RESEARCH-ARTICLE

ML-Cube: Accelerating Module-Lattice-Based Cryptography using Machine Learning Accelerators with a Memory-Less Design

TIAN ZHOU, University of Science and Technology of China, Hefei, Anhui, China

FANGYU ZHENG, University of Chinese Academy of Sciences, Beijing, China

ZHUOYU XIE, University of Chinese Academy of Sciences, Beijing, China

WENXU TANG, University of Science and Technology of China, Hefei, Anhui, China

GUANG FAN, University of Chinese Academy of Sciences, Beijing, China

YIJING NING, University of Science and Technology of China, Hefei, Anhui, China

View all

PDF Download
3719027.3765130.pdf
23 December 2025
Total Citations: 0
Total Downloads: 806

Published: 19 November 2025

Citation in BibTeX format

CCS '25: ACM SIGSAC Conference on Computer and Communications Security
October 13 - 17, 2025
Taipei, Taiwan

Conference Sponsors:
SIGSAC

ML-Cube: Accelerating Module-Lattice-Based Cryptography using Machine Learning Accelerators with a Memory-Less Design

Tian Zhou
University of Science and Technology of China
School of Cyber Science and Technology
Hefei, China
weekdayzt@mail.ustc.edu.cn

Fangyu Zheng*
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
zhengfangyu@ucas.ac.cn

Zhuoyu Xie
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
xiezhuoyu25@mails.ucas.ac.cn

Wenxu Tang
University of Science and Technology of China
School of Cyber Science and Technology
Hefei, China
wenxutang@mail.ustc.edu.cn

Guang Fan
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
fanguang328@gmail.com

Yijing Ning
University of Science and Technology of China
School of Cyber Science and Technology
Hefei, China
truegeorge@mail.ustc.edu.cn

Yi Bian
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
bianyi@ucas.ac.cn

Jingqiang Lin
University of Science and Technology of China
School of Cyber Science and Technology
Hefei, China
linjq@ustc.edu.cn

Jiwu Jing
University of Chinese Academy of Sciences
School of Cryptology
Beijing, China
jwjing@ucas.ac.cn

Abstract

The rapid advancement of AI technologies has led to a dramatic surge in computational demands, driving significant breakthroughs in ML accelerators. The powerful performance of these accelerators has attracted the attention of cryptography researchers, and recent studies have begun to explore their use in accelerating cryptographic operations. However, treating these accelerators as black boxes leads to high latency, and strict concurrency requirements, which hinder their practical deployment.

In this paper, we go beyond the black-box treatment of ML accelerators and introduce ML-Cube (ML^3), a novel memory-less framework that leverages ML accelerators to implement module-lattice-based PQC, FIPS 203 ML-KEM, and FIPS 204 ML-DSA. The performance benefits of ML-Cube arise from our thorough analysis of ML accelerator internals. Rather than treating the accelerators as black boxes, we dissect their operating mechanisms and design tailored mathematical transformations for cryptographic acceleration. This enables memory-less (I)NTT and polynomial multiplication that minimizes external memory dependencies and reduces latency. We further address the high latency and excessive parallelism demands of traditional SIMT-based implementations by fully parallelizing both ML-KEM and ML-DSA schemes. Our experiments show that our Tensor Core-based (I)NTT achieves a 2.03×–3.56× speedup over a highly-optimized CUDA-core implementation. Moreover, our memory-less polynomial multiplication attains a 10× speedup, and the full ML-KEM reaches up to a 3.58× speedup with only less than one-tenth of the latency compared with SOTA approach (CHES '24). Additionally, our enhanced ML-DSA implementation offers a 30% to 55% throughput improvement over the previous SOTA methods (TDSC '24) under the server-oriented model. Importantly, by confining core computations within registers, our approach inherently mitigates memory disclosure and cache-based side-channel attacks, thereby enhancing overall security.

* Fangyu Zheng is the corresponding author (*E-mail: zhengfangyu@ucas.ac.cn*).

CCS '25, Taipei
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765130

CCS Concepts

• **Security and privacy → Public key (asymmetric) techniques**; • **Computing methodologies → Parallel computing methodologies**.

Keywords

GPU, Tensor Core, Lattice-based Cryptography, Number Theory Transformation

ACM Reference Format:
Tian Zhou, Fangyu Zheng*, Zhuoyu Xie, Wenxu Tang, Guang Fan, Yijing Ning, Yi Bian, Jingqiang Lin, and Jiwu Jing. 2025. ML-Cube: Accelerating Module-Lattice-Based Cryptography using Machine Learning Accelerators with a Memory-Less Design. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765130

1 Introduction

To meet the growing computational demands of artificial intelligence (AI) in the era of large-scale models, major processor vendors have introduced dedicated machine learning (ML) accelerators, such as Google TPU [7], Intel VNNI [8], Apple Neural Engine [16], NVIDIA Tensor Core [9], and AMD MI300X [3]. Compared to general-purpose processors, these accelerators focus on low-precision arithmetic and offer significantly higher computational throughput. For example, NVIDIA's Volta architecture introduced Tensor Cores optimized for tensor operations, which have since evolved, with the Tesla H100 achieving up to 1979 INT8 Tensor TOPS.

At the same time, to counter the threat of quantum computing, NIST has released several post-quantum cryptographic (PQC) standards, including FIPS 203 [25], FIPS 204 [24], FIPS 205 [26], and the upcoming FIPS 206. Among the selected schemes, module-lattice-based algorithms are particularly attractive due to their balanced trade-offs between security, performance, and key size.

The intersection of AI and cryptography has become a compelling research direction, covering areas such as privacy-preserving machine learning [19, 22] and ML-assisted cryptanalysis [6, 30]. This raises a natural question for cryptographic engineers and users alike:

> *Can ML accelerators be harnessed to accelerate cryptographic computations, enabling cryptography to benefit from AI's hardware advances?*

1.1 Motivations

Cryptography and computing have historically evolved in parallel, with each influencing the development of the other. In fact, if cryptographic implementations fail to integrate with emerging platforms such as ML accelerators, the performance gap between plaintext and ciphertext processing will become even more challenging to bridge.

The computational power of ML accelerators presents new opportunities for optimizing PQC implementations. Consequently, an increasing number of researchers have begun leveraging ML accelerators for cryptographic applications, including fully homomorphic encryption (FHE) and PQC. Recent works exemplifying this trend include [10, 37, 38, 41], which explore novel ways to exploit AI hardware for secure and efficient cryptographic computations. However, these studies exhibit several critical limitations:

- Previous works often *treat ML accelerators as black boxes*, lacking control over their internal mechanisms. For example, Fan *et al.*[10] used NVIDIA's cuBLAS for Tensor Core-based matrix multiplication instead of custom implementations, limiting fine-grained control and NTT integration. While Zhou *et al.*[41] optimized memory layouts and used PTX for register-level computations, NTT inputs and outputs still relied on memory storage.
- Moreover, most of the previous implementations [15, 17, 37, 41] *prioritize throughput but suffer from high latency* due to massive concurrency, making them impractical. For example, Zhou *et al.*[41] parallelized core operations, yet the overall system remains largely serial. Tasks like rejection sampling in ML-DSA require high concurrency, further increasing latency. Additionally, while these works optimize NTT, they often neglect related computations like multiplication, leading to inefficiencies in memory access that offset performance gains.

1.2 Contributions

Building on these observations, we introduce ML-Cube (ML^3), a novel **m**emory-**l**ess framework that leverages **ML** accelerators to implement standard **m**odule-**l**attice-based PQC schemes, namely FIPS 203 ML-KEM and FIPS 204 ML-DSA. Our contributions are three-fold and can be summarized as follows:

- **An in-depth analysis of ML accelerators** is conducted, investigating the internal mechanisms of the Tensor Core and examining how various usage patterns of thread-internal fragment registers influence computations. Based on these insights, we introduce *synthetic fragments* to elegantly manipulate fragments for efficient general chained matrix multiplication entirely within the registers, avoiding additional transformation overhead. This contribution is not only applicable in PQC but can also be utilized for conventional matrix multiplication. **(Section 3)**
- **A memory-less framework for polynomial multiplication** is implemented for optimizing polynomial computations in ML-KEM and ML-DSA. By utilizing the native interconnection mechanism of the Tensor Core, we developed a warp-level parallel polynomial computation model that eliminates the need for inter-thread communication, thereby maximizing the potential of the Tensor Core's ALU. Additionally, we proposed a modular arithmetic scheme suitable for low-precision arithmetic instructions of Tensor Core. **(Section 4)**
- **Fully-parallelized MLWE-based PQC** is proposed to address the high latency and high-concurrency requirement in existing implementations. This approach distributes computational workloads across threads, effectively leveraging the SIMT execution model to reduce per-thread computation and hardware resource consumption. Building on the

memory-less polynomial multiplication framework, we complete multiple computational stages entirely within registers and fuse core-value-dependent multiplications, minimizing memory usage and access overhead. Especially, a highly compact implementation of *rejection sampling loop* in ML-DSA, addressing loop imbalance and minimizing unnecessary resource consumption. **(Section 5)**

The above efforts have achieved tremendous performance improvement according to a comprehensive evaluation of the ML-Cube framework in **Section 6**. Our memory-less (I)NTT (designed for ML-DSA and ML-KEM) achieves a performance improvement of 2.03 to 3.56× compared to conventional CUDA-core-based implementations. When compared to state-of-the-art Tensor Core-based polynomial multiplication, our approach, which confines computations within registers as much as possible, further achieves up to a 10.25× speedup. For the complete ML-KEM, our fully parallelized design enables ML-Cube to achieve 1.67×, 1.72×, 3.58×, and 1.85× speedups for KeyGen, Enc/Encaps, Dec, and Decaps at Level 5. Notably, the overall latency is reduced to less than 1/10 of the prior SOTA, while the required number of parallel requests to reach peak performance is also reduced to 1/10. Additionally, our enhanced ML-DSA implementation delivers a 30% to 55% increase in throughput over previous state-of-the-art methods under the server-oriented model.

2 Preliminary

This section briefly introduces module-lattice-based cryptography, as well as ML-KEM and ML-DSA.

2.1 Notation

For a prime q, $\mathbb{Z}_q = \{0, 1, \dots, q-1\}$ 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 (including elements in $\mathbb{Z}_q$), while bold lowercase letters represent vectors with coefficients in R_q. By default, all vectors are column vectors. Bold uppercase letters denote matrices. Matrix multiplication is expressed as $\mathrm{C} = \mathrm{A} \cdot \mathrm{B}$, and the Hadamard product of two matrices is written as $\mathrm{C} = \mathrm{A} \circ \mathrm{B}$. The rank k of a polynomial vector indicates that the polynomial vector contains k polynomials.

