiplication methods on implementation performance.

6.3.1 Execution latency of the proposed scheme

Our primary focus is on throughput, making our solution well-suited for high-concurrency scenarios like servers. However, it does not offer a specific advantage in terms of latency when compared with CPU and FPGA-based implementations.

In future work, we aim to explore how to harness the internal parallelism of the algorithm on GPUs to achieve a relatively low-latency implementation of Kyber. Section 6.1.2 demonstrates the potential for ultra-low latency in the NTT implementation. Nevertheless, for the overall solution, pursuing extremely low latency in GPU implementations is considered imprudent. This is particularly true for components that require frequent synchronization or strict serial execution, such as SHA-3. Forcing fine-grained parallelization may result in a significant sacrifice in throughput.

In our future endeavors, we plan to design coarse-grained parallelism for relatively low-latency Enc and Dec processes while maintaining high throughput. It is important to note that KeyGen does not necessitate a latency-oriented implementation and can be offline computed into a pool. For instance, in the Enc process, leveraging our 3-Phase NTT implementation, which does not demand extensive concurrent requirements, we can dispatch multiple CUDA threads to independently compute a SHA-3 instance for sampling and matrix generation.

6.3.2 Generalization to other PQC candidates

For other PQC candidates, in fact, as long as NTT is used, our proposed solution should be applicable. For instance, the implementation of Dilithium [DKL+18], which is also based on MLWE, can be easily adapted using our techniques.

In Dilithium, the polynomial ring is denoted as $\mathbb{Z}_q[X]/(X^n + 1)$ with $n = 256$, just like in Kyber. However, in Dilithium, $q = 2^{23} - 2^{13} + 1$ means that the coefficients are represented as 23-bit integers. Therefore, the most challenging aspect of applying our proposed schemes to Dilithium lies in the larger modulus size (from 12-bit integers in Kyber to 23-bit integers in Dilithium), which necessitates more complex multi-precision or RNS decomposition. Overall, the difficulty in doing so is not significant; it merely requires some customized modifications to harness the performance of AI accelerators for efficient NTT in Dilithium.

6.3.3 Different modular multiplication (ModMult) methods

When performing modular reduction in our NTT implementation, we adopt the Montgomery ModMult [Mon85], as in Kyber's reference implementation [Sch]. Notably, other commonly used modular reduction algorithms include Barrett ModMult [Bar86], and Shoup ModMult [Sho01].

Firstly, we undertake a simple theoretical analysis of these three ModMult algorithms. On the one hand, Shoup ModMult algorithm has lower computational complexity than the conventional Barrett ModMult algorithm as analyzed in [KLC+19], Section II. On the other hand, Shoup ModMult necessitates the computation of a precomputed value related to one of the multiplicands (e.g., when computing $a \times b \mod q$, $\lfloor \frac{\beta b}{q} \rfloor$ should be precomputed, where β is the radix), and thus reduction must be performed immediately after each multiplication. However, it is essential to highlight that in our proposed scheme, we do not perform modular reduction immediately after each multiplication. Instead, we accumulate multiple product results (utilizing Tensor Cores) and perform the reduction only once at the end and the multiplicand used in each multiplication is different. This

approach is compatible with both Montgomery and Barrett modular reduction since they work for general multiplication products, but Shoup ModMult is not well-suited for our proposed accumulation pattern.

Secondly, when analyzing the computational cost of Montgomery ModMult and Barrett ModMult, excluding the precomputation part, we observe that both reduction operations involve two multiplications, one shift and one addition. Therefore, the computational overhead is quite similar, and the choice between the two has little impact on performance. In both cases, the accumulated product is first computed and then subjected to reduction.

7 Conclusion

This paper proposes an NTT calculation scheme dedicated to Kyber and further implements an entire implementation of Kyber. Based on our explored internal mechanism of Tensor Core and the unique design of the customized NTT in Kyber, our implementation outperforms existing Tensor Core-based solutions. We achieve significant speed-ups of 1.93x, 1.65x, 1.22x and 3.55x for polyvec_ntt, KeyGen, Enc and Dec in Kyber-1024, respectively. . Looking ahead, we plan to extend our achievements to accelerate NTT operations in Dilithium and Fully Homomorphic Encryption (FHE) cryptographic schemes.

Acknowledgements

References

[AEL+20] Erdem Alkim, Hülya Evkan, Norman Lahr, Ruben Niederhagen, and Richard Petri. ISA extensions for finite field arithmetic. *IACR TCHES*, 2020(3):219–242, 2020. `https://tches.iacr.org/index.php/TCHES/article/view/8589`.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *28th ACM STOC*, pages 99–108. ACM Press, May 1996.

