cuML-DSA: Optimized Signing Procedure and Server-Oriented GPU Design for ML-DSA

Shiyu Shen, Hao Yang, Wenqian Li, and Yunlei Zhao

***Abstract*—The threat posed by quantum computing has precipitated an urgent need for post-quantum cryptography. Recently, the post-quantum digital signature draft FIPS 204 has been published, delineating the details of the ML-DSA, which is derived from the CRYSTALS-Dilithium. Despite these advancements, server environments, especially those equipped with GPU devices necessitating high-throughput signing, remain entrenched in classical schemes. A conspicuous void exists in the realm of GPU implementation or server-specific designs for ML-DSA.**

In this paper, we propose the first server-oriented GPU design tailored for the ML-DSA signing procedure in high-throughput servers. We introduce several innovative theoretical optimizations to bolster performance, including depth-prior sparse ternary polynomial multiplication, the branch elimination method, and the rejection-prioritized checking order. Furthermore, exploiting server-oriented features, we propose a comprehensive GPU hardware design, augmented by a suite of GPU implementation optimizations to further amplify performance. Additionally, we present variants for sampling sparse polynomials, thereby streamlining our design. The deployment of our implementation on both server-grade and commercial GPUs demonstrates significant speedups, ranging from 284.9× to 485.3× against the CPU baseline, and an improvement of up to 60.9% compared to related work, affirming the effectiveness and efficiency of the proposed GPU architecture for ML-DSA signing procedure.

***Index Terms*—post-quantum cryptography, digital signature, ML-DSA, sparse polynomial multiplication, GPU acceleration.**

I. Introduction

DIGITAL signatures, serving as the bedrock for data integrity and authentication, have always been indispensable in the realm of data security. The essence lies in detecting unauthorized alterations to data and validating the identity of the signatory. However, the strides in quantum computing technology threaten the foundation of existing digital signature schemes primarily based on integer factorization and discrete logarithms [1]. Such schemes, while robust against classical computing attacks, crumble before quantum computers.

Recognizing the impending challenge, there has been an international endeavor to identify and standardize cryptographic algorithms resistant to quantum attacks. Spearheading this movement, the National Institute of Standards and Technology (NIST) initiated an extensive public vetting process in search of quantum-resistant public-key cryptographic algorithms [2]. This rigorous initiative saw a deluge of proposals, reflecting the global urgency and effort in thwarting quantum threats. After three rounds, NIST select one key encapsulation mechanism (KEM) and three digital signature algorithms for standardization in commerce, where the CRYSTALS-Dilithium emerged as the primary choice [3]. The recent unveiling of the draft standard FIPS 204 [4] epitomizes this endeavor, detailing the ML-DSA (Module Lattice Digital Signature Algorithm) – a derivative of the Dilithium [5], [6]. This standard promises strong unforgeability and is envisioned to provide long-term security in the impending quantum era.

In the commercial arena, throughput is not just desirable, but indispensable. Contemporary businesses grapple with an immense volume of online transactions, each utilizing digital signatures to guarantee message integrity and authenticity. This necessitates the implementation of high-throughput and real-time cryptographic solutions. Servers, being the backbone of such transactions, need designs emphasizing throughput. One of the key allies in this challenge is the GPU. With its inherent parallelism, GPUs offer concurrent processing capabilities, making them prime candidates for accelerating server-grade tasks. Recent literature showcases several optimizations of Dilithium, targeting diverse hardware like ASICs and FPGAs [7]–[9], and software platforms ranging from high-performance processors [10]–[13] to embedded devices [14], [15]. Moreover, while GPU implementations of post-quantum KEMs abound [16]–[21], there is a conspicuous dearth of GPU implementations for post-quantum digital signature schemes [11], [12], [22], and high-throughput designs remain largely in classical schemes. To date, there exists no GPU design of the ML-DSA and for Dilithium, only studies [11], [12] have been reported.

Contributions. In this work, we introduce several theoretical optimizations as well as the first server-centric GPU design for the ML-DSA signing procedure. Our aim is to enhance both signing performance and throughput. A summary of our contributions is as follows:

- *Optimization of the Rejection Process*. Utilizing the sparse ternary polynomial multiplication technique, we introduce an enhanced depth-prior version that facilitates earlier rejection, and we leverage both vertical and horizontal parallelism to bolster parallelism. Then, we present a method to eradicate branching, promoting constant-time execution which is more amenable to parallel operations. Additionally, we recommend a rejection-prioritized norm checking sequence for the initial three checks, allowing for more prompt identification of invalid signatures.
- *Server-Oriented Design*. We delve into the possible accelerations that a server-centric design might offer and put forward a comprehensive hardware architecture for the

S. Shen, W. Li, and Y. Zhao are with the Fudan University, Shanghai, China. H. Yang is with the Nanjing University of Aeronautics and Astronautics. Y. Zhao is the corresponding author. E-mail: ylzhao@fudan.edu.cn.

Manuscript received October, 2023.

ML-DSA signing process. Then, we introduce an optimized GPU acceleration engine, accompanied by several implementation enhancements. These include integration with early evaluation, refined memory access patterns, and the caching of key component, resulting a reduction in IO latency and an uptick in overall performance.

- *Performance Analysis*. We execute our implementation on both server-grade and commercial GPUs, assessing both batching and streaming methods. Compared to the CPU baseline, we achieve performance gains of 271.6× to 333.9× on the A100 and 348.1× to 485.3× on the 4090 GPU across the three parameter sets. In comparison to the AVX2 optimized implementation, our implementation is faster by factors ranging from 60.3× to 75.7× on the A100 and 73.6× to 110.0× on the 4090 GPU. Furthermore, we record improvements of up to 60.9% against recent GPU-based work on identical platforms.

II. Preliminaries

A. Notation

Let n be a power of two and q a prime where $q \equiv 1 \pmod{2n}$. Define $\mathbb{Z}$ as the integer set, and $R = \mathbb{Z}[X]/(X^n + 1)$ as the $2n$-th cyclotomic ring. Additionally, $\mathbb{Z}_q$ is defined as $\mathbb{Z}/q\mathbb{Z}$, while R_q is given by $R/qR \cong \mathbb{Z}_q[X]/(X^n + 1)$. We use integer intervals, e.g., $[0, n] = \{0, 1, ..., n\}$.

Elements in R (or R_q) are represented as polynomials, typically denoted by bold, italic lowercase letters (e.g., $\boldsymbol{f}$). Vectors are indicated using bold, upright lowercase letters (e.g., $\mathbf{x}$). Each polynomial $\boldsymbol{f} \in R$ or R_q is expressed as $\boldsymbol{f} = \sum i = 0^{n-1} f_i X^i$, where $f_i \in \mathbb{Z}$ or $\mathbb{Z}_q$ for $i \in [0, n)$. Matrices over R or R_q use uppercase boldface letters (e.g., $\mathbf{M}$). For a polynomial $\boldsymbol{f} \in R$, its ℓ_∞-norm is given by $||\boldsymbol{f}||_\infty = \max\{|f_i|\}$. Similarly, for a vector $\mathbf{x} = (\boldsymbol{x}_0, \boldsymbol{x}_1, ..., \boldsymbol{x}_{n-1}) \in R^n$, its ℓ_∞-norm is defined as $||\mathbf{x}||_\infty = \max\{||\boldsymbol{x}_i||_\infty\}$.

B. Polynomial Multiplication

Given polynomials $\boldsymbol{a} = \sum_{i=0}^{n-1} a_i X^i$, $\boldsymbol{c} = \sum_{i=0}^{n-1} c_i X^i \in R_q$, their multiplication over R_q yields $\boldsymbol{b} = \sum_{i=0}^{n-1} b_i X^i$. We discuss two common methods to compute this result.

Number-Theoretic Transform (NTT). The NTT is an efficient strategy for polynomial multiplications in R_q. Let ζ be the primitive $2n$-th root of unity. This method applies forward NTT to $\boldsymbol{a}$ and $\boldsymbol{c}$, performs point-wise multiplication in the NTT domain, and applies inverse NTT to the result. The process is formulated as:

$$\hat{\boldsymbol{a}} = \sum_{j=0}^{n-1} \hat{a}_j X^j := \mathsf{NTT}(\boldsymbol{a}), \hat{a}_j = \sum_{i=0}^{n-1} a_i \zeta^{(2i+1)j} \pmod{q}$$

$$\boldsymbol{a} = \sum_{i=0}^{n-1} a_i X^i := \mathsf{INTT}(\hat{\boldsymbol{a}}), a_i = \frac{1}{n} \sum_{j=0}^{n-1} \hat{a}_j \zeta^{-(2i+1)j} \pmod{q}$$

Then, performing $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{c} = \mathsf{INTT}(\mathsf{NTT}(\boldsymbol{a}) \cdot \mathsf{NTT}(\boldsymbol{c}))$ yields the result. This reduces computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, thus improving the performance.