For a finite field $\mathbb{Z}_q$, a primitive n-th root of unity ω exists if n divides $q - 1$, where $\omega^n \equiv 1 \pmod q$. The notation $\hat{f}$ represents the NTT domain representation of the polynomial $f \in R_q$.

2.2 Module-Lattice-Based Cryptography

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) [2], Learning With Errors (LWE) [31] and RLWE [21] problems. Combining the security advantages of LWE and the flexibility of Ring-LWE, Langlois *et al.* [18] demonstrated the worst-case to average-case reductions for module lattices. Intuitively, the size of matrix $\mathbf{A}$ in Module-LWE is $k \times k$, where k is the rank. The elements in the matrix are vectors selected from $\mathbb{Z}_q^n$.

2.2.1 *ML-KEM.*

At a high level, the construction of the scheme ML-KEM proceeds in two steps. First, the ideas discussed previously are used to construct a public-key encryption (PKE) scheme from the MLWE problem. First, the public-key encryption (PKE) of ML-KEM scheme is constructed from the MLWE problem. Second, this PKE scheme is converted into a key-encapsulation mechanism using the so-called Fujisaki-Okamoto (FO) transform [11]. As a result, ML-KEM is believed to satisfy IND-CCA2 security.

The PKE scheme in ML-KEM is introduced as follows.

- KeyGen: In ML-KEM, the private key is $\mathrm{s} \in R_q^k$, and the corresponding public key is a collection of "noisy" linear equations $(\mathrm{A}, \mathrm{As} + \mathrm{e})$ based on s. The results are appropriately encoded into byte arrays and output.
- IND-CPA Enc: The encryptor generates a vector $\mathrm{y} \in R_q^k$ and noise terms $\mathrm{e}_1 \in R_q^k$ and $e_2 \in R_q$ then computes the "new noisy equation", which is $(\mathrm{A}^\top \mathrm{y} + \mathrm{e}_1, \mathrm{t}^\top \mathrm{y} + e_2)$. The input message m is encoded appropriately to μ, and added to the latter term in the pair. An appropriate encoding μ of the input message m is then added to the latter term in the pair. Finally, the resulting pair (u, v) is compressed, serialized into a byte array, and output as the ciphertext.
- IND-CPA Dec: Here, one can think of u' as the coefficients of the equation and v' as the constant term. The decryptor uses the private key to compute the true constant term $v = \mathrm{s}^\top \mathrm{u}'$ and calculate $v' - v$. The decryption algorithm ends by decoding the plaintext message m from $v' - v$ and outputting m.

The key feature of the KEM scheme, derived from the PKE scheme, is the application of the FO transformation in its decapsulation algorithm, which involves decrypting the ciphertext to obtain M' and re-encrypting M' to generate c' for comparison with c. If they match, the decrypted shared key is accepted; otherwise, the decryption is implicitly rejected with random bytes from the private key.

2.2.2 *ML-DSA.*

The high-level concept of ML-DSA can be summarized as follows:

- KeyGen: in ML-DSA, a polynomial matrix $\mathrm{A} \in R_q^{k \times l}$ and polynomial vectors $\mathrm{s}_1 \in R_q^l$ and $\mathrm{s}_2 \in R_q^k$ are first generated, and the core operation computes the public value $\mathrm{t} = \mathrm{As}_1 + \mathrm{s}_2$. The polynomial vectors $(\mathrm{t}_1, \mathrm{t}_0)$ are produced such that $\mathrm{t} = \mathrm{t}_1 \cdot 2^d + \mathrm{t}_0$. The public key is a binary tuple $(\mathrm{A}, \mathrm{t}_1)$, while the corresponding private key is a 6-tuple $(\mathrm{A}, K, tr, \mathrm{s}_1, \mathrm{s}_2, \mathrm{t}_0)$, where K is a private random seed and tr is a hash of the public key.
- Signing: the signer first computes $\mu \leftarrow \mathrm{H}(tr \parallel M)$ and $\rho'' \leftarrow \mathrm{H}(K \parallel rnd \parallel \mu)$, where M is the message and rnd is a random number. The signing algorithm's core involves a *rejection sampling loop*, which repeats until a valid signature is generated. In detail, the signer generates a polynomial vector $\mathrm{y} \in R_q^l$ using ρ'' and a counter κ. Next, the signer computes $\mathrm{w} = \mathrm{Ay}$, from which a "commitment" w_1 is derived. Then, a "challenge" c is generated from the commitment hash $\tilde{c}$, which is obtained by hashing w_1 and the message representative μ. Subsequently, the products of c with s_1, s_2, and t_0 are computed, followed by a norm check,

and the response $z = y + cs_1$ is generated. If all checks pass, the signer can compute a hint polynomial h, and the final signature is a triple $(\tilde{c}, z, h)$.

- Verify: the verifier derives c from $\tilde{c}$ and computes $w'_{\text{Approx}} = Az - ct_1 \cdot 2^d$. Then verifier uses the hint h to obtain w'_1 from w'_{Approx}. Finally, the verifier checks that the response z and hint h are valid and that the reconstructed w'_1 matches the commitment hash w_1 in the signature. If all these checks succeed, the signature is valid; otherwise, it is invalid.

3 Understanding Tensor Cores

This section provides an in-depth study of the internal mechanisms of NVIDIA Tensor Cores, which is the foundation of our work.

3.1 Tensor Cores in a Nutshell

ML accelerators (including Tensor Core) are designed for low precision arithmetic, typically using formats like INT8, FP16, or BF16. They are also limited to specific computation modes, mainly supporting matrix multiplication or inner product operations, which are core to machine learning.

Listing 1 (a slightly modified version of the one in *CUDA Programming Guide* [28]) to perform a 16×16 matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ using Tensor Core, where the element types of the input matrices $\mathbf{A}$ (`int_8 *a`) and $\mathbf{B}$ (`int_8 *b`) are INT8, and the element type of the accumulator matrix $\mathbf{C}$ (`int *c`) is INT32. In fact, Tensor Core supports matrix multiplication and accumulation $\mathbf{C} = \mathbf{A} \cdot \mathbf{B} + \mathbf{C}$, but we use matrix multiplication for demonstration.

The main `wmma` instructions are:

- `wmma.load` loads $\mathbf{A}$ or $\mathbf{B}$ from memory into 8 INT8 registers (called *fragment*) for each of the 32 threads in the warp. This instruction requires specifying the multiplicands (as the left matrix, `matrix_a`, or right matrix, `matrix_b`) and the read order (row-major or column-major).
- `wmma.mma` performs the matrix multiplication on the Tensor Core, $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$.
- `wmma.store` stores the result $\mathbf{C}$ from 8 INT32 registers for each of the 32 threads in the warp back to memory. This instruction requires specifying the store order, either row-major or column-major.

```
__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);
}
```

Listing 1: Standard invocation methods of WMMA API

Row and column-major ordering merely define how matrix elements are indexed; however, this distinction is irrelevant within the internal mechanisms. Throughout this paper, we adopt the row-major convention for all descriptions.

3.2 Beyond the Programming Guide

In fact, NVIDIA does not have a specific manual on how its Tensor Core works internally, so many previous works [10, 37] treated the ML accelerator as a black box for relevant computations. ConvKyber [41] found that the internal mechanisms of ML accelerators can facilitate cryptographic algorithm implementations. This section will significantly extend the findings of [41].

Tensor Core supports matrix multiplication with various data types and matrix sizes. However, as noted in [37], INT8 input elements and 16×16 matrix multiplication are the most practical. Therefore, we take this configuration as an example in the following discussion.

Through our experiments, we have discovered that Tensor Cores process matrix elements using the following approach, as shown in Figure 1.

Remark 1. First, for 16 × 16 matrices A, B, each thread in a warp (consisting of 32 threads) processes $\frac{16 \times 16}{32} = 8$ elements per thread. These 8 elements come from two rows or two columns, depending on whether the matrix is served as `matrix_a` or `matrix_b`. Then, the 32 threads in a warp are divided into 8 groups, each containing 4 threads. In Group i ($0 \leq i \leq 7$), the four threads store elements from the i-th and $(i + 8)$-th rows/columns. When executing the `wmma.mma` instruction, Group i multiplies its stored fragment of A with all 8 fragments of B (including its own) via a series of 4-element inner products. The four inner products are then accumulated to produce a 16-element inner product result for fragment C, corresponding to rows/columns i and $(i+8)$ of the output matrix.

3.2.1 *Bypassing the APIs via synthetic fragments.*

In the *CUDA Programming Guide* [28], the usage of Tensor Cores is subject to strict requirements, where computations are expected to follow a predefined pattern only when data is properly encapsulated into fragments. For example, the *PTX ISA Guide* [29], in the section on `wmma.mma`, explicitly states: "The qualifiers .alayout and .blayout must match the layout specified on the `wmma.load` instructions that produce the contents of operands, respectively."

In fact, we have found that developers can bypass `wmma.load` entirely by directly manipulating the register values of input and output operands and passing them to `wmma.mma` using a developer-defined format. We refer to it as a *synthetic fragment*, because it is manually assembled by developers rather than generated through the standard API `wmma.load`. Therefore, it is crucial to analyze the underlying mechanisms of `wmma.load` and `wmma.store`.