[AMI+23] Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. Kali: A crystal for post-quantum security using Kyber and Dilithium. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 70(2):747–758, 2023.

[App] Apple. Apple unleashes M1. `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`. Accessed 19 May 2021.

[Bar86] Paul 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*, pages 311–323. Springer, 1986.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In *2018 IEEE*

European Symposium on Security and Privacy (EuroS&P), pages 353–367. IEEE, 2018.

[BHK⁺22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster dilithium, kyber, and saber on cortex-A72 and apple M1. *IACR TCHES*, 2022(1):221–244, 2022.

[Blo] Chromium Blog. Protecting chrome traffic with hybrid kyber kem. `https://blog.chromium.org/2023/08/protecting-chrome-traffic-with-hybrid.html`.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–737. Springer, 2012.

[BZW⁺23] Yi Bian, Fangyu Zheng, Yuewu Wang, Lingguang Lei, Yuan Ma, Jiankuo Dong, and Jiwu Jing. Asyncgbp: Unleashing the potential of heterogeneous computing for SSL/TLS with GPU-based provider. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023*, pages 337–346. ACM, 2023.

[Clo] Google Cloud. Cloud TPU. `https://cloud.google.com/tpu/`. Accessed 19 May 2021.

[DKL⁺18] Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–268, 2018.

[GJCC20] Naina Gupta, Arpan Jati, Amit Kumar Chauhan, and Anupam Chattopadhyay. PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):575–586, 2020.

[Kar63] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[KLC⁺19] Sunwoong Kim, Keewoo Lee, Wonhee Cho, Jung Hee Cheon, and Rob A. Rutenbar. FPGA-based accelerators of fully pipelined modular multipliers for homomorphic encryption. In David Andrews, René Cumplido, Claudia Feregrino, and Marco Platzner, editors, *2019 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2019, Cancun, Mexico, December 9-11, 2019*, pages 1–8. IEEE, 2019.

[LLZ⁺18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, and Kunpeng Wang. LAC: Practical ring-LWE based public-key encryption with byte-level modulus. Cryptology ePrint Archive, Report 2018/1009, 2018. `https://eprint.iacr.org/2018/1009`.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

[LSZH22] Wai-Kong Lee, Hwajeong Seo, Zhenfei Zhang, and Seong Oun Hwang. Tensorcrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU. *IEEE Access*, 10:20616–20632, 2022.

[MJP+] Kannwischer M, Rijneveld J, Schwabe P, Stebila D, and Wiggers. The PQClean project. `https://github.com/PQClean/PQClean`. Accessed 8 Apr 2022.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[MWB21] Liejun Ma, Xingjun Wu, and Guoqiang Bai. Parallel polynomial multiplication optimized scheme for CRYSTALS-KYBER post-quantum cryptosystem based on FPGA. In *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*, pages 361–365, 2021.

[NVIa] NVIDIA. CUDA C++ Programming Guide (Release 12.2). `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed 6 Oct 2023.

[NVIb] NVIDIA. NVIDIA tensor cores–unprecedented acceleration for HPC and AI. `https://www.nvidia.com/en-us/data-center/tensor-cores/`. Accessed 19 May 2021.

[NVIc] NVIDIA. The Future of Industrial-Grade Edge AI - NVIDIA Jetson AGX Orin Industrial module. `https://www.nvidia.com/en-gb/autonomous-machines/embedded-systems/jetson-orin/`.

[PZZ+17] Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, and Jiwu Jing. An efficient elliptic curve cryptography signature server with GPU acceleration. *IEEE Transactions on Information Forensics and Security*, 12(1):111–122, 2017.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

[SAB+22] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022. available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022`.

[Sch] Peter Schwabe. CRYSTALS-Cryptographic Suite for Algebraic Lattices. `https://pq-crystals.org/kyber/index.shtml`. Accessed 18 May 2021.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[Sho01] Victor Shoup. NTL: A library for doing number theory. 2001.

[SKS+21] Pakize Sanal, Emrah Karagoz, Hwajeong Seo, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Kyber on ARM64: Compact implementations of Kyber on 64-bit ARM Cortex-A processors. In *International Conference on Security and Privacy in Communication Systems*, pages 424–440. Springer, 2021.

[Too63] Andrei L Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. In *Soviet Mathematics Doklady*, volume 3, pages 714–716, 1963.

[WZF⁺22] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part III*, volume 13556 of *Lecture Notes in Computer Science*, pages 514–534. Springer, 2022.