Sparse Ternary Polynomial Multiplication (STPM). For polynomial $\boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{c} = \sum_{k=0}^{n-1} b_k X^k$, the schoolbook method computes each coefficient of $\boldsymbol{b}$ as:

Algorithm 1 Sparse ternary polynomial multiplication

Input: $\boldsymbol{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$, $\boldsymbol{a} = \sum_{i=0}^{n-1} a_i \cdot x^i \in R_q$
Output: $\boldsymbol{u} = \boldsymbol{c} \cdot \boldsymbol{a} \in R_q$

```
for i ∈ [0, n) do
    w_i := 0, v_i := a_i, v_{i−n} := −a_i
for i ∈ [0, n) do
    if c_i = 1 then
        for j ∈ [0, n) do
            w_j := w_j + v_{j−i}
    if c_i = −1 then
        for j ∈ [0, n) do
            w_j := w_j − v_{j−i}
for i ∈ [0, n) do
    u_i := w_i (mod q)
return u = Σ_{i=0}^{n−1} u_i · x^i
```

Algorithm 2 ML-DSA.Sign

Input: $sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, $M \in \{0,1\}^*$
- $\rho, K, tr \in \{0,1\}^{256}$
- $\mathbf{s}_1 := [\boldsymbol{s}_1^{(0)}, \ldots, \boldsymbol{s}_1^{(\ell-1)}] \in R_q^\ell$, $\mathbf{s}_2 := [\boldsymbol{s}_2^{(0)}, \ldots, \boldsymbol{s}_2^{(k-1)}] \in R_q^k$
- $\mathbf{t}_0 := [\boldsymbol{t}_0^{(0)}, \ldots, \boldsymbol{t}_0^{(k-1)}] \in R_q^k$

Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

```
μ ∈ {0,1}^512 := H(tr‖M)
rnd := {0}^256                                   ▷ Deterministic variant
ρ' ∈ {0,1}^512 := H(K‖rnd‖μ)
Â ∈ R_q^{k×ℓ} := ExpandA(ρ)
κ := 0; (z, h) :=⊥
while (z, h) =⊥ do
    y ∈ S_{γ1}^ℓ := ExpandMask(ρ', κ)
    w := Â · y
    w_1 := HighBits_q(w, 2γ_2)
    c̃ := (c̃_1, c̃_2) ∈ {0,1}^256 × {0,1}^{2λ−256} := H(μ‖w_1)
    c ∈ B_τ := SampleInBall(c̃_1)
    z := y + c · s_1
    r_0 := LowBits_q(w − c · s_2, 2γ_2)
    if ‖z‖_∞ ≥ γ_1 − β or ‖r_0‖_∞ ≥ γ_2 − β then
        (z, h) :=⊥
    else
        h := MakeHint_q(−c · t_0, w − c · s_2 + c · t_0, 2γ_2)
        if ‖c · t_0‖_∞ ≥ γ_2 or number of 1's in h exceeds ω then
            (z, h) :=⊥
    κ := κ + ℓ
```

$$b_k = \sum_{i=0}^{k} a_i c_{k-i} - \sum_{i=k+1}^{n-1} a_i c_{k+n-i} \pmod{q}$$

The original computational complexity of this method is $\mathcal{O}(n^2)$. However, if $\boldsymbol{c}$ is a sparse ternary polynomial with τ non-zero coefficients (± 1), we can optimize by evaluating $\boldsymbol{c}$'s coefficients and adding or subtracting $\boldsymbol{a}$'s coefficients. Algorithm 1 details this process. With $\boldsymbol{c}$ having only τ non-zero coefficients, multiplication complexity is reduced.

C. ML-DSA and Parallel STPM

ML-DSA is a digital signature scheme proven strongly unforgeable in the QROM based on decisional Module-LWE and SelfTargetMSIS assumptions [4]. It emerges from the CRYSTALS-Dilithium, a proposal submitted to the NIST PQC standardization project. Algorithm 2 summarizes the signing procedure, with parameter sets in Table I. This function uses the secret key sk and message M as input and generates a valid signature $\sigma := (\tilde{c}, \mathbf{z}, \mathbf{h})$ after several rounds of checks. The specifics of sub-procedures are available in [4]–[6].

TABLE I
ML-DSA PARAMETER SETS.

Parameter Set	n	q	d	τ	γ_1	γ_2	(k,ℓ)	η	β	ω	λ
ML-DSA-44	256	8380417	13	39	2^{17}	95232	(4,4)	2	78	80	128
ML-DSA-65	256	8380417	13	49	2^{19}	261888	(6,5)	4	196	55	192
ML-DSA-87	256	8380417	13	60	2^{19}	261888	(8,7)	2	120	75	256

Algorithm 3 Parallel sparse ternary polynomial multiplication

Input: $(\boldsymbol{c}, \mathbf{a})$, where $\mathbf{a} = [\boldsymbol{a}^{(0)}, \cdots, \boldsymbol{a}^{(r-1)}]^T \in R_q^r$, every $\boldsymbol{a}^{(j)} = \sum_{i=0}^{n-1} a_i^{(j)} \cdot x^i \in R_q$, and $\boldsymbol{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$
Output: $\mathbf{u} = \boldsymbol{c} \cdot \mathbf{a} = [\boldsymbol{u}^{(0)}, \cdots, \boldsymbol{u}^{(r-1)}]^T \in R_q^r$, where $\boldsymbol{u}^{(j)} = \boldsymbol{c} \cdot \boldsymbol{a}^{(j)} = \sum_{i=0}^{n-1} u_i^{(j)} X^i \in R_q$

for $i \in \{0, 1, \cdots, n-1\}$ **do**
  $w_i := 0$, $v_i := 0$, $v_{i-n} := 0$
  for $j \in \{0, 1, \cdots, r-1\}$ **do**
    $v_i := v_i \cdot M + \left(U + a_i^{(j)}\right)$
    $v_{i-n} := v_{i-n} \cdot M + \left(U - a_i^{(j)}\right)$
$\gamma := 2U \cdot \frac{M^r - 1}{M - 1}$
for $i \in [0, n)$ **do**
  if $c_i = 1$ **then**
    for $j \in [0, n)$ **do**
      $w_j := w_j + v_{j-i}$
  if $c_i = -1$ **then**
    for $j \in [0, n)$ **do**
      $w_j := w_j + (\gamma - v_{j-i})$
for $i \in \{0, 1, \cdots, n-1\}$ **do**
  $t := w_i$
  for $j \in \{0, 1, \cdots, r-1\}$ **do**
    $u_i^{(r-1-j)} := (t \bmod M) - \tau U \pmod q$
    $t := \lfloor t / M \rfloor$
return $\mathbf{u} = [\boldsymbol{u}^{(0)}, \cdots, \boldsymbol{u}^{(r-1)}]^T$

A distinctive feature of the ML-DSA signing procedure is the sparse ternary nature of $\boldsymbol{c}$, and vectors $\mathbf{s}_1$, $\mathbf{s}_2$, and $\mathbf{t}_0$ composed of small-norm polynomials. Specifically, for $i \in [0, n)$ and $j \in [0, l)$ for $\mathbf{s}_1$ or $[0, k)$ for $\mathbf{s}_2$, we have:

- $\boldsymbol{c}$ is a ternary polynomial where c_i ranges within $\{-1, 0, 1\}$, with only τ non-zero coefficients.
- The polynomial coefficients in $\mathbf{s}_1$ and $\mathbf{s}_2$ are confined to $[-\eta, \eta]$, denoting that $s_{1,i}^{(j)}$, $s_{2,i}^{(j)} \in [-\eta, \eta]$.
- The polynomial coefficients in $\mathbf{t}_0$ are restricted to $[-2^{d-1} + 1, 2^{d-1}]$, a subset of $[-2^{d-1}, 2^{d-1}]$.

These properties make conventional NTT approach inefficient for computing $\boldsymbol{c} \cdot \mathbf{s}_1$, $\boldsymbol{c} \cdot \mathbf{s}_2$, and $\boldsymbol{c} \cdot \mathbf{t}_0$, as it transforms small integers in $[-\eta, \eta]$ and $[-2^{d-1}, 2^{d-1}]$ to $\mathbb{Z}_q$, where η and $2^{d-1} \ll q$, increasing memory usage.