For *synthetic fragments*, since they are not generated by the native `wmma.load`, they do not inherently retain information about whether the data was originally stored in row-major or column-major order. Through further experiments, we found that the CUDA compiler, when handling such operands that are not loaded via standard `wmma.load` instructions, defaults to interpreting them in row-major order (i.e., indexed in Figure 1 by row). The mapping of register elements to the original matrix depends on the operand's

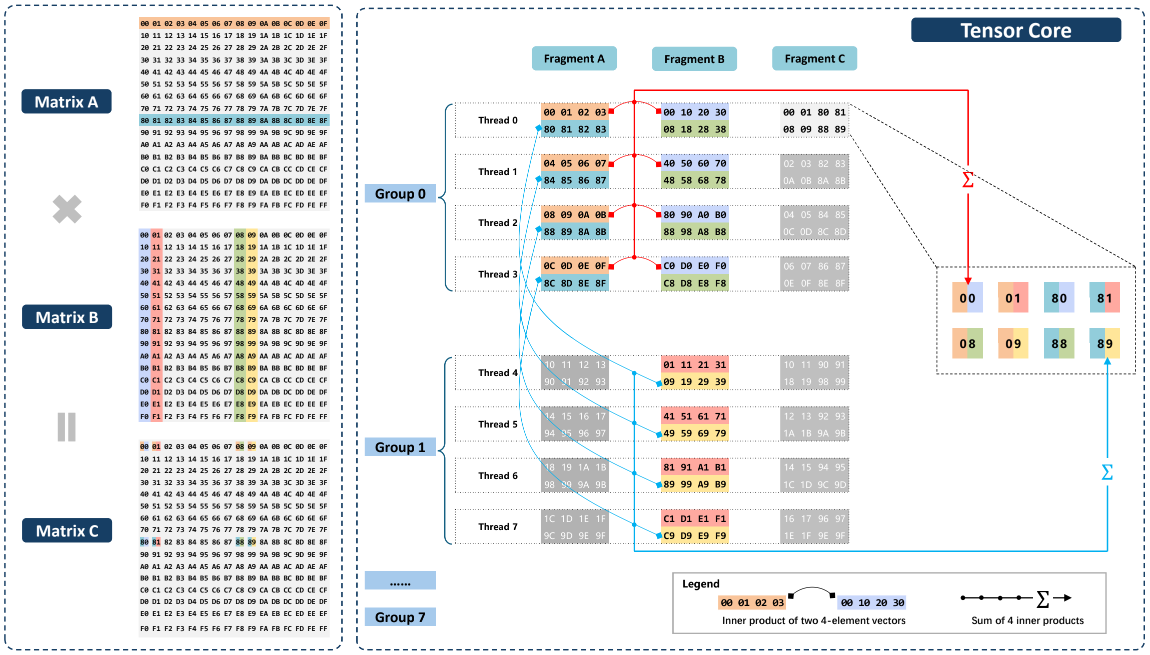

Figure 1: How Tensor Core computes matrix multiplications (This figure demonstrates the multiplication process between the 0th and 8th rows of matrix A and the 0th, 1st, 8th, and 9th columns of matrix B. During this process, the interaction of data between Thread 0–3 and other threads is primarily involved. The computation of eight matrix products within Thread 0 are specifically focused on as an example.)

position in the `wmma` instruction. Simply put, the synthetic fragment for the left-hand matrix (`matrix_a`) is the transpose of the right-hand matrix (`matrix_b`).

Remark 2. A noteworthy implication of this finding is that *matrix transposition can be efficiently achieved with no overhead.* For instance, to obtain the transpose of $C = A \cdot B$, one can simply swap the positions of the synthetic fragments corresponding to A and B. This results in computing $C^\top = B^\top \cdot A^\top$ without introducing any additional computational overhead. Notably, this optimization is not feasible using the standard API, as the compiler strictly enforces fragment position constraints.

3.2.2 *Free linear transformations by register shuffle.*

Synthetic fragments provide significant flexibility. Beyond adjusting operand positions of *synthetic fragments* for free matrix transposition, we can also rearrange internal registers to achieve certain matrix transformations at no additional cost.

A straightforward example is swapping the 0th and 1st elements, as well as the 4th and 5th elements, of `matrix_a` across all 32 threads. This effectively swaps the first and second columns of the matrix.

Remark 3. It should be noted that these transformations are not unrestricted. In Appendix A.1, we analyze what constitutes a "good" transformation — i.e., a meaningful linear transformation where the resulting matrix can be interpreted as the original matrix undergoing a structured transformation. We identify that such linear transformations must satisfy that: After register shuffling, all $(j, j+4)$ element pairs within each thread corresponds to elements in the original input matrix with an index stride of 128, for $0 \le j < 4$.

3.2.3 *Adjusting product fragments for linear transformation.*

As illustrated in Figure 1, there is a significant difference in memory locality between the operands and the results of matrix multiplication. For example, the `matrix_a` elements of Thread 0 change from the original 0x00, 0x01, **0x02, 0x03**, 0x80, 0x81, **0x82, 0x83** to 0x00, 0x01, 0x80, 0x81, **0x08, 0x09, 0x88, 0x89**. This means that the direct result of matrix multiplication does not conform to the layout requirements of `wmma.load`, making it impossible to perform chained matrix multiplications directly.

An insight is that the transformation from input to output in `wmma.mma` can be regarded as a special "transformation" to the corresponding matrix. However, as analyzed in the previous section, this is not a benign linear transformation, requiring additional effort to reorganize the layout, which may introduce inter-thread interaction overhead. We can reformulate this problem as how to turn this "bad" transformation into a "good" one (i.e., a linear transformation).

Remark 4. We denote the 8 registers in each thread as r_0, r_1, r_2, r_3, r_4, r_5, r_6, and r_7. Upon swapping (r_2, r_3) with (r_4, r_5) (note that this transformation itself does still not lead to a linear transformation, we denote it as RegShuf), the combined transformation of `wmma.mma` and RegShuf satisfies our analysis in the previous subsection, meaning that it becomes a linear transformation. When the resulting synthetic fragment serves as `matrix_a`, its equivalent form is $R \cdot P$. When it serves as `matrix_b`, the equivalent form is $P^{-1} \cdot R^{\top}$ (Note that P is a permutation matrix, i.e., $P^{-1} = P^{\top}$). The specific form of matrix P is detailed in Fig. 2.

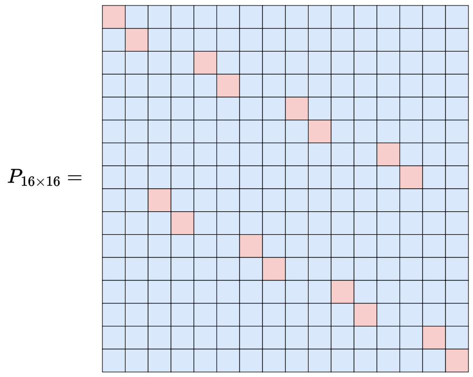

Figure 2: Permutation Matrix P (The blue blocks denote 0 and the red ones denote 1)

It is worth noting that the aforementioned approach is not the only way to achieve a linear transformation. For example, swapping (r_0, r_1) with (r_6, r_7) can produce a similar effect. The key requirement is that the composite transformation resulting from these operations satisfies the condition outlined in Section 3.2.2.

3.2.4 Chained matrix multiplications. Putting everything together, we can see how to implement chained matrix multiplications within registers and with no overhead. Consider a scenario: given matrices A, B, and C, we aim to compute $D = A \cdot B \cdot C$. After matrix A is loaded into a warp, each thread stores it in a register group $(a_0, a_1, \ldots, a_7)$, and matrices B and C are handled similarly.

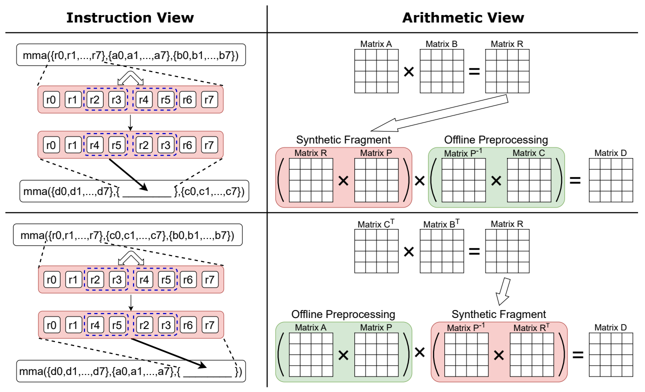

Figure 3: Two methods for bypassing standard APIs via synthetic fragments

Here we compare the provided two implementation methods for chained matrix multiplications are shown in Figure 3 and are elaborated as follows (assuming that the leftmost A is always laid out in the `matrix_a` format, while the rightmost C is always laid out in the `matrix_b` format):

The first method, assuming that B is laid out in the `matrix_b` format, takes the previous product as the left-hand operand. After the RegShuf operation, the resulting matrix $R = A \cdot B$ serves as `matrix_a`, and can be considered as $R' = R \cdot P$. We then prepare a precomputed matrix $C' = P^{-1} \cdot C$, so that the final result is

$$D = (A \cdot B \cdot P) \cdot (P^{-1} \cdot C) = A \cdot B \cdot C.$$

The second method, assuming that B is laid out in the `matrix_a` format, takes the previous product as the right-hand operand. This approach is slightly more involved because when it serves as `matrix_b`, it is treated as the transpose of the original matrix. Leveraging the properties we described earlier, we employ a simple trick: first, compute $R = C^{\top} \cdot B^{\top}$ (note that transposition incurs *no* overhead, as it only requires adjusting operand positions). Then, by applying the RegShuf operation, we obtain $R' = P^{-1} \cdot R$. We prepare a precomputed matrix $A' = A \cdot P$, leading to the final result:

$$D = (A \cdot P) \cdot \left[P^{-1} \cdot (C^{\top} \cdot B^{\top})^{\top}\right] = A \cdot B \cdot C.$$

One potential limitation of the above algorithm is that certain matrices (e.g., $A' = A \cdot P$, $C' = P^{-1} \cdot C$) require precomputation. However, in some practical applications, these matrices are often constants, making precomputation a non-issue. Moreover, even if these parameters need to be computed online, they are merely linear transformations and can be efficiently handled using register shuffling.

4 Memory-Less NTT and Polynomial Multiplication via Chained Matrix Multiplications

This section details how to use the aforementioned Tensor Core-based chained multiplication to implement (I)NTT and polynomial multiplication for ML-DSA and ML-KEM.

4.1 Memory-Less NTT via NTT decomposition

As much of the previous literature [10, 37, 41] has pointed out, NTT can be decomposed into matrix multiplications. This section will detail the implementation of a *memory-less* NTT. It is important to note that a *memory-less* NTT is not our ultimate goal; rather, it constitutes only a part of our broader *memory-less* framework.

In ML-KEM and ML-DSA, NTT is defined as a bijection NTT : $R_q \rightarrow R_q$ that maps $f \in R_q$ to its NTT representation:

$$\mathrm{NTT}(f) = \hat{f} = \sum_{i=0}^{255} \hat{f}_i X^i$$

These parameters have 512 distinct primitive roots, allowing for NTT computation with up to 256 points.

ConvKyber [41] proposed the method of decomposing NTTs into matrix multiplications, but this method imposes strict constraints on the form of matrix multiplication and cannot achieve general matrix multiplication. Based on this approach, the NTTs for both ML-KEM and ML-DSA, which are based on MLWE, can be unified into the following patterns.

The matrix formed by the 256 coefficients in the normal field of polynomial f, arranged from lowest to highest degree in a row-major order, is called the coefficient matrix F. Similarly, the matrix formed by the corresponding coefficients in the NTT field, arranged in the same way, is called the coefficient matrix F'.