[WZG⁺21] Rong Wei, Fangyu Zheng, Lili Gao, Jiankuo Dong, Guang Fan, Lipeng Wan, Jingqiang Lin, and Yuewu Wang. Heterogeneous-PAKE: Bridging the gap between PAKE protocols and their real-world deployment. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*, pages 76–90. ACM, 2021.

[WZL21] Lipeng Wan, Fangyu Zheng, and Jingqiang Lin. TESLAC: accelerating lattice-based cryptography with AI accelerator. In Joaquín García-Alfaro, Shujun Li, Radha Poovendran, Hervé Debar, and Moti Yung, editors, *Security and Privacy in Communication Networks - 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6-9, 2021, Proceedings, Part I*, volume 398 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 249–269. Springer, 2021.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-KYBER on FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 328–356, 2021.

Appendix

A INTT

A.1 Start from formula

$$\text{INTT}(\hat{f}) = f$$

$$f_{2u} = n^{-1} \cdot \sum_{v=0}^{127} \hat{f}_{2v}\zeta^{-(2\mathbf{br}_7(v)+1)u} \tag{18}$$

$$f_{2u+1} = n^{-1} \cdot \sum_{v=0}^{127} \hat{f}_{2v+1}\zeta^{-(2\mathbf{br}_7(v)+1)u} \tag{19}$$

To facilitate subsequent descriptions, we introduce the following notations:

$$u = 16u_0 + u_1, v = 16v_0 + v_1, u_0, v_0 \in \mathbb{Z}_8, u_1, v_1 \in \mathbb{Z}_{16}$$

$$f_0(u_0, u_1) = f_{2(16u_0+u_1)} = f_{2u}, f_1(u_0, u_1) = f_{2(16u_0+u_1)+1} = f_{2u+1}$$

$$\hat{f}_0(u_0, u_1) = \hat{f}_{2(16u_0+u_1)} = \hat{f}_{2u}, \hat{f}_1(u_0, u_1) = \hat{f}_{2(16u_0+u_1)+1} = \hat{f}_{2u+1}$$

Using these notations, Equation (18) can be deduced as follows:

$$
\begin{aligned}
\hat{f}_0(u_0, u_1) &= n^{-1} \cdot \sum_{v=0}^{127} f_0(v_0, v_1) \zeta^{-u(2\mathbf{br}_7(v)+1)} \\
&= n^{-1} \cdot \sum_{v_0=0}^{7} \sum_{v_1=0}^{15} f_0(v_0, v_1) \zeta^{-(16u_0+u_1)(2\mathbf{br}_3(v_0)+16\mathbf{br}_4(v_1)+1)} \\
&= n^{-1} \cdot \sum_{v_0=0}^{7} \sum_{v_1=0}^{15} f_0(v_0, v_1) \zeta^{-16u_0(2\mathbf{br}_3(v_0)+1)} \cdot \zeta^{-u_1(2\mathbf{br}_3(v_0)+1)} \cdot \zeta^{-32u_1\mathbf{br}_4(v_1)}
\end{aligned}
\tag{20}
$$

We introduce notations for the sub-twiddle factors like NTT:

$$
\begin{aligned}
\zeta_0(u_0, v_0) &= \zeta^{-16u_0(2\mathbf{br}_3(v_0)+1)} \\
\zeta_1(u_1, v_0) &= \zeta^{-u_1(2\mathbf{br}_3(v_0)+1)} \\
\zeta_2(u_1, v_1) &= \zeta^{-u_1(16\mathbf{br}_4(v_1))}
\end{aligned}
$$

With these notations, $\hat{f}_0(u_0, u_1)$ can be represented as:

$$
\hat{f}_0(u_0, u_1) = \sum_{v_0=0}^{7} \sum_{v_1=0}^{15} f_0(v_0, v_1) \cdot \zeta_0(u_0, v_0) \cdot \zeta_1(u_1, v_0) \cdot \zeta_2(u_1, v_1)
\tag{21}
$$

Similarly

$$
\hat{f}_1(u_0, u_1) = \sum_{v_0=0}^{7} \sum_{v_1=0}^{15} f_1(v_0, v_1) \cdot \zeta_0(u_0, v_0) \cdot \zeta_1(u_1, v_0) \cdot \zeta_2(u_1, v_1)
\tag{22}
$$