To optimize based on the property, a parallel STPM method has been introduced as an alternative to NTT for Dilithium in [13]. This technique calculates the polynomial multiplications concurrently as detailed in Algorithm 3. The process packs vector $\mathbf{a}$ into array $\boldsymbol{v} := \{v_i\}$, then stores parallel computation results in array $\boldsymbol{w} := \{w_j\}$, which is derived by summing elements in $\boldsymbol{v}$ according to c_i. Each polynomial multiplication result is extracted by decomposing $\boldsymbol{w}$. Here, U denotes the upper limit of a_i, M represents multiplicative additions boundary, and γ assists in decomposition. The vector $\mathbf{a}$ can be substituted by $\mathbf{s_1}$, $\mathbf{s_2}$, or $\mathbf{t_0}$ to determine multiplication outcomes with $\boldsymbol{c}$.

D. Target Platform

Our focus is high-performance GPU platforms, leveraging their parallel capabilities to enhance signing procedure speed. While CPUs excel at managing the logical flow of general-purpose programs, GPUs are specially designed for tasks requiring intense parallel processing. This design renders GPUs exceptionally effective for computationally demanding tasks, offloading much of the burden traditionally borne by CPUs.

In this architecture, threads execute instructions in streams and can be organized into blocks. Functions that operate on the GPU are termed kernels. A Streaming Multiprocessor (SM) is the primary unit responsible for executing a thread block of a kernel. During execution, blocks are partitioned into warps for single-instruction-multiple-thread execution. Each warp comprises 32 threads with sequential indices.

The GPU memory hierarchy is structured to facilitate rapid data access. Closest to the CUDA cores and also part of each SM are the register file, L1 cache, shared memory, and constant caches, with registers having the fastest access. Beyond these are larger, higher-latency regions shared across all SMs, such as L2 cache, global memory, local memory, texture, and constant memory. Among them, only the register, shared, and global memory support read-write operations, while constant memory is cached in constant caches.

III. Theoretical Optimizations to ML-DSA Signing Procedure

This section details theoretical improvements to the ML-DSA signing procedure. We present our depth-prior optimization, branch elimination strategies, and a refined norm check order, all aimed at enhancing signing efficiency.

A. Signing Architecture Overview

To provide a clear understanding, we summarize the arithmetic involved. We decompose and reconstruct operations intrinsic to the ML-DSA signing procedure, considering associativity. Fig. 1 presents a detailed graphical representation.

1) CRH*:* This function refers to a collision resistant hash function that maps to $\{0,1\}^{512}$ and is instantiated through SHAKE-256. During signing, it is invoked twice with different inputs, i.e., the concatenated $tr||M$ and $K||rnd||\mu$, to produces the first 64 bytes for μ and ρ', respectively.

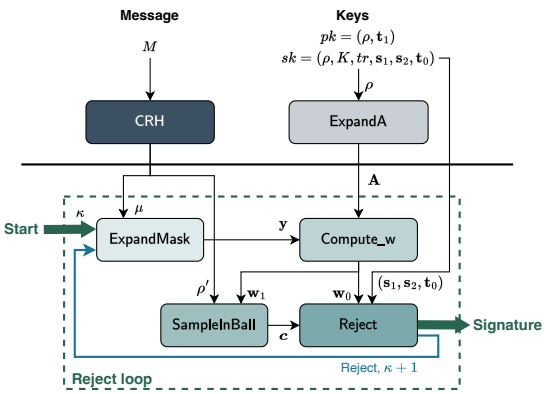


Fig. 1. Structure of the ML-DSA signing procedure.

2) ExpandA*:* This function adopts rejection sampling to sample uniform polynomials and obtain matrix $\mathbf{A} \in R_q^{k\times\ell}$. The random streams are generated from ρ via SHAKE-128. In this process, 3-byte sequences are sequentially extracted and compared against q, of which that numerically less than q are retained as coefficients. The derived matrix is interpreted in the NTT domain to enable fast polynomial multiplication.

3) ExpandMask*:* This function employs a principle similar to ExpandA but uses SHAKE-256 to generate a random polynomial vector $\mathbf{y} \in S_{\gamma_1}^{\ell}$. As γ_1 is a power of 2, rejection does not occur and coefficients are sampled sequentially. The input is the concatenation of ρ' and κ that introduces randomness to each round. Outputs are generated by taking $2\gamma_1$ bits as a positive integer and subtracting $\gamma_1 - 1$ from each result.

4) Compute_w*:* This process computes the inner-product of matrix $\mathbf{A}$ and vector $\mathbf{y}$ to derive $\mathbf{w} \in R_q^k$, then decomposes it into high-order and low-order vectors $\mathbf{w}_1$ and $\mathbf{w}_0$. Moreover, a serialization process is invoked on $\mathbf{w}_1$ in preparation of the subsequent hashing stage.

5) SampleInBall*:* This process uses SHAKE-256 to absorb $\mu||\mathbf{w}_1$, yielding $\tilde{c}$. The result is then re-input to SHAKE-256 to generate a random stream for $\boldsymbol{c}$ with τ nonzero ±1. The first τ bits serve as sign determinants, and τ distinct positions in $[0, n)$ for nonzero values are generate by rejection sampling.

6) Reject*:* This process involves sk-related computations and checks the resultant value to ensure correctness and avoid secret information disclosure. It computes $\mathbf{z} := \mathbf{y}+\boldsymbol{c}\cdot\mathbf{s}_1$, $\mathbf{r}_0 := \mathsf{LowBits}_q(\mathbf{w}-\boldsymbol{c}\cdot\mathbf{s}_2)$, and $\boldsymbol{c}\cdot\mathbf{t}_0$. Then it examines if conditions $||\mathbf{z}||_\infty < \gamma_1 - \beta$, $||\mathbf{r}_0||_\infty < \gamma_2 - \beta$, and $||\boldsymbol{c}\cdot\mathbf{t}_0||_\infty < \gamma_2$ are satisfied. Meanwhile, the output $\mathbf{h}$ of MakeHint must meet a hamming weight bound. Any deviation from these conditions results in process abortion and a new iteration begins.

Potential for Acceleration. Despite extensive research, further acceleration of the ML-DSA scheme remains possible, particularly in specific scenarios. We discuss potential acceleration strategies below.

1) Server-oriented design: In scenarios where a server operates as the subject, two primary strategies emerge to optimize computational efficiency for high-volume signing requests. Firstly, given the server's consistent key, certain key-related operations can be precomputed offline. Secondly, several operations are common across signing tasks and, thus, need not be redundantly executed.

2) Different multiplication techniques: During the Reject procedure, one can capitalize on the characteristic that $\boldsymbol{c}$ is a sparse ternary polynomial. A general NTT approach might overlook this property, leading to inefficiencies. Leveraging specialized multiplication techniques tailored for sparse polynomials can reduce computational complexity and eliminate the need for domain conversions.

3) Earlier rejection: In original ML-DSA, a comprehensive polynomial evaluation result must be computed before norm checks begin. By refining the computational pattern, certain coefficients can be determined early, enabling checkpoint inspection and facilitating swifter rejection. Moreover, the conditions to be checked vary in their likelihood of rejection. Prioritizing the assessment of conditions with higher rejection probabilities can further mitigate unnecessary computations.

Algorithm 4 Depth-prior sparse ternary polynomial multiplication

Input: $\boldsymbol{c} = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$, $\boldsymbol{a} = \sum_{i=0}^{n-1} a_i \cdot x^i \in R_q$
Output: $\boldsymbol{u} = \boldsymbol{c} \cdot \boldsymbol{a} \in R_q$
1: **for** $i \in [0, n)$ **do**
2: $\quad w_i := 0$, $v_i := a_i$, $v_{i-n} := -a_i$
3: **for** $j \in [0, n)$ **do** ▷ Traverse polynomial $\boldsymbol{a}$
4: $\quad$ **for** $i \in [0, n)$ **do** ▷ Traverse sparse polynomial $\boldsymbol{c}$
5: $\quad\quad$ **if** $c_i = 1$ **then**
6: $\quad\quad\quad w_j := w_j + v_{j-i}$
7: $\quad\quad$ **if** $c_i = -1$ **then**
8: $\quad\quad\quad w_j := w_j - v_{j-i}$ ▷ w_j can be obtained after jth loop
9: **for** $i \in [0, n)$ **do**
10: $\quad u_i := w_i \pmod q$
11: **return** $u = \sum_{i=0}^{n-1} u_i \cdot x^i$

In the following, we start from these aspects and propose our optimizations to accelerate the signing of the ML-DSA.