By decomposing the exponents in the twiddle factor (denoted as ζ) of the NTT formula and leveraging the properties of the bit reverse operation, the matrix-based NTT formula can be derived [41], as shown in Equation (1). Here, P_1, P_2, and P_3 are the Twiddle Factor Matrices (TFMs) for NTT.

$$F' = [(P_1 \cdot F) \circ P_2] \cdot P_3 \tag{1}$$

When considering INTT, the derivation follows a similar approach, but due to the exchange of positions between indices i and j in the powers of ζ, the order of matrix multiplication is the opposite of that in NTT. The matrix-based INTT formula is shown in Equation (2) [41]. Here, Q_1, Q_2, and Q_3 are the TFMs for INTT.

$$F = Q_3 \cdot [(F' \cdot Q_1) \circ Q_2] \tag{2}$$

4.2 Memory-Less Polynomial Multiplication

Traditional implementations of lattice-based cryptographic algorithms, whether on CPU, GPU, or FPGA platforms, are designed with memory-intensive NTT/INTT as independent computational modules (including ConvKyber [41]), relying on memory for data interaction. However, this approach has significant limitations. Although ConvKyber [41] completes internal NTT/INTT calculations in registers, it still depends on memory for element-wise multiplication, thus failing to overcome the memory access bottleneck.

The situation has changed now. We have unlocked the NTT-EWM (element-wise multiplication)-INTT chain and directly focused on the more fundamental polynomial multiplication, surpassing the limitations of memory access, with the general chained matrix multiplications. With this technology, memory-less polynomial multiplication (ML-PolyMul) can be achieved.The most notable point is that, as illustrated in Figure 3, the synthetic fragment from RegShuf depends on whether the reconstructed register group serves as `matrix_a` or `matrix_b` in the instruction. The details of the algorithm are elaborated as follows:

(1) The polynomial coefficients must be decomposed into multiple precisions due to the 8-bit input limitation of the `wmma.mma` instruction. Given the algorithm's modulus $8(\theta-1) < \log_2 q \le 8\theta$ ($\theta \in \mathbb{Z}^+$), each input element a is decomposed into θ precision components $a_0, \dots, a_{\theta-1}$, such that $a = \sum_{i=0}^{\theta-1} a_i \cdot 2^{8i}$. When represented as a matrix multiplication of θ precision components, θ^2 `wmma.mma` instructions are required to generate θ^2 intermediate results, which are subsequently combined to form the complete element.

(2) In the second matrix multiplication in NTT and the first matrix multiplication in INTT, the product matrix serves as `matrix_a`. To address this, the corresponding TFMs P_3 and Q_1 need to be left-multiplied by P^{-1} during preprocessing. This step is essential to offset the equivalent transformation resulting from the product matrix $(R \cdot P)$ of the previous matrix multiplication.

(3) The second matrix multiplication in INTT involves the product matrix serving as `matrix_b`. Thus, the corresponding TFM Q_3 needs to be transposed and right-multiplied by P during preprocessing to offset the equivalent transformation of the product matrix $(P^{-1} \cdot R^{\top})$ from the previous matrix multiplication.

We leave the detailed algorithm in Algorithm 1 in Appendix A.2. This example illustrates the simplest form of polynomial multiplication in the MLWE problem. In practical algorithms, the specific forms of computation may involve matrix-vector polynomial multiplications or other forms. The intermediate processes of various forms of polynomial multiplication can be completed entirely within registers, and the specific implementation methods are introduced in the subsequent ML-KEM/ML-DSA implementations.

4.3 Number Representation and Modular Reduction

When applied ML-PolyMul to ML-KEM and ML-DSA implementations, additional details must be considered. Figure 4 outlines the implementation roadmap for ML-PolyMul, facing three detail processing issues elaborated in subsequent sections.

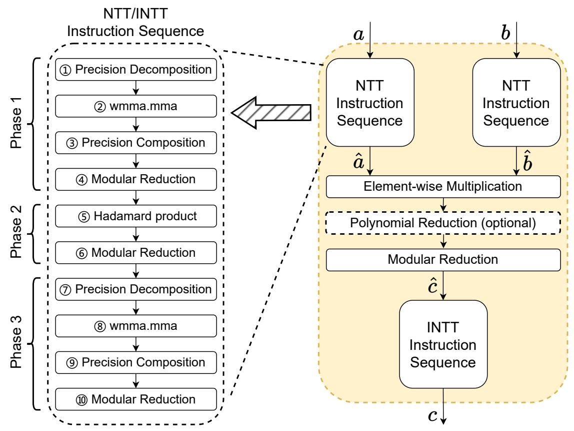

Figure 4: Implementation roadmap of ML-PolyMul (Polynomial reduction is only required for ML-KEM.)

4.3.1 *Precision decomposition.*

As described in Section 4.2, all multiplicands in the `wmma.mma` operation must undergo decomposition into 8-bit units. Hence, it is preferable to split into as few parts as possible to minimize computational overhead.

For ML-KEM, the modulus is $\log_2 q = 12$ bits wide, permitting decomposition into high 6 bits and low 6 bits, which reserves ample bits for potential signs and basic linear operations. Conversely, for ML-DSA, the modulus is $\log_2 q = 23$ bits wide. A straightforward approach involves decomposing it into the highest 7 bits, middle 8 bits, and lowest 8 bits through bit shifting.

However, this representation of ML-DSA coefficients introduces an unexpected issue. Since the two multiplicands for `wmma.mma` must both be `s8` or `u8`, the TFMs (with 23-bit values) cannot be directly represented by shifting to split the value into three `s8` variables (`s8` has only 7 bits for value representation), especially when the coefficients of the polynomial for NTT are already small signed integers stored in `s8` (e.g., coefficients of s_1 and s_2 are typically within ± 4).

To address this issue, we represent the coefficient a of ML-DSA as:

$$a = a_2 \cdot 2^{16} + a_1 \cdot 2^8 + a_0$$

where a_0, a_1, and a_2 are s8-type data (negative values indicate borrowing from higher-order units), with a_2 always being a positive value. This method allows the element to be split into three s8 data types, efficiently handling the representation of polynomial coefficients while avoiding redundant computational overhead.

4.3.2 Modular reduction. As described in Section 4.2, the product matrix of two matrices in multi-precision representation is computed using θ^2 wmma.mma instructions, yielding θ^2 intermediate results. Specifically, in ML-DSA, the coefficient a is represented as $a = \sum_{i=0}^{2} a_i \cdot 2^{8i}$. The product in multi-precision representation is denoted as $c = a \times b = A \cdot 2^{32} + B \cdot 2^{24} + C \cdot 2^{16} + D \cdot 2^8 + E$, where $A, B, \ldots, E$ are direct results from the wmma.mma instruction. These results represent the sums of the products of the various precision parts of a and b. Specifically, A is 18 bits, while B to E are 20 to 22 bits in size.

Reduction of values composed of $A, B, \ldots, E$ presents a special case in modular reduction tasks. Using b64 registers to accommodate all values is a straightforward but suboptimal solution. This approach introduces overhead due to register type conversion, as PTX assembly cannot directly compute the sum of b64 and b32 registers, requiring conversion of b32 to b64. Additionally, it necessitates replacing the 32-bit optimized Montgomery reduction instruction sequence with a 64-bit version, which is more time-consuming.

To optimize modular reduction in this scenario, we designed a multi-precision modular multiplication scheme using only b32 registers. The 50-bit product is split into a high part (*hi*, 22 bits) and a low part (*lo*, 28 bits), as follows:

$$A' = A << 4, D' = D << 8$$

$$B'_0 = (B \ \& \ 0\text{xf}) << 24, \ B'_1 = (B \ \& \ 0\text{xfffffff0}) >> 4$$

$$C'_0 = (C \ \& \ 0\text{xfff}) << 16, \ C'_1 = (C \ \& \ 0\text{xfffff000}) >> 12$$

$$hi = A' + B'_1 + C'_1, lo = B'_0 + C'_0 + D' + E$$

The reduction process computes:

$$\text{Mont\_Mul}(hi, 2^{28}) + \text{Mont\_Rec}(lo) \in (-2q, 2q)$$

While this method extends the reduction range to $(-2q, 2q)$ instead of the original $(-q, q)$ in Montgomery reduction, this does not introduce extra overhead, as shown in the following scenarios.

- In the first phase of (I)NTT, the reduced result is stored in 32-bit registers as an intermediate result for subsequent Montgomery multiplication.
- In the third phase of (I)NTT, the reduced result serves as the final output. For NTT, the polynomial is multiplied with others in the NTT domain, with no restrictions on coefficient range. For INTT, further addition/subtraction follows, with a final Barrett reduction to obtain coefficients in $[0, q)$ for norm checking.

In short, our multi-precision modular reduction maintains computational correctness without additional overhead.

4.3.3 Element-wise multiplication and polynomial reduction within registers. The selected modulus q of ML-KEM results in an incomplete NTT. Therefore, for a polynomial f in ML-KEM, the natural algebraic representation of $\text{NTT}(f) = \hat{f}$ consists of 128 polynomials of degree 1, expressed as:

$$\text{NTT}(f) = \hat{f} = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, \ldots, \hat{f}_{254} + \hat{f}_{255} X)$$

The product $f \cdot g$ of two elements $f, g \in R_q$ is based on $\hat{f} \circ \hat{g} = \hat{h}$, which involves 128 products computed as follows:

$$\hat{h}_{2i} + \hat{h}_{2i+1} X = (\hat{f}_{2i} + \hat{f}_{2i+1} X)(\hat{g}_{2i} + \hat{g}_{2i+1} X) \bmod X^2 - \zeta^{2\text{br}_7(i)+1}$$

Implementing such polynomial modular reduction in registers requires pairs of coefficients, $\hat{f}_{2i}$ and $\hat{f}_{2i+1}$, within the same register group of a thread (and g must also satisfy this condition correspondingly); failure to meet this condition compromises correctness. Additionally, the Tensor Core's design for 256-element matrix multiplication conflicts with the 128-point NTT and needs careful handling.

To resolve this, we arrange coefficients in a matrix using an interleaved odd-even pattern. This allows ML-PolyMul to be applied such that after the NTT phase, the data layout in registers contains pairs of coefficients $\hat{f}_{2i}$ and $\hat{f}_{2i+1}$ within the same thread. For instance, thread 0's register group $(f0, f1, \ldots, f7)$ holds $\hat{f}_0, \hat{f}_1, \ldots, \hat{f}_{136}$ and $\hat{f}_{137}$ of $\hat{f}$.

5 Fully Parallelized ML-KEM and ML-DSA

In fact, in previous sections, we have already achieved full parallelization of polynomial multiplication, as Tensor Core operations are executed in a warp-wise manner—meaning that a single warp with 32 threads can complete one polynomial multiplication. In the following subsections, we will describe how to further parallelize the entire ML-KEM and ML-DSA while preserving the memory-less property as much as possible.

5.1 Warp-wise ML-KEM Implementation

The two major computational components in algorithms based on the MLWE are hashing and polynomial operations.