A.2   Iteration-based INTT Algorithms

A.2.1   2-Phase INTT

Merging ζ_1 and ζ_2, therefore
let $\zeta_{12}(u_1, v_0, v_1) = \zeta_1(u_1, v_0) \cdot \zeta_2(u_1, v_1)$

$$
\hat{f}_0(u_0, u_1) = \underbrace{\sum_{v_0=0}^{7} \underbrace{\left[\sum_{v_1=0}^{15} f_0(j_0, j_1) \cdot \zeta_{12}(u_1, v_0, v_1)\right]}_{\text{Phase 1}} \cdot \left[\zeta_0(u_0, v_0)\right]}_{\text{Phase 2}}
$$

A.2.2   3-Phase INTT

Computing ζ_2, ζ_1, and ζ_0 separately.

$$
\hat{f}_0(u_0, u_1) = \underbrace{\left\{ \underbrace{\sum_{v_0=0}^{7} \underbrace{\left[\sum_{v_1=0}^{15} f_0(v_0, v_1) \cdot \zeta_2(u_1, v_1)\right]}_{\text{Phase 1}} \cdot \zeta_1(u_1, v_0) }_{\text{Phase 2}} \right\} \cdot \zeta_0(u_0, v_0)}_{\text{Phase 3}}
$$

B   Precomputed Table of Twiddle Factors

Since the powers of ζ can be known in advance, then all the twiddle factors can be precomputed and stored in the memory before the procedure. When NTT is executed, these values can be obtained by directly looking up the table instead of multiplying, like the original implementation.

Furthermore, considering the Montgomery reduction, the precomputed table should be multiplied by $R = 2^{12} \pmod q$ in advance to transform it into the Montgomery form, reducing the computational load during the reduction process.

Please note, the generation method provided here pertains only to the NTT precomputated table; the part for INTT can be similarly derived.

B.1   Precomputed Table of 2-Phase NTT

Phase 1

In the first phase of 2-phase NTT, the 8 sections of the coefficient matrix will be multiplied by the 8 precomputed matrices.

For $k = 0, 1, \ldots, 7$,

$$
\begin{aligned}
&P_{1k} \quad (2-\text{phase}) \\
&= \{\zeta_{01}(k)\}_{16 \times 16} \cdot R \\
&= \begin{pmatrix}
\zeta^{(8 \times 0 + k)(2\text{br}_4(0)+1)} & \zeta^{(8 \times 0 + k)(2\text{br}_4(1)+1)} & \cdots & \zeta^{(8 \times 0 + k)(2\text{br}_4(15)+1)} \\
\zeta^{(8 \times 1 + k)(2\text{br}_4(0)+1)} & \zeta^{(8 \times 0 + k)(2\text{br}_4(1)+1)} & \cdots & \zeta^{(8 \times 0 + k)(2\text{br}_4(15)+1)} \\
\vdots & \vdots & \ddots & \vdots \\
\zeta^{(8 \times 15 + k)(2\text{br}_4(0)+1)} & \zeta^{(8 \times 15 + k)(2\text{br}_4(1)+1)} & \cdots & \zeta^{(8 \times 15 + k)(2\text{br}_4(15)+1)}
\end{pmatrix}_{16 \times 16} \cdot R
\end{aligned}
$$

Phase 2

In the second phase of 2-phase NTT, we perform matrix multiplication.

$$
\begin{aligned}
&P_2 \quad (2-\text{phase}) \\
&= \{\zeta_2\}_{16 \times 16} \cdot R \\
&= \begin{pmatrix}
\zeta^{32 \times 0 \times \text{br}_3(0)} & 0 & \cdots & \zeta^{32 \times 0 \times \text{br}_3(7)} & 0 \\
0 & \zeta^{32 \times 0 \times \text{br}_3(0)} & \cdots & 0 & \zeta^{32 \times 0 \times \text{br}_3(7)} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\zeta^{32 \times 7 \times \text{br}_3(0)} & 0 & \cdots & \zeta^{32 \times 7 \times \text{br}_3(7)} & 0 \\
0 & \zeta^{32 \times 7 \times \text{br}_3(0)} & \cdots & 0 & \zeta^{32 \times 7 \times \text{br}_3(7)}
\end{pmatrix}_{16 \times 16} \cdot R
\end{aligned}
$$

B.2   Precomputed Table of 3-Phase NTT

Phase 1 In the first phase of 3-phase NTT, we perform matrix multiplication.