B. Depth-Prior Sparse Polynomial Multiplication

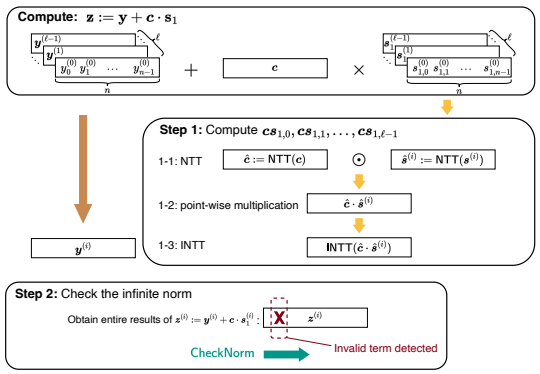
The ML-DSA scheme often encounters computational waste when the norms of the resulting values exceed the set bounds. Conventional methods like NTT-based [4], [6] and PSPM-based [13] all face this inefficiency. The applied width-prior approach requires a full polynomial arithmetic operation evaluation before any bounds check can be performed. Motivated by the need to obtain checkable coefficient values more rapidly, we explore a depth-first strategy. This strategy aims to curtail wasteful computations, which is particularly beneficial when the coefficients that exceed the limits have small indices.

Building upon the sparse ternary polynomial multiplication detailed in Algorithm 1, we introduce a depth-prior method depicted in Algorithm 4. The original methodology computes the accumulation on w_j based on c_i values, requiring all w_j values to be determined. In contrast, our approach commences from the w_j index. For each w_j, we traverse polynomial $\boldsymbol{c}$, accumulating to w_j according to each c_i value. This allows some w_j values to be obtained faster, enabling earlier checks and minimizing potential unnecessary operations. Based on this, we further devise vertical and horizontal parallelism, which improve the entire parallelism of the process to facilitate parallel execution. We give a visualization of our approach and the comparison with NTT-based method in Fig. 2.

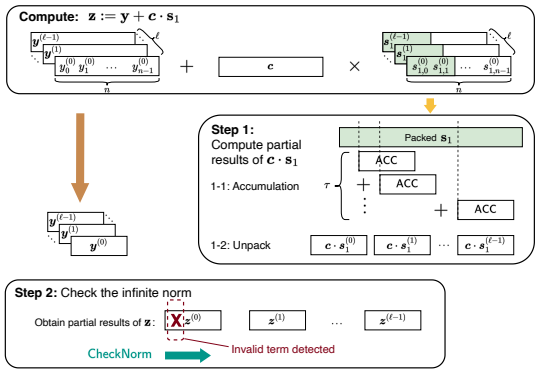
Vertical Parallelism. Given the bounded coefficient values of the input polynomial $\boldsymbol{a}$ and a constant τ, each w_j possesses an upper limit linearly related to the coefficient bounds of $\boldsymbol{a}$. If the infinity norm is relatively compact, especially compared to machine word size, we can employ packing techniques. This enables bundling multiple polynomials into a single machine word unit, facilitating single-instruction-multiple-data (SIMD) computations. Consequently, this approach allows operations on polynomial vectors to be performed simultaneously, introducing a vertical parallelism approach.

For a given $\mathbf{a} = [\boldsymbol{a}^{(0)}, \cdots, \boldsymbol{a}^{(r-1)}]^T \in R_q^r$, where $||\boldsymbol{a}^{(j)}||_\infty = U \ll q$, $j \in [0, r)$, we utilize a similar construct as in Algorithm 3 to define:

$$v_i^{(j)} = \begin{cases} U + a_i^{(j)}, & i \in [0, n) \\ U - a_{n+i}^{(j)}, & i \in (-n, 0) \end{cases}, j \in [0, r)$$



(a) NTT



(b) DPSTPM

Fig. 2. Comparisons between the (a) NTT and (b) our DPSTPM approach. In (a), the entire result of $\mathbf{z}$ must be obtained before checking the infinite norm. In (b), the depth-prior approach allows partial results to be checked first, potentially reducing computation when current values are invalid.

Here, $v_i^{(j)} \in [0, 2U]$, implying all non-negative values. Consequently, the value of $w_i^{(j)}$ is constrained to the limit of $2\tau U$. Let M be a power-of-two that satisfies $M > 2\tau U$, ensuring that $2\tau U < M \ll q$. Given this, we can define:

$$v_i = \begin{cases} (U + a_i^{(r-1)}) \cdot M^{r-1} + \ldots + U + a_i^{(0)}, & i \in [0, n) \\ (U - a_{n+i}^{(r-1)}) \cdot M^{r-1} + \ldots + U - a_{n+i}^{(0)}, & i \in (-n, 0) \end{cases}$$

Typically, we set $M = 2^{\lceil \log_2(1+2\tau U) \rceil}$ to simplify implementation through bit-shifting. We discern the upper-bound for w_j as $\gamma = 2U \cdot \frac{M^r - 1}{M - 1}$. Within ML-DSA, w_j conforms to either 32-bit or 64-bit dimensions, allowing streamlined software representation. After accumulation, we subtract the extraneous τU, with the ultimate result procured via bit-shifting and unpacking. This methodology achieves vertical parallel computation by packing r polynomial coefficients.

Horizontal Parallelism. Since the computation of each v_i is independent, we introduce horizontal parallelism beyond vertical parallelism, enabling simultaneous computation of multiple v_i values. This parallelism resembles NTT, where distinct butterfly operations can be executed concurrently. With the computational capabilities of modern systems, such parallelism is readily achievable. For instance, on a GPU, multiple threads can be launched, with each thread handling the computation for a single v_i or a set thereof.

Algorithm 5 Depth-prior parallel sparse ternary polynomial multiplication

Input: $(c, \mathbf{a})$, where $\mathbf{a} = [a^{(0)}, \cdots, a^{(r-1)}]^T \in R_q^r$, every $a^{(j)} = \sum_{i=0}^{n-1} a_i^{(j)} \cdot x^i \in R_q$, and $c = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$
Output: $\mathbf{u} = c \cdot \mathbf{a} = [u^{(0)}, \cdots, u^{(r-1)}]^T \in R_q^r$, where $u^{(j)} = c \cdot a^{(j)} = \sum_{i=0}^{n-1} u_i^{(j)} \cdot x^i \in R_q$

for $i \in \{0, 1, \cdots, n-1\}$ **do**
  $w_i := 0$, $v_i := 0$, $v_{i-n} := 0$
  for $j \in \{0, 1, \cdots, r-1\}$ **do** ▷ Pack r coefficients of $\mathbf{a}$ in v
    $v_i := v_i \cdot M + \left(U + a_i^{(j)}\right)$
    $v_{i-n} := v_{i-n} \cdot M + \left(U - a_i^{(j)}\right)$
$\gamma := 2U \cdot \frac{M^r - 1}{M - 1}$
for $j \in \{0, 1, \cdots, n-1\}$ **do** ▷ Depth-prior evaluation
  for $i \in \{0, 1, \cdots, n-1\}$ **do**
    if $c_i = 1$ **then** $w_j := w_j + v_{j-i}$
    if $c_i = -1$ **then** $w_j := w_j + (\gamma - v_{j-i})$
  $t := w_j$
  for $i \in (0, 1, \cdots, r-1)$ **do** ▷ Unpack to get results
    $u_j^{(r-1-i)} := (t \bmod M) - \tau U (\bmod q)$
    $t := \lfloor t / M \rfloor$
return $\mathbf{u} = [u^{(0)}, \cdots, u^{(r-1)}]^T$

Algorithm 6 Unified depth-prior parallel sparse ternary polynomial multiplication

Input: $(c, \mathbf{a})$, where $\mathbf{a} = [a^{(0)}, \cdots, a^{(r-1)}]^T \in R_q^r$, every $a^{(j)} = \sum_{i=0}^{n-1} a_i^{(j)} \cdot x^i \in R_q$, and $c = \sum_{i=0}^{n-1} c_i \cdot x^i \in B_\tau$
Output: $\mathbf{u} = c \cdot \mathbf{a} = [u^{(0)}, \cdots, u^{(r-1)}]^T \in R_q^r$, where $u^{(j)} = c \cdot a^{(j)} = \sum_{i=0}^{n-1} u_i^{(j)} \cdot x^i \in R_q$