5.1.1 Warp-wise SHA-3. The hashing algorithm (the SHA-3 family is chosen by ML-KEM and ML-DSA) is inherently "resistant to parallelization" by design. As a result, on GPUs, hashing computations have traditionally been executed on thread-wise, with each thread iterating through many rounds of the permutation function.

Lee *et al.* [20] proposed a warp-based method for implementing Keccak-f[1600] (the permutation function of the SHA-3 family). In this method, 25 threads within the warp model the 25 lanes of the state array, with each thread maintaining its own state and using shuffle instructions for inter-thread data exchange.

We adopted this approach in our warp-wise ML-KEM and ML-DSA with amelioration: instead of conditionally executing operations based on thread indices in the final stage (i.e. ι), we compute a mask (all 1s for thread 0, 0s otherwise) and perform XOR with this mask across all threads, thereby eliminating performance losses from thread synchronization.

5.1.2 The application of ML-PolyMul in ML-KEM. Figure 4 illustrates the basic implementation form of ML-PolyMul, which must be adapted to the specific polynomial operations in the algorithm for practical implementation. Therefore, it is necessary to first analyze the types of polynomial operations in the algorithm. For ML-KEM (see Section 2.2.1), the polynomial operations in KeyGen primarily involve matrix-vector polynomial multiplication (abbreviated as M-V_PolyMul), those in Dec focus on computing the inner product of two vectors (abbreviated as Inner_Prod), and Enc incorporates both of these polynomial operations. These two operations, M-V_PolyMul and Inner_Prod, constitute the main polynomial computations in ML-KEM.

We first introduce Dec for easier comprehension, whose core polynomial operation is presented as follows:

$$w \leftarrow v' - \mathrm{NTT}^{-1}(\hat{\mathrm{s}}^\top \circ \mathrm{NTT}(\mathrm{u}'))$$

The Inner_Prod in this operation aligns with the form depicted in Figure 4, making it an ideal scenario for applying ML-PolyMul and fully leveraging the potential of memory-less polynomials. In this case, only one set of registers (eight per thread) is required for accumulation, and iterating polynomial multiplications k times (depending on different parameter sets of ML-KEM, $k = 2/3/4$) over the same temporary registers efficiently utilizes hardware resources to achieve optimal performance.

We proceed to discuss the more complex Enc algorithm, which involves both M-V_PolyMul and Inner_Prod. Specifically, it requires computing the product of the polynomial vector y with the polynomial matrix $\hat{\mathrm{A}}$ and the polynomial vector $\hat{\mathrm{t}}$. This necessitates an extension of ML-PolyMul. As depicted in Figure 5, we have integrated these two polynomial operations in our implementation. The process begins with computing NTT(y), caching the result $\hat{\mathrm{y}}$ in k register groups. Subsequently, M-V_PolyMul with $\hat{\mathrm{A}}$ and Inner_Prod with $\hat{\mathrm{t}}$ are performed consecutively in registers. Finally, the results undergo INTT and subsequent computations directly in registers before being written back to memory. This approach eliminates internal memory access overhead, keeping the computations entirely within registers.

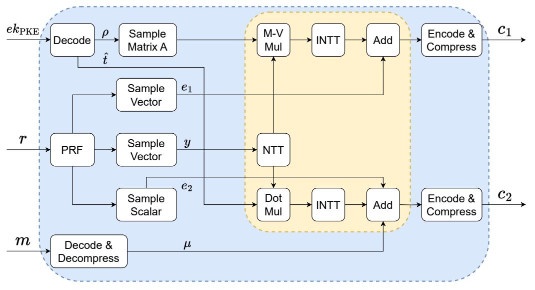

Figure 5: ML-KEM Enc based on ML-PolyMul (The blue box indicates that all computational loads are completed within the same fused kernel. The yellow boxes indicate polynomial multiplication operations implemented in registers.)

In the overall implementation, as shown in Figure 5, we pursue maximum kernel fusion by implementing the algorithm within a single kernel. This approach reduces kernel launch overhead and enhances data access efficiency.

5.2 Warp-wise ML-DSA Implementation

The key challenge in implementing ML-DSA lies in the Signing process, which requires candidates to pass a series of checks in a loop until all checks are passed. This process is called the *rejection sampling loop* (for NIST security levels 2, 3, and 5, the expected loop repetitions for signatures are 4.25, 5.1, and 3.85, respectively). In this section, we first discuss the scheduling mechanism for this loop and then explore how ML-PolyMul is applied in ML-DSA.

5.2.1 Scheduling mechanism for rejection sampling loop. When implementing ML-DSA on GPUs, the *rejection sampling loop*'s inherent variability introduces significant warp-to-warp divergence. The warps processing instances with fewer repetitions become idle while waiting for slower warps with higher repetition counts. This leads to underutilized compute units and degraded throughput.

To address this, Shen *et al.* [32] proposed a *Dynamic Task Scheduling* (DTS) strategy, where a fixed-size execution window iteratively processes all pending instances, performing one trial per instance per round and re-enqueuing failed instances. While this reduces inter-warp diversity, it incurs substantial memory overhead: each round reloads instance-specific public parameters (e.g., matrix A, vectors s_1, s_2, t_0) from memory, as these values are not cached across rounds.

To address this, we propose the *Task Fusion Scheduling* (TFS) strategy, which leverages the statistical properties of the reject loop. The key insight is to assign each warp to process t sequential ML-DSA instances, starting a new instance only after the previous one succeeds.

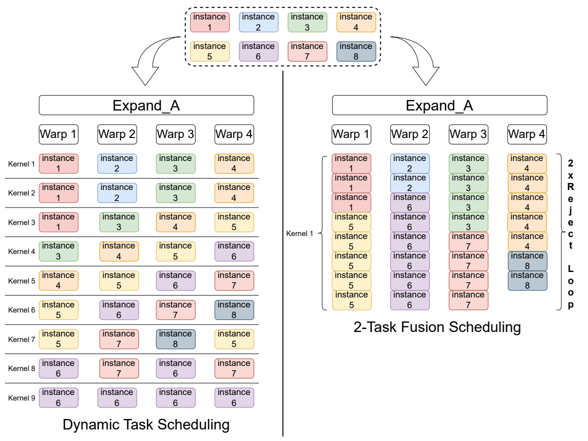

Figure 6: Comparison of two scheduling strategies for rejection sampling loop

As depicted in Figure 6, when scheduling eight signature instances across four warps, DTS and TFS are employed respectively. Under DTS, the four warps synchronously scan and process the remaining instances in rounds. If fewer than four instances remain, they are reused. Conversely, TFS assigns each warp to process multiple instances within a single kernel until success, such as completing t instances (illustrated with $t = 2$). As can be seen from the figure, TFS offers the following three key advantages:

(1) TFS processes t instances in one kernel launch, not only reducing the average number of iterations but also avoiding the repeated kernel invocation overhead observed in DTS.
(2) Each instance in TFS loads key data from global memory only one time, whereas DTS necessitates re-reading keys each round, thereby significantly reducing memory access costs.
(3) TFS generates one valid signature per instance, unlike DTS, where an instance may yield multiple signatures, thus preventing resource waste.

Experimental results indicate optimal throughput at $t = 4$, with performance gradually diminishing for $t = 8, 16, 32$. We can theoretically characterize this using the negative binomial distribution, whose convergence properties demonstrate the minimization of inter-warp divergence. The empirical equilibrium point at $t = 4$ reflects a hardware-aware equilibrium between statistical optimization and GPU resource constraints.

5.2.2 The application of ML-PolyMul in ML-DSA. The Signing is central to ML-DSA and is its most distinctive component. We take it as an example to discuss the application of ML-PolyMul in ML-DSA. Analyzing the Signing (see Section 2.2.2), it comprises two types of polynomial operations: M-V_PolyMul and uniquely three consecutive scalar-vector polynomial multiplications (abbreviated as S-V_PolyMul).

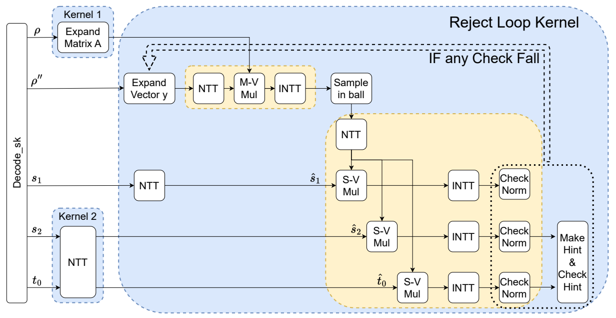

Figure 7: ML-DSA Signing based on ML-PolyMul (The blue box represents computations within the same kernel. The yellow boxes indicate polynomial multiplication operations implemented in registers.)

Based on the summary of polynomial operation types, the implementation of Signing is shown in Figure 7. For M-V_PolyMul, we apply ML-PolyMul in a manner similar to Enc in ML-KEM (Section 5.1.2), but here, the number of register groups used for caching is l (depending on ML-DSA's parameter sets, $l = 4/5/7$). For the three consecutive S-V_PolyMul operations, we fuse them and implement them in registers. Specifically, these three S-V_PolyMul operations involve multiplying $\hat{c}$ with $\hat{s}_1$, $\hat{s}_2$, and $\hat{t}_0$. Thus, only one register group is required to cache $\hat{c}$, enabling the application of ML-PolyMul (which involves one NTT and one to three EWM and INTT operations, depending on the check results) to obtain the product polynomials and perform norm checks.

In the overall implementation of Signing, considering that the loop iterates multiple times, we implement the expansion of matrix A and the NTT of s_2 and t_0 as separate kernels. Note that we specifically schedule the NTT of s_1 within the reject loop kernel. This is because cs_1 is the first check in the loop and has the highest failure rate (approximately 40%). By placing the NTT of s_1 in the reject loop kernel, memory access overhead associated with reloading when the check fails can be optimized.

6 Performance Evaluation & Discussion

In this section, we evaluate the performance of ML-Cube. Our evaluation platform consists of an Intel Xeon Silver 4410Y CPU with 24 cores at a base frequency of 2.0 GHz and an NVIDIA GeForce RTX 4090 GPU, which includes 128 Streaming Multiprocessors (SMs) and 512 Tensor Cores. The software environment is Ubuntu 20.04.6 LTS with Linux kernel version 5.15.0-125-generic. The CUDA C++ and PTX code is compiled and executed using the CUDA Toolkit version 12.1 with the nvcc compiler.

6.1 Evaluation and Comparison

Our evaluation answers the following research questions.

RQ1: How much performance improvement can the memory-less design achieve for polynomial multiplication?