$$
\begin{aligned}
&P_1 \quad (3-\text{phase}) \\
&= \{\zeta_0\}_{16 \times 16} \cdot R \\
&= \begin{pmatrix}
\zeta^{(8 \times 0)(2\text{br}_4(0)+1)} & \zeta^{(8 \times 0)(2\text{br}_4(1)+1)} & \cdots & \zeta^{(8 \times 0)(2\text{br}_4(15)+1)} \\
\zeta^{(8 \times 1)(2\text{br}_4(0)+1)} & \zeta^{(8 \times 0)(2\text{br}_4(1)+1)} & \cdots & \zeta^{(8 \times 0)(2\text{br}_4(15)+1)} \\
\vdots & \vdots & \ddots & \vdots \\
\zeta^{(8 \times 15)(2\text{br}_4(0)+1)} & \zeta^{(8 \times 15)(2\text{br}_4(1)+1)} & \cdots & \zeta^{(8 \times 15)(2\text{br}_4(15)+1)}
\end{pmatrix}_{16 \times 16} \cdot R
\end{aligned}
$$

Phase 2

The second phase of the 3-phase NTT involves computing the Hadamard Product of matrices. Since the product matrix obtained from the previous step has a layout in the registers that is equivalent to applying the inverse of the `wmma.store` transformation (refer to Figure 13), the precomputed matrix also needs to apply the inverse of the `wmma.store` transformation to achieve a one-to-one correspondence of multiplication elements.

$$P_2 \quad (3-\text{phase})$$
$$=R \cdot \texttt{wmma.store}^{-1}(\{\zeta_1\}_{16 \times 16})$$
$$=R \cdot \texttt{wmma.store}^{-1}$$
$$\begin{pmatrix} \zeta^{0 \times (2\text{br}_4(0)+1)} & \zeta^{0 \times (2\text{br}_4(0)+1)} & \cdots & \zeta^{7 \times (2\text{br}_4(0)+1)} & \zeta^{7 \times (2\text{br}_4(0)+1)} \\ \zeta^{0 \times (2\text{br}_4(1)+1)} & \zeta^{0 \times (2\text{br}_4(1)+1)} & \cdots & \zeta^{7 \times (2\text{br}_4(1)+1)} & \zeta^{7 \times (2\text{br}_4(1)+1)} \\ \vdots & \vdots & \ddots & \vdots & \\ \zeta^{0 \times (2\text{br}_4(15)+1)} & \zeta^{0 \times (2\text{br}_4(15)+1)} & \cdots & \zeta^{7 \times (2\text{br}_4(15)+1)} & \zeta^{7 \times (2\text{br}_4(15)+1)} \end{pmatrix}_{16 \times 16}$$
$$=R \cdot \begin{pmatrix} \zeta^{0 \times (0\text{br}_4(0)+1)} & \zeta^{0 \times (2\text{br}_4(0)+1)} & \cdots & \zeta^{5 \times (2\text{br}_4(8)+1)} & \zeta^{5 \times (2\text{br}_4(8)+1)} \\ \zeta^{2 \times (2\text{br}_4(0)+1)} & \zeta^{2 \times (2\text{br}_4(0)+1)} & \cdots & \zeta^{7 \times (2\text{br}_4(8)+1)} & \zeta^{7 \times (2\text{br}_4(8)+1)} \\ \vdots & \vdots & \ddots & \vdots & \\ \zeta^{2 \times (2\text{br}_4(7)+1)} & \zeta^{2 \times (2\text{br}_4(7)+1)} & \cdots & \zeta^{7 \times (2\text{br}_4(15)+1)} & \zeta^{7 \times (2\text{br}_4(15)+1)} \end{pmatrix}_{16 \times 16}$$

Phase 3

In the third phase of 3-phase NTT, we perform matrix multiplication. Based on the analysis from the previous section, the coefficient matrix at this moment is equivalent to having been right-multiplied by the permutation matrix P (refer to Figure 15). Therefore, the precomputed matrix in this phase needs to be pre-multiplied by P^{-1} on the left to obtain the correct result.

$$P_3 \quad (3-\text{phase})$$
$$=R \cdot P^{-1} \times \{\zeta_2\}_{16 \times 16}$$
$$=R \cdot P^{-1} \times$$
$$\begin{pmatrix} \zeta^{32 \times 0 \times \text{br}_3(0)} & 0 & \cdots & \zeta^{32 \times 0 \times \text{br}_3(7)} & 0 \\ 0 & \zeta^{32 \times 0 \times \text{br}_3(0)} & \cdots & 0 & \zeta^{32 \times 0 \times \text{br}_3(7)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \zeta^{32 \times 7 \times \text{br}_3(0)} & 0 & \cdots & \zeta^{32 \times 7 \times \text{br}_3(7)} & 0 \\ 0 & \zeta^{32 \times 7 \times \text{br}_3(0)} & \cdots & 0 & \zeta^{32 \times 7 \times \text{br}_3(7)} \end{pmatrix}_{16 \times 16}$$