$\boldsymbol{cp} := \{cp_k\}_{k \in [0, \tau)}$, $j := 0$ ▷ Position array generation phase
for $i \in [0, n)$ **do**
  if $c_i = 1$ **then**
    $cp_j := -i$, $j = j + 1$
  if $c_i = -1$ **then**
    $cp_j := n - i$, $j = j + 1$
for $i \in [0, n)$ **do** ▷ Vector packing phase
  $w_i := 0$, $v_i := 0$, $v_{i-n} := 0$
  for $j \in (0, 1, \cdots, r-1)$ **do**
    $v_i := v_i \cdot M + \left(U + a_i^{(j)}\right)$
    $v_{i-n} := v_{i-n} \cdot M + \left(U - a_i^{(j)}\right)$
    $v_{i+n} := v_{i+n} \cdot M + \left(U - a_i^{(j)}\right)$
for $j \in [0, n)$ **do** ▷ Evaluation phase
  for $i \in [0, \tau)$ **do**
    $w_j := w_j + v_{j + cp_i}$
  $t := w_j$
  for $i \in [0, r)$ **do** ▷ Results unpacking phase
    $u_j^{(r-1-i)} := (t \bmod M) - \tau U (\bmod q)$
    $t := \lfloor t / M \rfloor$
return $\mathbf{u} = [u^{(0)}, \cdots, u^{(r-1)}]^T$

C. Methods for Branch Elimination

In the algorithm described, a remaining concern is the inclusion of branching statements. Notably, while these do not raise side-channel security issues, their presence considerably hinders the potential for code optimization. Branching statements inhibit several compilation-phase optimizations, including loop unrolling and constant folding. Additionally, the unpredictability introduced by these statements disrupts the optimal alignment of pipeline scheduling due to instruction indeterminacy. Further complications arise in multi-threaded parallel systems, where differing execution paths can culminate in challenges like thread divergence. These factors

collectively degrade performance and impede optimal execution, highlighting the need to devise strategies that effectively minimize branching instructions.

To eliminate branching statements within the algorithm, it is pivotal to first elucidate the foundational reasons for their incorporation. The execution logic can be categorized into two predominant pathways, dependent on the values of $\boldsymbol{c}$:

– if $c_i = 1$, $w_j := w_j + v_{j-i}$;
– if $c_i = -1$, $w_j := w_j + (\gamma - v_{j-i})$.

Here, $\gamma := 2U \cdot \frac{M^r-1}{M-1}$. Thus the primary impetus for these branches arises from the heterogeneity in the addends.

To address the complexity introduced by these diverse addends, our preliminary strategy seeks to standardize them. By defining $\bar{v}_{j-i} = \gamma - v_{j-i}$, $j \in [0, r)$, we obtain:

$$\bar{v}_i^{(j)} = \begin{cases} (U - a_i^{(r-1)}) \cdot M^{r-1} + \cdots + U - a_i^{(0)}, & i \in [0, n) \\ (U + a_{n+i}^{(r-1)}) \cdot M^{r-1} + \cdots + U + a_{n+i}^{(0)}, & i \in (-n, 0) \end{cases}$$

By refining the range of i within this equation, we can incorporate it into the original expression of $v_i^{(j)}$. Consequently,

$$v_i^{(j)} = \begin{cases} (U + a_{n+i}^{(r-1)}) \cdot M^{r-1} + \cdots + U + a_{n+i}^{(0)}, & i \in [n, 2n) \\ (U - a_i^{(r-1)}) \cdot M^{r-1} + \cdots + U - a_i^{(0)}, & i \in [0, n) \\ (U + a_{n+i}^{(r-1)}) \cdot M^{r-1} + \cdots + U + a_{n+i}^{(0)}, & i \in (-n, 0) \end{cases}$$

Therefore, the previous branching can be reformulated as:

– if $c_i = 1$, $w_j := w_j + v_{j-i}$;
– if $c_i = -1$, $w_j := w_j + v_{n+j-i}$.

One residual challenge lies in the inconsistent index of the addends. To address this, we propose altering the structure of $\boldsymbol{c}$. Instead of preserving the complete $\boldsymbol{c}$ during sampling, we only conserve the respective τ positions, thereby mitigating the index variations. Let $\boldsymbol{cp}$ represent the positions array, thus $cp_j := -i$ if $c_i = 1$ and $cp_j := n - i$ if $c_i = -1$. This configuration unifies the accumulation phase expression as $w_j := w_j + v_{j+cp_i}$, eliminating the need for branching. A detailed representation of this refined approach is delineated in Algorithm 6.

D. Rejection-Prioritized Norm Checking Order

In ML-DSA's signing procedure, four distinct conditions must be met for a valid signature, as shown in Algorithm 2. The initial two conditions stipulate that $||\mathbf{z}||_\infty < \gamma_1 - \beta$ and $||\mathbf{r}_0||_\infty < \gamma_2 - \beta$. The likelihood of all coefficients meeting these bounds are $e^{-256 \cdot \beta l/\gamma_1}$ and $e^{-256 \cdot \beta k/\gamma_2}$, respectively. The cumulative probability of both conditions being met is $e^{-256 \cdot \beta(l/\gamma_1 + k/\gamma_2)}$. Based on the parameter values in Table I, these two conditions primarily dictate rejection process restarts, rather than the conditions encompassing $||\boldsymbol{c} \cdot \mathbf{t}_0||_\infty$ and hints. Furthermore, as k/γ_2 typically surpasses l/γ_1, $\mathbf{r}_0$ is more susceptible to rejection than $\mathbf{z}$. Drawing from this analysis, we advocate for a rejection-prioritized norm checking sequence. In the DPSTPM computation, we commence with $\boldsymbol{c} \cdot \mathbf{s}_2$, which is attributed with the highest rejection probability, followed sequentially by $\boldsymbol{c} \cdot \mathbf{s}_1$ and $\boldsymbol{c} \cdot \mathbf{t}_0$. Implementing this refined norm checking sequence enables early checking of conditions with high rejection probabilities, effectively curtailing redundant execution procedures.

E. Discussions of Proposed Methods

Below, we discuss some features of our techniques, including the advantages, novelty, correctness, and security.

Advantage of Depth-Prior PSTPM. The principal merit of our method lies in its rapid rejection capability via the depth-prior computational pattern, significantly reducing redundant calculations during the rejection phase. Besides this, our technique offers additional benefits:

- *Multi-dimensional parallelism*: Unlike the singularly horizontal parallelism of NTT, our DPSTPM introduces an additional vertical parallelism dimension. This facilitates simultaneous operations on diverse polynomials even within standard machine word boundaries.
- *Flexibility*: DPSTPM's parallelism boasts inherent adaptability due to its minimal constraints on parallelism degree. Conversely, NTT entails stringent requirements for parameter selection and parallelism degree. Our approach permits an arbitrary number of v_i computations at once, each functioning independently.
- *Lightweight and constant-time implementation*: Multiplication operations can be burdensome and potentially variable in execution time on lightweight platforms, introducing a risk of side-channel attacks. Our DPSTPM eliminates multiplications, relying solely on lightweight addition and bit-shifting. Additionally, branching only pertain to the public $\boldsymbol{c}$, avoiding energy analysis concerns.

Novelty. While Dai et al. [23] also applied STPM on GPUs to accelerate NTRU signatures and proposed storing non-zero coefficient positions, our approach have several differences. First, their method uses fixed-size tables due to constant numbers of 1 and -1 coefficients. Second, the different ring setting makes the detail computation different in the accumulation. Additionally, we devise novelty techniques, including the multi-dimensional parallelism and depth-prior computational pattern. Unlike [23], which lacks coefficient packing, our method enables SIMD computation for vertical parallelism. Our ML-DSA-tailored depth-prior computation enhances the rejection process, improving overall performance.

Effects on Original ML-DSA design. Our approach only accelerates the detection of invalid signatures without affecting the original design. The checks of $||\mathbf{z}||_\infty < \gamma_1 - \beta$ and $||\mathbf{r}_0||_\infty < \gamma_2 - \beta$ are independent. Valid signatures must satisfy both conditions, while if either is not satisfied, the current value will be rejected and the loop restarted. Therefore, our modification does not affect correctness. Additionally, this modification preserves the nature of the MLWE- and SelfTargetMSIS-based problems. For Unforgeability under Chosen Message Attacks (UF-CMA) security, the adversary must output a valid pair $(M, (\mathbf{z}, \mathbf{h}, \boldsymbol{c}))$ satisfying $||\mathbf{z}||_\infty < \gamma_1 - \beta$, $\mathcal{H}\left(\mu \| \mathsf{UseHint}_q\left(\mathbf{h}, \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d, 2\gamma_2\right)\right) = c$, and the number of coefficients in $\mathbf{h}$ equaling 1 being less than ω. All these can be satisfied since our method does not change any condition checks. Similarly, the strong UF-CMA can also be proved. For implementation, we apply constant-time techniques to prevent side-channel leakage. Meanwhile, our implementation provides the same test vectors, proving adherence to the original ML-DSA design.