Previous works have only indirectly reflected polynomial multiplication through (I)NTT performance evaluations. To address this, we designed two experiments aimed at demonstrating the performance improvement of memory-less polynomial multiplication.

In the first experiment, we compared the performance of (I)NTT in our ML-KEM and ML-DSA implementations with those in their SOTA implementations (without Tensor Core), as shown in Table 1. As indicated in the table, our implementation shows an approximate 3-fold and 2-fold increase for ML-KEM and ML-DSA, respectively.

In the second experiment, we implemented the inner product of two polynomial vectors (with $k = 4$ as the parameter in ML-KEM at Level 5) using both the SOTA and our memory-less approaches, as shown in Table 2. Compared to the SOTA implementation, our memory-less method reduces latency to approximately $1/10$, highlighting its effectiveness in overcoming memory access limitations.

Table 2: Comparison of the performance of polynomial vector inner products (evaluated at batch size 8192, where N: NTT, EWM: Element-wise Multiplication, IN: INTT, G: Global memory.)

Zhou *et al.* [41]	Operation	G→N→G	G→EWM→G	G→IN→G	Sum
	Latency (μs)	31.64	14.45	3.03	49.12
This work	Operation	G→Memory-less Polynomial Multiplication→G			
	Latency (μs)	4.79			4.79

RQ2: What is the minimum batch size that the warp-wise implementation can achieve while maintaining the (almost) peak throughput?

This experiment aims to demonstrate that our warp-wise implementation can effectively lower the threshold to achieve peak

Table 1: Performance comparison of this work with CUDA core-Based schemes in (I)NTT performance

ML-KEM ($\log q = 12, \theta = 2$)°				ML-DSA ($\log q = 23, \theta = 3$)°			
Scheme	Operation	Input Precision (bit)	Throughput (kops)	Scheme	Operation	Input Precision (bit)	Throughput (kops)
Ji *et al.* [17]*	NTT	12	189,788	Shen *et al.* (QWarp) [32]	NTT	23	240,991
	INTT	12	194,532		INTT	23	213,867
This work	NTT	8	676,910 **(3.56×)**	**This work**	NTT	8	654,428 **(2.72×)**
		12	571,808 **(3.01×)**			23	490,170 **(2.03×)**
	INTT	12	584,783 **(3.01×)**		INTT	23	485,827 **(2.27×)**

* The data is scaled to R4090 performance by multiplying with the ratio of TFLOPS of R4090 to R3080, which is 82.6/29.77 ≈ 2.77.
° In the given algorithm, θ represents the number of parts into which a coefficient is decomposed during precision decomposition.

throughput, thereby reducing latency with minimal loss in throughput. In this experiment, we evaluated our warp-wise implementation and compared it with the previous SOTA thread-wise implementation.

We first illustrate how the throughput of the warp-wise and thread-wise ML-KEM varies with increasing batch size, as shown in Figure 8. The results indicate that the warp-wise implementation can reach approximately 90% of the peak throughput at a very small batch size (1024), about ten times lower than the threshold of the thread-wise implementation, which reaches its peak at a batch size of around 10240.

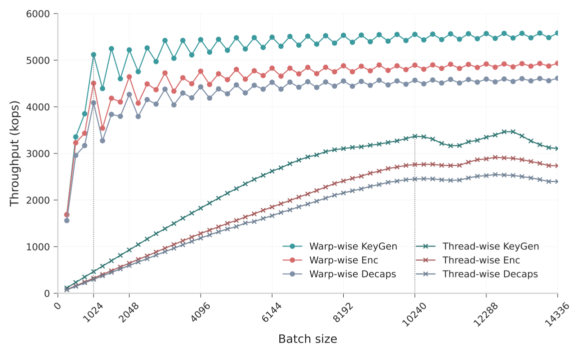

Figure 8: Thread-wise vs. warp-wise ML-KEM (Level 5): throughput variation comparison with batch size (Thread-wise ML-KEM evaluated with fixed Block_size 256 and increasing Grid_size; warp-wise ML-KEM evaluated with fixed Block_size 128 and increasing Grid_size.)

Furthermore, we compare the performance data of the two implementations under exact batch sizes, as shown in Table 3. The warp-wise implementation achieves over 90% reduction in latency with about one-tenth of the batch size required by the thread-wise approach, while also delivering a 1.5× to 3.6× performance speed-up compared to Zhou *et al.* [41]. In brief, besides greatly enhancing the implementation's usability, our implementation reduced performance threshold also significantly cuts down on latency.

RQ3: How much can the signature performance of ML-DSA be improved by adopting the *Task Fusion Scheduling* and ML-PolyMul technologies?

This experiment aims to demonstrate the impact of our proposed technologies on the performance improvement of ML-DSA. In this experiment, we compare the performance of ML-Cube with the SOTA implementations of two types of signatures: the full-process and the server-oriented (referred to as such in [33]), as shown in Table 4.

The results indicate that, for the full signature, ML-Cube achieves more than 10% improvement compared to [32]. Meanwhile, in terms of the signature with precomputed private key, ML-Cube shows a 30% to 55% enhancement in comparison to [33].

It should be noted that in practice, *the significant performance improvement observed in the the server-oriented case is particularly meaningful*, as GPUs are predominantly used in server-oriented implementations and the private key derivation operations can be precomputed offline. Importantly, the observed performance gains primarily stem from our TFS strategy and memory-less design. These findings highlight the practical value of architecture-aware cryptographic design and offer important insights for building high-throughput, low-latency PQC services on heterogeneous platforms.

RQ4: How much improvement do ML-KEM and ML-DSA offer compared to the current SOTA implementations on other platforms?

This experiment aims to demonstrate that our ML-KEM and ML-DSA implementations fully leverage the computational power of GPUs to set new performance records. In this experiment, we compare the peak performance of ML-KEM and ML-DSA at Level 5 with SOTA implementations on various platforms (CPU and FPGA), with results listed in Table 5 and Table 6. From these tables, it is evident that our ML accelerator-based implementations, both for ML-KEM and ML-DSA, achieve throughput that is one to two orders of magnitude higher than those of existing works on various platforms.

Table 3: Thread-wise vs. warp-wise ML-KEM: latency and throughput comparison at different security levels (Zhou *et al.* [41] evaluated at batch size 32,768; this work evaluated at batch size 4,096.)

Level	1				3				5			
Latency (ms)												
Scheme	KeyGen	Enc/Encaps*	Dec	Decaps°	KeyGen	Enc/Encaps*	Dec	Decaps°	KeyGen	Enc/Encaps*	Dec	Decaps°
Zhou *et al.* [41]	3.42	4.49	0.81	5.30	6.39	7.88	1.31	9.19	10.09	11.80	1.74	13.54
This work	0.28 **(-91.8%)**	0.30 **(-93.3%)**	0.04 **(-95.1%)**	0.34 **(-93.6%)**	0.45 **(-92.9%)**	0.52 **(-93.4%)**	0.05 **(-96.2%)**	0.57 **(-93.8%)**	0.76 **(-92.5%)**	0.86 **(-92.7%)**	0.06 **(-96.6%)**	0.92 **(-93.2%)**
Throughput (kops)												
Scheme	KeyGen	Enc/Encaps*	Dec	Decaps°	KeyGen	Enc/Encaps*	Dec	Decaps°	KeyGen	Enc/Encaps*	Dec	Decaps°
Zhou *et al.* [41]	9,564	7,303	40,095	6,177	5,126	4,159	24,959	3,564	3,248	2,775	18,836	2,418
This work	14,750 **(1.54×)**	13,795 **(1.89×)**	106,243 **(2.65×)**	12,210 **(1.97×)**	9,021 **(1.76×)**	7,826 **(1.88×)**	83,279 **(3.34×)**	7,153 **(2.00×)**	5,417 **(1.67×)**	4,784 **(1.72×)**	67,340 **(3.58×)**	4,466 **(1.85×)**

* 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.

Table 4: Performance evaluation of two types of ML-DSA Signing

Level	2	3	5
Full Signature (ops)			
Shen *et al.* [32]	984,803	649,498	488,006
This work	1,153,853 **(1.17×)**	735,731 **(1.13×)**	564,184 **(1.16×)**
Server-oriented Signature (ops)			
Shen *et al.* [33]	1,080,871	884,195	785,277
This work	1,669,645 **(1.55×)**	1,148,682 **(1.30×)**	1,112,853 **(1.42×)**

Table 5: Performance comparison of ML-KEM across multiple platforms at Level 5

Scheme	Platform	KeyGen (kops)	Enc/Encaps* (kops)	Dec (kops)	Decaps (kops)
Ni *et al.*§ [27]	Artix-7 @263MHz	51.8 **(104×)**	44.2 **(108×)**	/	34.8 **(128×)**
Becker *et al.*† [5]	Cortex-A72 @1.5GHz	9.6 **(564×)**	7.8 **(613×)**	/	8.1 **(551×)**
Aikata *et al.*§ [1]	US+Z @270MHz	29.7 **(182×)**	23.8 **(201×)**	/	19.4 **(230×)**
Avanzi *et al.*† [4]	i7-4770K @3492MHz	47.6 **(114×)**	36 **(132×)**	44.2 **(1,523×)**	/
This work	RTX4090 @2.7GHz	5,417	4,784	67,340	4,466°

* 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 data is derived from reported cycle count and the device's frequency.
§ The data is derived from reported latency.

6.2 Discussion

Besides its performance benefits, the memory-less oriented design of ML-Cube inherently offers superior side-channel resistance compared to other schemes that rely on global or shared memory for thread data exchange (e.g., [32]).

6.2.1 Mitigation to Side channel attacks. Our memory-less design leverages the inherent characteristics of the Tensor Core, which can be treated as an atomic instruction execution unit that operates with a fixed number of cycles. Furthermore, because most computations are performed entirely within registers, there is minimal timing variation due to cache hits or misses. For Flush+Reload [39] attacks, the fact that ML-Cube executes operations solely within registers renders it naturally immune to data-access-driven attacks (e.g., adversaries probe LUT memory access patterns) or control-flow-driven attacks (e.g., those targeting ECDSA point multiplication code). In addition, the task fusion design in our ML-DSA implementation makes it extremely difficult for an attacker to distinguish delays caused by individual operations.

Table 6: Performance comparison of ML-DSA (full signature) across multiple platforms at Level 5

Scheme	Platform	KeyGen (kops)	Sign (kops)	Verify (kops)
Truong *et al.*§ [35]	VU+ @300MHz	36.50 **(27×)**	5.90 **(96×)**	37.17 **(27×)**
Zhao *et al.* [40]	Artix-7 @96.9MHz	11.05 **(89×)**	1.98 **(285×)**	10.72 **(95×)**
Aikata *et al.*§ [1]	US+Z @270MHz	6.80 **(145×)**	5.03 **(112×)**	5.81 **(176×)**
Becker *et al.*† [5]	Cortex-A72 @1.5GHz	1.92 **(516×)**	1.04 **(542×)**	1.96 **(523×)**
This work	RTX4090 @2.7GHz	991.12	564.18	1,026.82

† The data is derived from reported cycle count and the device's frequency.
§ The data is derived from reported latency.