IV. Server-Oriented GPU Accelerator Design

In this section, we introduce cuML-DSA, a tailored GPU accelerator optimized for server-centric environments. We explore the potential benefits under this scenario, and outline a dedicated GPU architecture for the ML-DSA signing and describe several optimization techniques. This encompasses integrating DPSTPM with early evaluation, optimizing memory access, and strategic component caching, collectively culminating in minimized IO latency and heightened performance.

A. Design Overview

Our implementation focuses the proposed optimized signing of ML-DSA. Unlike previous works, which used 32 threads [12] or alternated between 32 and 128 threads [11], we allocate a block of 128 threads to each task. This offers significant advantages, including optimal memory access and 100% SM occupancy. In contrast, the 32-thread configuration achieves only 33.3% occupancy. Notably, lower occupancy doesn't always reduce performance. It allows greater use of registers and shared memory, potentially reducing overall IO latency. However, our scenario benefits from the 100% occupancy configuration, which provides sufficient memory resources without sacrificing data access speed. Moreover, [11] shows that several operations are more efficient with 128 threads compared to 32. Based on these, we adhere to the design in Sec III-A and implement corresponding kernels. We also introduce a Pack kernel to preprocess sk components. Given our server-oriented focus, only m-related kernels remain online, while ExpandA and Pack kernels are relegated to offline operations.

To efficiently manage multiple signing tasks, we implement batch processing within our kernels for simultaneous task execution. We also adopt the memory pool design from [11] to ensure secure and efficient memory access. Furthermore, we utilize the dynamic scheduling mechanism described in [11] to optimize hardware resource allocation. This approach addresses potential occupancy declines due to varying repetitions across different signing tasks. Consequently, we maintain consistently high occupancy and maximizes hardware utilization.

B. Implementation Details

In pursuit of a coherent and streamlined design, we deconstruct the six stages in ML-DSA signing to extract and identify the underlying arithmetic structures. Table II presents a detailed breakdown of each resultant step and its encompassing functions. From this analytical deconstruction, we derive a comprehensive structural design, as illustrated in Fig. 3.

We delineate our GPU implementation of the inline functions below, with pseudocode detailed in Algorithm 7. The inline functions can be stratified into two categories. The first comprises functions that operate singularly on one operand, without requiring data or thread interaction. Every thread processes a single element, invoking the functions to derive the outcomes. The functions within this category include:

1) ModRed*:* We use a variant of Barrett Reduction [6], [24] for modular reduction, replacing division with efficient multiplication and bit-shifting. Unlike the original [24], we compute t with only addition and bit-shifting, leading to stricter input/output bounds but more effective strategy.

TABLE II
Structured decomposition of the implemented GPU kernels and the constituent inline functions.

Kernels	Composing inline functions
CRH	SHAKE
ExpandMask	SHAKE, Serialization
Compute_w	NTT, MontMul, ModRed, Decompose, Serialization
SampleInBall	SHAKE, Sampling
Reject	DPSTPM, CheckNorm, MakeHint, Serialization

Algorithm 7 Implementation details of inline functions

function MontMul($x \in [-\frac{\nu}{2}, \frac{\nu}{2}], \zeta \in (-q, q)$)
  `.reg .s32` a_h, a_l ▷ $a := a_h \cdot \nu + a_l$
  `mul.hi.s32` a_h, x, ζ
  `mul.lo.s32` a_l, x, ζ ▷ $a \leftarrow x \cdot \zeta$
  `mul.lo.s32` t, a_l, p ▷ $t \leftarrow [a \cdot p]_\nu$
  `mul.hi.s32` t, t, q ▷ $t \leftarrow t \cdot q/\nu$
  `sub.s32` t, a_h, t ▷ $a \leftarrow a_h - t$
function ModRed($a \leq 2^{31} - 2^{22} - 1$)
  $t := (a + (1 \ll 22)) \gg 23$
  return $t := a - t \cdot q$ ▷ $-6283009 \leq t \leq 6283007$
function Decompose(a)
  $a_1 = (a + 127) \gg 7$
  # IF $\gamma_2 == (q-1)/32$
    $a_1 = (1025 \cdot a_1 + (1 \ll 21)) \gg 22$, $a_1 = a_1 \& 15$
  # ELIF $\gamma_2 == (q-1)/88$
    $a_1 = (11275 \cdot a_1 + (1 \ll 23)) \gg 24$
    $a_1 = a_1 \wedge (((43 - a1) \gg 31) \& a_1)$
  $a_0 = a - 2\gamma_2 \cdot a_1$
  $a_0 = a_0 - (((q-1)/2 - a_0) \gg 31) \& q$
  return (a_1, a_0)
function CheckNorm(a, B)
  $t := a \gg 31$
  $t := a - (t \& 2 \cdot a)$
  return $1 - ((t - B) \gg 31)$
function MakeHint(a_1, a_0)
  if $a_0 > \gamma_2 || a_0 < -\gamma_2 || (a_0 = -\gamma_2 \& a_1 \neq 0)$ **then**
    return 1
  return 0

2) MontMul*:* We leverage Montgomery Reduction [25] to implement modular multiplication. It accepts two integers in the Montgomery domain and yields a result in $(-q, q)$. As the product of two 32-bit integers surpasses the register size, we utilize CUDA PTX assembly instructions to optimize register usage and minimize instruction count.

3) Decompose*:* We use macro definitions to encapsulate the two cases of distinct γ_2 values in this function. The computation of high- and low- elements adheres to the definitions in [6]. Results are derived using an approximately equal method that ensures constant-time execution and cost-efficiency.

4) CheckNorm*:* This function evaluates a against the threshold B. Given the potential negativity, we derive $|a|$ in constant time using the sign. We obtain the absolute value by masking $2a$ with the sign and subtracting from a. The comparison result is discerned from the sign after subtraction.

5) MakeHint*:* This function takes (a_1, a_0) as input, evaluates three conditions, and subsequently yields the hint bit.

The subsequent category is inherently more intricate, encompassing functions necessitating data interchange, including SHAKE, NTT, and DPSTPM. In implementing SHAKE, we follow the optimized warp-level design as delineated in [11]. The specifics of the remaining functions are elucidated below.

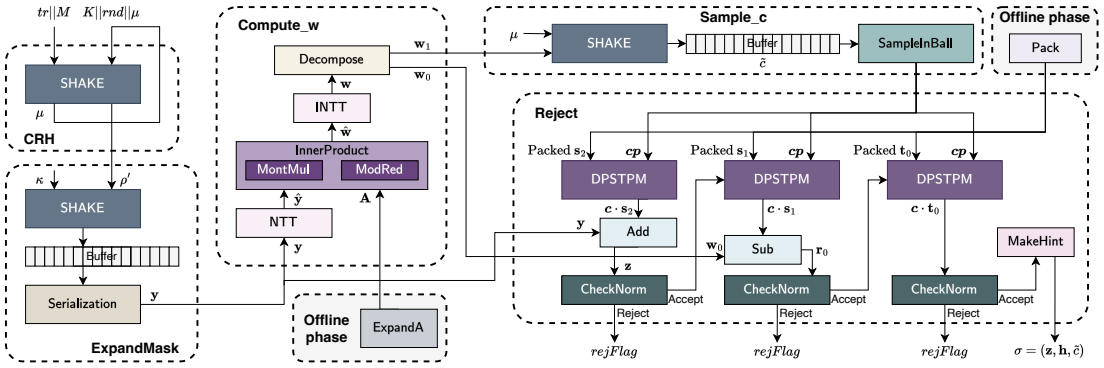


Fig. 3. Architectural design of the ML-DSA signing procedure, segmented by the implemented kernels and comprised of the associated inline functions.

TABLE III
PARAMETER CONFIGURATION FOR THE DPSTPM IMPLEMENTATION.

Scheme	Operation	τ	U	$2\tau U$	M	r
ML-DSA-44	$\boldsymbol{c} \cdot \mathbf{s}_1$	39	2	156	2^8	4
	$\boldsymbol{c} \cdot \mathbf{s}_2$	39	2	156	2^8	4
	$\boldsymbol{c} \cdot \mathbf{t}_0$	39	2^{13}	638976	2^{20}	1
ML-DSA-65	$\boldsymbol{c} \cdot \mathbf{s}_1$	49	4	392	2^9	5
	$\boldsymbol{c} \cdot \mathbf{s}_2$	49	4	392	2^9	6
	$\boldsymbol{c} \cdot \mathbf{t}_0$	49	2^{13}	802816	2^{20}	1
ML-DSA-87	$\boldsymbol{c} \cdot \mathbf{s}_1$	60	2	240	2^8	7
	$\boldsymbol{c} \cdot \mathbf{s}_2$	60	2	240	2^8	8
	$\boldsymbol{c} \cdot \mathbf{t}_0$	60	2^{13}	983040	2^{20}	1

1) NTT*:* Unlike [11] that employs both radix-2 and radix-8 approaches, we exclusively employ radix-2 with 128 threads. We store temporary values in shared memory and cache twiddle factors in constant memory. Each thread processes two elements with distance 2^{8-i} at level i, $i \in [1, 8]$. To prevent performance degradation due to bank conflicts arising from stride memory accesses, we strategically pad bank units, ensuring conflict-free access within identical read/write cycles.

2) DPSTPM*:* Our implementation focuses on the unified DPSTPM evaluation phase. The function accepts a packed vector v and position array $\boldsymbol{cp}$ as inputs and produce evaluation results. We deploy 128 threads and accomplish the evaluation in two rounds. In the kth round, the jth thread manages the element w_{128k+j}, conducting a τ-loop that sequentially adds $v_{128k+j+cp_i}$ to w_{128k+j}, with $k \in [0, 2)$, $j \in [0, 128)$, and $i \in [0, \tau)$. Notably, elements accessed by neighboring threads during identical read/write cycles are contiguous, enabling coalesced memory access for maximum speed.

Algorithm 8 GPU implementation of DPSTPM with early evaluation for $\boldsymbol{c} \cdot \mathbf{s}_2$

Input: $\boldsymbol{cp}$, $pack\_s_2$
Output: $\boldsymbol{c} \cdot \mathbf{s}_2$
__shared__ $s\_table, s\_s_2, s\_cp, s\_f$ ▷ Allocate shared memory
$s\_cp \leftarrow cp$, $s\_f := 0$, $reg := 0$
$tmp := pack\_s_2[\mathsf{tid}]$
$s\_table[\mathsf{tid} + N] := tmp$ ▷ Prepare PSTPM table
$s\_table[\mathsf{tid} + N + 128] := tmp$
$s\_table[\mathsf{tid}]$, $s\_table[\mathsf{tid} + 2N] := Mask\_s_2 - tmp$
$s\_table[\mathsf{tid} + 128]$, $s\_table[\mathsf{tid} + 2N + 128] := Mask\_s_2 - tmp$
for $j \in [0, \tau)$ **do** ▷ First round DPSTPM evaluation
  $\mathsf{idx} := \mathsf{tid} + s\_cp[j]$
  $reg = reg + s\_table[\mathsf{idx}]$
__syncthreads()
for $i \in [0, k)$ **do**
  $res := (reg \& Mask\_P) - \tau \cdot \eta$
  $reg := reg \gg Bit\_P$
  $res := \mathbf{w}_0[i][\mathsf{tid}] - res \bmod q$
  $rejFlag :=$ __any_sync(CheckNorm$(\gamma_2 - \beta, res)$)
  if $\mathsf{lid} = 0$ *and* $rejFlag = 1$ **then**
    $s\_f := 1$
  __syncthreads()
  if s_f **then return**
  $s\_s_2[i][\mathsf{tid}] := res$ ▷ Write results to shared memory
for $j \in [0, \tau)$ **do** ▷ Second round DPSTPM evaluation
  $\mathsf{idx} := \mathsf{tid} + s\_cp[j]$
  $reg = reg + s\_table[\mathsf{idx} + 128]$
__syncthreads()
for $i \in [0, k)$ **do**
  $res := (reg \& Mask\_P) - \tau \cdot \eta$
  $reg := reg \gg Bit\_P$
  $res := \mathbf{w}_0[i][\mathsf{tid} + 128] - res \bmod q$
  $rejFlag :=$ __any_sync(CheckNorm$(\gamma_2 - \beta, res)$)
  if $\mathsf{lid} = 0$ *and* $rejFlag = 1$ **then** $s\_f := 1$
  __syncthreads()
  if s_f **then return**
  $s\_s_2[i][\mathsf{tid} + 128] := res$ ▷ Write results to shared memory

C. Optimized Reject Kernel

Our implemented kernel for the reject process receives the challenge $\boldsymbol{cp}$, the secret key elements, and relevant polynomials as inputs, conducts arithmetic operations over R, and checks the norms to derive a valid signature. Within this, we incorporate three previously discussed optimizations. First, we employ the DPSTPM algorithm illustrated in Section III-B. Second, we introduce a merging technique that leverages the early evaluation approach. Third, we embrace an order prioritizing $\mathbf{r}_0$ norm-checking for easier rejection. The conventional signing demands numerous NTT and INTT computations. Transitioning to the DPSTPM approach reduces multiplications and facilitates the evaluation of multiple polynomials through a singular computation. Combining the depth-first computational mode with this optimized order, we can expedite rejection, thereby curtailing unnecessary computations.

Merging with Early Evaluation. We introduce a technique merging DPSTPM with early evaluation to expedite invalid signature rejection, which is compatible with all three evaluation processes. The principle is splitting the evaluation into two rounds. After each round, a sequential unpacking is executed

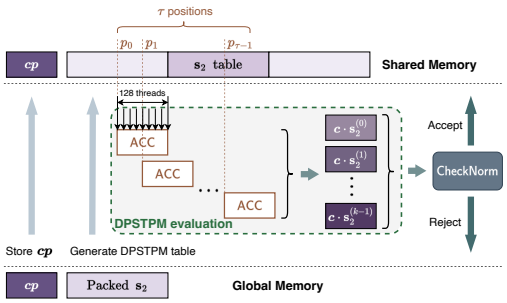


Fig. 4. Optimized memory access pattern for DPSTPM evaluation of $\boldsymbol{c} \cdot \mathbf{s}_2$.

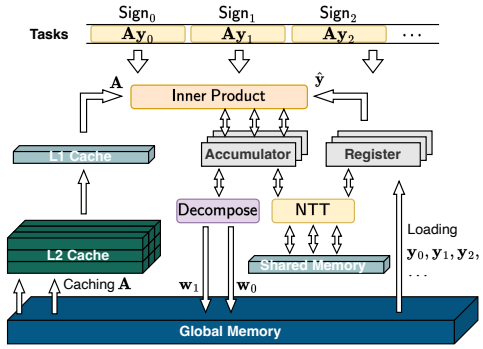


Fig. 5. Layered Caching Strategy for the computation of $\mathbf{A} \cdot \mathbf{y}$.

to derive each half results, followed by arithmetic operations and norm bound assessments. An illustrative implementation of the merged method for computing $\mathbf{r}_0 := \mathbf{w}_0 - \boldsymbol{c} \cdot \mathbf{s}_2$ and verifying $||\mathbf{r}_0||_\infty < \gamma_2 - \beta$ in ML-DSA-44 is detailed in Algorithm 8. Here, $r = k$, tid represents the thread indices, and lid signifies the lane indices of threads within a warp, with $\text{tid} \in [0, 128)$ and $\text{lid} \in [0, 32)$. The procedure commences with generating a table base on $\mathbf{s}_2$ for DPSTPM evaluation. Each thread subsequently undertakes the τ-loop for a singular round of evaluation, and then unpacks the ith polynomial coefficient and subtract it from $\mathbf{w}_0[i][\text{tid}]$ to check the norm. Finally, thread states are synchronized to determine if rejection is required, subsequently updating the rejection flag.

Optimized Memory Access Pattern. To achieve low latency data access across all evaluations, we allocate $\boldsymbol{cp}$ and the three DPSTPM tables to shared memory, while frequently accessed accumulators are designated to registers. Unlike other evaluations, $\boldsymbol{c} \cdot \mathbf{s}_2$ needs to be temporarily stored for subsequent computations. Thus, we transfer it from registers to shared memory to maintain balance, as illustrated in Fig. 4. Such an approach presents dual advantages. First, it conserves registers for other computations, reducing the risk of register overuse, which diminish kernel occupancy. Second, it ensures relatively fast access speeds for both writing and retrieving $\boldsymbol{c} \cdot \mathbf{s}_2$.