6.2.2 Mitigation to Memory disclosure attacks. Attacks such as cold-boot and DMA attacks [34] primarily target off-chip memory. Traditional schemes (e.g., [32, 33]) typically rely on global memory for data synchronization, which exposes them to such threats. In contrast, the memory-less nature of ML-Cube significantly alleviates these vulnerabilities. With additional fine-grained control—achieving a register-bound design similar to TRESOR [23], PRIME [12], or PixelVault [36], or even a cache-bound design as seen in Copker [13] and Mimosa [14]—it is possible to further guard against memory disclosure attacks.

7 Conclusion

This paper introduces ML-Cube, a memory-less framework that leverages ML accelerators for cryptographic tasks. By analyzing ML accelerator architectures, we designed custom transformations for memory-less NTT/INTT and polynomial multiplication, reducing external memory use, latency, and parallelism overhead. Our approach overcomes black-box limitations and enhances security by mitigating memory disclosure and cache side-channel risks. Experiments show significant speedups in ML-KEM and ML-DSA. Future work will expand ML-Cube to more accelerators and explore its use in post-quantum cryptography and FHE.

Acknowledgments

This work is supported in part by the National Cryptologic Science Fund of China No. 2025NCSF02005, the National Natural Science Foundation of China No. 61902392. Fangyu Zheng is the corresponding author (*E-mail: zhengfangyu@ucas.ac.cn*).

References

[1] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. 2022. KaLi: A crystal for post-quantum security using Kyber and Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers* 70, 2 (2022), 747–758.

[2] Miklós Ajtai. 1996. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 99–108.

[3] AMD. 2024. AMD Instinct MI300X Accelerator. https://www.amd.com/en/products/instinct-accelerators-mi300x Accessed: March 19, 2025.

[4] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2019. CRYSTALS-Kyber algorithm specifications and supporting documentation. *NIST PQC Round* 2, 4 (2019), 1–43.

[5] Hanno Becker, Vincent Hwang, Matthias J Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. 2021. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and apple M1. *Cryptology ePrint Archive* (2021).

[6] Adrien Benamira, David Gerault, Thomas Peyrin, and Quan Quan Tan. 2021. A Deeper Look at Machine Learning-Based Cryptanalysis. In *Advances in Cryptology – EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I* (Zagreb, Croatia). Springer-Verlag, Berlin, Heidelberg, 805–835. doi:10.1007/978-3-030-77870-5_28

[7] Google Cloud. 2024. Tensor Processing Units (TPUs). https://cloud.google.com/tpu Accessed: March 19, 2025.

[8] Intel Corporation. 2024. Intel Deep Learning Boost (VNNI). https://www.intel.com/content/www/us/en/architecture-and-technology/deep-learning-boost.html Accessed: March 19, 2025.

[9] NVIDIA Corporation. 2024. NVIDIA Tensor Core GPU Architecture. https://www.nvidia.com/en-us/data-center/tensor-cores/ Accessed: March 19, 2025.

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

[11] Eiichiro Fujisaki and Tatsuaki Okamoto. 1999. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*. Springer, 537–554.

[12] Behrad Garmany and Tilo Müller. 2013. PRIME: Private RSA infrastructure for memory-less encryption. In *Proceedings of the 29th Annual Computer Security Applications Conference*. 149–158.

[13] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. 2014. Copker: Computing with private keys without RAM.. In *NDSS*. 23–26.

[14] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 3–19.

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

[16] Apple Inc. 2024. Apple Neural Engine. https://www.apple.com/newsroom/2020/11/introducing-the-next-generation-of-mac/ Accessed: March 19, 2025.

[17] Xinyi Ji, Jiankuo Dong, Tonggui Deng, Pinchang Zhang, Jiafeng Hua, and Fu Xiao. 2024. HI-Kyber: A novel high-performance implementation scheme of Kyber based on GPU. *IEEE Transactions on Parallel and Distributed Systems* (2024).

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

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

[20] Wai-Kong Lee, Xian-Fu Wong, Bok-Min Goi, and Raphael C-W Phan. 2017. CUDA-SSL: SSL/TLS accelerated by GPU. In *2017 International Carnahan Conference on Security Technology (ICCST)*. IEEE, 1–6.

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

[22] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[23] Tilo Müller, Felix C Freiling, and Andreas Dewald. 2011. TRESOR Runs encryption securely outside RAM. In *20th USENIX Security Symposium (USENIX Security 11)*.

[24] National Institute of Standards and Technology. 2024. *Module-Lattice-Based Digital Signature Standard*. Federal Information Processing Standards Publication 204. National Institute of Standards and Technology. DOI: https://doi.org/10.6028/NIST.FIPS.204.

[25] National Institute of Standards and Technology. 2024. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Federal Information Processing Standards Publication 203. National Institute of Standards and Technology. DOI: https://doi.org/10.6028/NIST.FIPS.203.

[26] National Institute of Standards and Technology. 2024. *Stateless Hash-Based Digital Signature Standard*. Federal Information Processing Standards Publication 205. National Institute of Standards and Technology. DOI: https://doi.org/10.6028/NIST.FIPS.205.

[27] Ziying Ni, Ayesha Khalid, Weiqiang Liu, and Máire O'Neill. 2025. A highly hardware efficient ML-KEM accelerator with optimised architectural layers. *ACM Transactions on Embedded Computing Systems* 24, 2 (2025), 1–24.

[28] NVIDIA. 2025. CUDA C programming guide 12.8. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[29] NVIDIA. 2025. Parallel thread execution ISA version 8.7. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[30] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. 2019. The curse of class imbalance and conflicting metrics with machine learning for side-channel evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 209–237.

[31] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)* 56, 6 (2009), 1–40.

[32] Shiyu Shen, Hao Yang, Wangchen Dai, Hong Zhang, Zhe Liu, and Yunlei Zhao. 2024. High-throughput GPU implementation of Dilithium post-quantum digital signature. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[33] Shiyu Shen, Hao Yang, Wenqian Li, and Yunlei Zhao. 2024. cuML-DSA: Optimized Signing Procedure and Server-Oriented GPU Design for ML-DSA. *IEEE Transactions on Dependable and Secure Computing* (2024).

[34] Patrick Stewin and Iurii Bystrov. 2012. Understanding DMA malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 21–41.

[35] Quang Dang Truong, Phap Ngoc Duong, and Hanho Lee. 2024. Efficient Low-Latency Hardware Architecture for Module-Lattice-Based Digital Signature Standard. *IEEE Access* (2024).

[36] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. PixelVault: Using GPUs for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1131–1142.

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

[38] Lipeng Wan, Fangyu Zheng, and Jingqiang Lin. 2021. TESLAC: Accelerating lattice-based cryptography with AI accelerator. Springer, 249–269.

[39] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX security symposium (USENIX security 14)*. 719–732.

[40] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2022. A compact and high-performance hardware architecture for CRYSTALS-Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022), 270–295.

[41] Tian Zhou, Fangyu Zheng, Guang Fan, Lipeng Wan, Wenxu Tang, Yixuan Song, Yi Bian, and Jingqiang Lin. 2024. ConvKyber: unleashing the power of AI accelerators for faster Kyber with novel iteration-based approaches. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 2 (2024), 25–63.

A Appendix

A.1 Interplay Between Register-shuffling and Matrix-permutation

As our research on the internal mechanisms of Tensor Cores reveals in Section 3, the CUDA C++ `wmma` API treats `fragments` as opaque objects, which cannot be directly modified. Concurrently, the underlying PTX instructions (i.e., `wmma.load`, `wmma.mma`, and `wmma.store`) perform fixed hardware operations. Therefore, at the software level, the only controllable aspect for programmers is the direct manipulation of registers within a warp.

The central idea for addressing the limitations of the official API is to directly shuffle the product registers (denoted as $\{r0, r1, \dots, r7\}$) held by each thread after a `wmma.mma` operation. This allows us to construct an input `fragment` that can be used for the subsequent `wmma.mma` operation. Throughout this paper, we denote this register-level reordering operation as RegShuf. However, this operation itself introduces a complex, typically nonlinear, transformation to the matrix data stored in the registers. Our goal in this section is to logically bypass the inherent nonlinear transformations of `wmma.store` and `wmma.load` and, by composing a meticulously designed RegShuf transformation, ensure that the final equivalent transformation is linear, thereby obtaining a usable composite fragment.

Upon analysis, we found that reordering registers introduces a complex element permutation on the original matrix. However, if certain specific requirements are met, the composite result may be an elementary row or column permutation of the original matrix. For instance, swapping the register pairs $(r2, r3)$ with $(r4, r5)$ can achieve a column permutation of the original matrix. In contrast, a seemingly simpler operation, such as merely swapping $r1$ and $r2$, can lead to a complicated and nonlinear data transformation. Therefore, to find an effective RegShuf scheme, we must jointly analyze its effects with the permutation behavior introduced by `wmma.load`.

Since the data loading for `matrix_a` and `matrix_b` are transposed, we analyze `matrix_a` as a case study, and the conclusions can be generalized to `matrix_b`. We observe that the `wmma.load` operation can be viewed as a two-step process: first, an elementary row permutation is applied, where we permute rows i and $i+8$ of the input matrix A to rows $2i$ and $2i+1$ (for $0 \le i < 8$) to obtain matrix A'. Second, we partition matrix A' into 2×16 blocks of two rows each and apply an element permutation to each submatrix. We denote the latter permutation as the mapping σ.

For the first step, let the matrix row permutation here be $\pi \in S_{16}$:

$$\pi = \begin{pmatrix} 0 & 1 & 2 & \cdots & 7 & 8 & 9 & \cdots & 15 \\ 0 & 2 & 4 & \cdots & 14 & 1 & 3 & \cdots & 15 \end{pmatrix}$$

The permutation matrix corresponding to π is denoted as P_π, with elements $[P_\pi]_{i,j} = \delta_{j,\pi(i)}$, where δ is the Kronecker delta. The permuted matrix is $A' = P_\pi A \triangleq (M_0, \dots, M_7)^\top$, where M_i is the 2×16 submatrix composed of rows $2i$ and $2i+1$ of the permuted matrix. Thus, the `wmma.load` transformation for `matrix_a` is equivalent to:

$$\texttt{wmma.load} : A \mapsto (\sigma(M_0), \dots, \sigma(M_7))^\top$$

Since register shuffling only occurs within the same thread (i.e., within two 8-element subvectors split from a row vector of the matrix) and different threads execute the same register shuffling command, we can assume that each 2×16 submatrix uses the same register shuffle mapping ψ.

Therefore, a complete (`wmma.load`, `wmma.mma`, `wmma.load`$^{-1}$) execution path can be abstracted to the following mapping:

$$\texttt{load}^{-1} \circ \texttt{mma} \circ \texttt{load} : A \mapsto P_{\pi^{-1}} \left(\sigma^{-1} \circ \psi \circ \sigma(M_0), \dots, \sigma^{-1} \circ \psi \circ \sigma(M_7)\right)^\top$$

Consequently, we simplify the analysis of the complex composite transformation to an analysis of the transformation on the submatrix M_i.

A.1.1 Column Permutation. Since the mappings from M_0 to M_7 are identical, we omit the subscripts to avoid clutter. Let $\sigma : M \mapsto W$, $\psi : W \mapsto W'$, and $\sigma^{-1} : W' \mapsto M'$. Because we need to investigate what kind of register shuffling can make the resulting matrix M' a column permutation of M, we first analyze the inverse transformation σ^{-1}.

Let the shuffled register matrix be $W' \in \mathbb{R}^{2\times16}$ with elements $[W']_{i,j} = w'_{16i+j}$. The elements of the matrix $M' \in \mathbb{R}^{2\times16}$ after the σ^{-1} mapping are $[M']_{i,j} = m'_k$ (where $k = 16i + j, 0 \le k < 32$). The following inductive relation holds:

$$m'_k \triangleq w'_{\eta(k)} = w'_{8\lfloor k/4\rfloor - 28\lfloor k/16\rfloor + (k \bmod 4)} \quad \text{for } 0 \le k < 32.$$

Evidently, a necessary and sufficient condition for M' to be a valid column-permution of the matrix $M = [m_{16i+j}]_{0\le i<2, 0\le j<16} \in \mathbb{R}^{2\times16}$ is that, for any permutation function σ defined on $[0, 16)$, if $m'_k = m_{\sigma(k)}$, then we must have $m'_{k+16} = m_{\sigma(k)+16}$. This constraint ensures that all originally vertically adjacent elements in M remain vertically adjacent after the transformation.

Substituting the inductive relation into m'_{k+16}, we obtain:

$$m'_{k+16} = w'_{8\lfloor (k+16)/4\rfloor - 28\lfloor (k+16)/16\rfloor + ((k+16) \bmod 4)} = w'_{\eta(k)+4},$$

This implies that in the shuffled register matrix W', the element pairs $(w'_{\eta(k)}, w'_{\eta(k)+4})$ must correspond to vertically adjacent elements in the original matrix M. For $0 \le k < 16$, the values of $\eta(k)$ fall within the ranges $[0, 3]$, $[8, 11]$, $[16, 19]$, and $[24, 27]$.

For a register shuffle that meets the above requirements, the combined transformation $\sigma^{-1} \circ \psi \circ \sigma$ constitutes an elementary column transformation on the submatrix M, which is equivalent to right-multiplying M by a column permutation matrix, denoted as $MP_{\sigma^{-1}\circ\psi\circ\sigma}$. Consequently, for the entire 16×16 input matrix A, the equivalent linear transformation is $P_{\pi^{-1}} P_\pi A P_{\sigma^{-1}\circ\psi\circ\sigma} = AP_{\sigma^{-1}\circ\psi\circ\sigma}$, which is a column permutation identical to that on matrix M.

A.1.2 Row Permutation. Since the analyzed matrix M has only two rows, the only possible row permutation is to swap them. This operation can be achieved through a simple register shuffle by exchanging the register groups $(r0, r1, r2, r3)$ and $(r4, r5, r6, r7)$.

For this specific register shuffle, the combined transformation $\sigma^{-1} \circ \psi \circ \sigma$ constitutes an elementary row transformation on the submatrix M, which is equivalent to left-multiplying M by a row permutation matrix, denoted as $P_{\sigma^{-1}\circ\psi\circ\sigma} M$. For the entire 16×16 input matrix A, the equivalent linear transformation is

$$P_{\pi^{-1}} \operatorname{diag}\left(P_{\sigma^{-1}\circ\psi\circ\sigma}, \dots, P_{\sigma^{-1}\circ\psi\circ\sigma}\right) P_\pi A$$

Algorithm 1 memory-less polynomial multiplication based on Tensor Core

```
1: Input: polynomial $A, B$
2: Output: polynomial $C = A \cdot B$

3: procedure TFM_OFFLINE
4:     $(P_1, P_2, P_3) \leftarrow$ NTT_TFM
5:     $(Q_1, Q_2, Q_3) \leftarrow$ INTT_TFM
6:     $P_3' = P^{-1} \cdot P_3$
7:     $Q_1' = P^{-1} \cdot Q_1$
8:     $Q_3' = Q_3^\top \cdot P$
9:     NTT_TFMs$\leftarrow (P_1, P_2, P_3')$
10:    INTT_TFMs$\leftarrow (Q_1', Q_2, Q_3')$
11: end procedure

12: procedure NTT($A[1]$)
        $P_1\_Reg[\theta] \leftarrow P_1[\theta]$ (hard coded)
        $P_2\_Reg[1] \leftarrow P_2[1]$ (hard coded)
        $P_3\_Reg[\theta] \leftarrow P_3'[\theta]$ (hard coded)
13:    $A\_Reg[\theta] \leftarrow$ Decomposition($A\_Reg[1]$)
14:    $T1\_Reg[\theta^2] \leftarrow$ mma($P_1\_Reg[\theta], A\_Reg[\theta]$)
15:    $T2\_Reg[1] \leftarrow$ Composition($T1\_Reg[\theta^2]$)
16:    $T3\_Reg[1] \leftarrow$ Element_Mul($T2\_Reg[1], P_2\_Reg[1]$)
17:    $T4\_Reg[1] \leftarrow$ Register_shuffle($T3\_Reg[1]$)
                                          ▷ $T4\_Reg[1] = T3\_Reg[1] \cdot P$
18:    $T5\_Reg[\theta] \leftarrow$ Decomposition($T4\_Reg[1]$)
19:    $T6\_Reg[\theta^2] \leftarrow$ mma($T5\_Reg[\theta], P_3'\_Reg[\theta]$)
20:    $A'\_Reg[1] \leftarrow$ Composition($T6\_Reg[\theta^2]$)
21:    Return: $A'\_Reg[1]$
22: end procedure

23: procedure INTT($A'\_reg[1]$)
        $Q_1\_Reg[\theta] \leftarrow Q_1'[\theta]$ (hard coded)
        $Q_2\_Reg[1] \leftarrow Q_2[1]$ (hard coded)
        $Q_3\_Reg[\theta] \leftarrow Q_3'[\theta]$ (hard coded)
24:    $T1\_Reg[1] \leftarrow$ Register_shuffle($A'\_Reg[1]$)
                                          ▷ $T1\_Reg[1] = A'\_Reg[1] \cdot P$
25:    $T2\_Reg[\theta] \leftarrow$ Decomposition($T1\_Reg[1]$)
26:    $T3\_Reg[\theta^2] \leftarrow$ mma($T2\_Reg[\theta], Q_1\_Reg[\theta]$)
27:    $T4\_Reg[1] \leftarrow$ Composition($T3\_Reg[\theta^2]$)
28:    $T5\_Reg[1] \leftarrow$ Element_Mul($T4\_Reg[1], Q_2\_Reg[1]$)
29:    $T6\_Reg[1] \leftarrow$ Register_shuffle($T5\_Reg[1]$)
                                          ▷ $T6\_Reg[1] = P^{-1} \cdot T5\_Reg[1]^\top$
30:    $T7\_Reg[\theta] \leftarrow$ Decomposition($T6\_Reg[1]$)
31:    $T8\_Reg[\theta] \leftarrow$ mma($Q_3'\_Reg[\theta], T7\_Reg[\theta]$)
32:    $A\_Reg[1] \leftarrow$ Composition($T8\_Reg[\theta^2]$)
33: end procedure

34: (NTT_TFMs, INTT_TFMs) $\leftarrow$ TFM_OFFLINE
35: $A'\_Reg[1] \leftarrow$ NTT($A[1]$)
36: $B'\_Reg[1] \leftarrow$ NTT($B[1]$)
37: $C'\_Reg[1] \leftarrow$ Element_Mul($A'\_Reg[1], B'\_Reg[1]$)
38: return $C[1] \leftarrow$ INTT($C'\_Reg[1]$)
```

A.2 Detailed Procedure of Memory-Less Polynomial Multiplication

Algorithm 1 gives a detailed procedure of memory-less polynomial multiplication (ML-PolyMul).

As described in Section 4.2, the modulus q of the algorithm to which ML-PolyMul is applied determines that the elements participating in the `wmma.mma` operation must be decomposed into θ parts. During the `wmma.mma` operation, each thread holds 8 registers (referred to as a register group) to store data for computation. Therefore, the entire `wmma.mma` operation process includes three steps: "**Decomposition-mma-Composition**".

In the implementation, the direct objects of operation are the register groups of each thread within a warp, collectively represented as $T1\_Reg[\theta]$. Here, the letter T denotes that the register group is temporary, and the number in brackets indicates the group size. The `wmma.mma` operation involves three types of register groups, which are explained as follows:

- $T\_Reg[1]$ is used to store elements that have not yet been decomposed.
- $T\_Reg[\theta]$ is used to hold elements after they have been decomposed into θ 8-bit parts.
- $T\_Reg[\theta^2]$ is used to contain the results of the **mma** operation between the decomposed polynomial coefficients and the decomposed TFMs.

Descriptions of the operations involved in the algorithm 1 are as follows:

- **TFM_offline**: This operation computes the TFMs in (I)NTT offline and serves as a hard-coded data source.
- **Decomposition**: This operation decomposes the data from one register group of each thread within a warp and stores it across θ register groups.
- **mma**: This operation computes the product of two register groups as shown in Figure 1, storing the result in a register group, essentially performing matrix multiplication between the matrices represented by the two register groups.
- **Composition**: This operation combines data from θ^2 register groups of each thread within a warp and stores it in a single register group through bit-shifting and addition.
- **Element_Mul**: This operation computes the element-wise product of two register groups within each thread of a warp, storing the result in a single register group, which essentially performs the Hadamard product between the matrices represented by the two register groups.
- **Register_shuffle**: For each thread in a warp, the registers in a register group are denoted as $(r_0, r_1, \ldots, r_7)$. The operation swaps the registers (r_2, r_3) and (r_4, r_5) within each thread's register group across the warp.