D. Caching Key Component

In the basic design of the Compute_w kernel, a 128-thread block is dedicated to manage computations of a signing task. For the inner-product of matrix $\mathbf{A}$ and vector $\mathbf{y}$, each thread loads two coefficients of both entities from the global memory into the block's specific memory segment. The thread then performs multiplication and accumulation to complete the computation. Notably, when the server functions as an entity with an invariant key, matrix $\mathbf{A}$ is consistently reloaded across all active blocks. In the architecture of [11], the memory pool segment dedicated to each signing task reserves space for $\mathbf{A}$, prompting each signing task to load it autonomously. This approach offers no computational advantages during compilation and results in significant memory access overhead.

Leveraging caching for $\mathbf{A}$ presents a potential solution to the issue, but its substantial memory requirements exceed shared memory capacity, especially for larger parameter sets like ML-DSA-65 and ML-DSA-87. To avoid impacting SM occupancy and overall performance, we separate $\mathbf{A}$ from task-specific memory segments and allocate it to a discrete global memory region accessible by all blocks. Under this memory access paradigm, and given the frequent loading patterns, both L2 and L1 caches play pivotal roles in caching the matrix. The comprehensive design is depicted in Fig. 5. In this architecture, $\mathbf{A}$ follows the memory hierarchy, being cached in a stratified manner, transitioning from global memory to L2 and then to L1. Each block loads its associated $\mathbf{y}_i$ to registers to expedite access, given its recurrent use during computations. We allocate $2k$ registers per thread as accumulators for summative results, with shared memory reserved for NTT and INTT data interchange. Finally, we derive $\mathbf{w}_0$ and $\mathbf{w}_1$ by decomposition, which are then transferred back to global memory.

E. Sparse Polynomial Sampler and Adaptations

To cater to both the original sampling method and our DPSTPM strategy, we devise three distinct versions of the sparse polynomial sampler for the sampling of $\boldsymbol{c}$.

The first version follows the original procedure, preserving the entire $\boldsymbol{c}$. A random stream is generated from $\tilde{c}$, with the initial 8 bytes determining signs and the remainder generating positions. However, this approach requires traversing $\boldsymbol{c}$ to obtain the τ positions in DPSTPM. Consequently, our second variant directly archives positions $\boldsymbol{cp}$ during sampling, omitting the retention of $\boldsymbol{c}$. Here, we set $cp_j := -i$ when $c_i = 1$ and $cp_j := n - i$ when $c_i = -1$. Nonetheless, both versions are limited in parallel computation potential due to the random nature of the data, which can lead to data conflicts and read/write competition among threads.

Thus, we introduce a third version that allow parallel computations. We construct a Boolean lookup table to track existing positions, thereby ensuring generating τ distinct positions. This method eliminates the need for exhaustive comparisons or traversals, which is more efficient and can serve as a complementary solution given it inherently alters test vectors.

V. Performance Evaluation

A. Experimental Setup

The C/C++ source code is compiled utilizing g++ 12.2.0, while the GPU implementation uses CUDA 11.8. All compilations and executions are conducted on an Arch Linux system with kernel 5.15. For CPU benchmarks, we use an Intel(R) XEON W7-2495X CPU with 2.5GHz base frequency. The performance evaluations are performed on a NVIDIA Tesla

TABLE IV
KERNEL PROFILING RESULTS FOR $(\mathbf{w}_1, \mathbf{w}_0) = \mathbf{A} \cdot \mathbf{y}$ COMPUTATION PRE- AND POST-OPTIMIZATION ACROSS THE THREE ML-DSA PARAMETER SETS. HERE, FMA REPRESENTS THE FUSED MULTIPLY ADD/ACCUMULATE PIPELINE, WHILE ALU DENOTES THE ARITHMETIC LOGIC UNIT.

Parameter Set		ML-DSA-44			ML-DSA-65			ML-DSA-87		
Optimization Method		Before	After	Opt.	Before	After	Opt.	Before	After	Opt.
Execution Time (μs)		69.86	56.64	**1.23**×	104.64	80.8	**1.30**×	169.76	119.1	**1.43**×
Achieved Occupancy (%)		41.59	42.92	**+3.19%**	38.92	38.99	**+0.19%**	39.08	39.58	**+1.27%**
Throughput (%)	Compute	42.39	51.26	**+20.91%**	38.29	50.18	**+31.04%**	34.6	48.9	**+41.31%**
	Memory	47.36	56.89	**+20.12%**	44.33	55.98	**+26.28%**	49.08	52.73	**+7.44%**
Pipeline Utilization (%)	FMA	30.67	38.08	**+24.15%**	27.98	36.77	**+31.43%**	22.84	36.08	**+57.96%**
	ALU	29.16	36.57	**+25.39%**	26.38	36.63	**+38.84%**	25.31	36.02	**+42.32%**
Cache Hit Rate (%)	L1	9.2	57.34	**+523.40%**	3.21	60.78	**+1790.76%**	2.14	63.77	**+2876.22%**
	L2	42.67	72.15	**+69.08%**	41.7	76.21	**+82.76%**	38.71	78.31	**+102.31%**

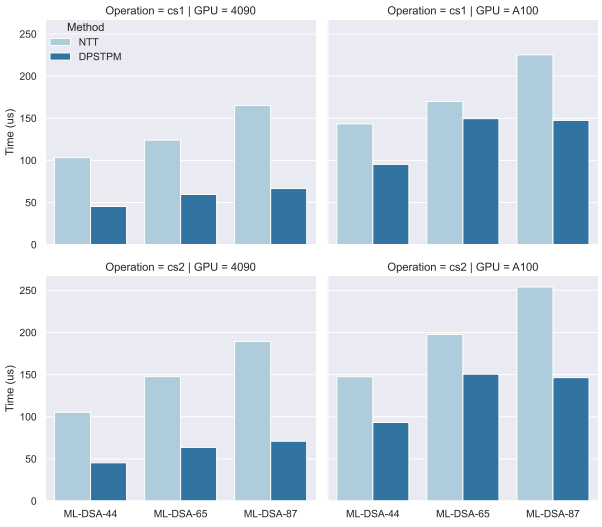


Fig. 6. Performance analysis of batch processing 10,000 executions for $\boldsymbol{c} \cdot \mathbf{s}_1$ and $\boldsymbol{c} \cdot \mathbf{s}_2$. Vertical parallelism values are $r = 4, 5, 7$ for $\boldsymbol{c} \cdot \mathbf{s}_1$, and $r = 4, 6, 8$ for $\boldsymbol{c} \cdot \mathbf{s}_2$, across the three respective parameter sets.

TABLE V
EXECUTION TIME OF THREE SPARSE POLYNOMIAL SAMPLERS (MEASURED IN MICROSECONDS (μs)).

Platform	Parameter Set	Sampling Method		
		I	II	III
GeForce RTX 4090	ML-DSA-44	87.52	87.62	71.04
	ML-DSA-65	86.72	87.36	72.54
	ML-DSA-87	101.57	100.32	88.93
Tesla A100	ML-DSA-44	130.85	131.84	115.3
	ML-DSA-65	132.99	133.7	117.63
	ML-DSA-87	159.42	160.19	145.18

A100 80G PCIe and a NVIDIA GeForce RTX 4090. In the experiments, we batch processing 10,000 computational tasks, with execution time recorded in microseconds (μs). Overall efficiency is represented as throughput in operations per second (OP/s), indicating the number of signatures successfully completed within a second.

B. Evaluation of Optimization Effectiveness

Below, we undertake a series of experiments to evaluate the efficacy of the proposed methods.

Comparison of DPSTPM and NTT. Fig. 6 compares execution times between the NTT approach and our DPSTPM for $\boldsymbol{c} \cdot \mathbf{s}_1$ and $\boldsymbol{c} \cdot \mathbf{s}_2$ evaluations, which extensively utilize our proposed parallel technique, with vertical parallelism denoted by $r = k$ or l, respectively. Performance metrics across both the 4090 and A100 GPUs are presented. For each computation of $\boldsymbol{c} \cdot \mathbf{s}_1$ and $\boldsymbol{c} \cdot \mathbf{s}_2$, a batch of 10,000 computational tasks is processed. Our DPSTPM consistently surpasses the original NTT method, showcasing execution speed enhancements of 52.0% to 59.7% for $\boldsymbol{c} \cdot \mathbf{s}_1$ and 56.9% to 62.5% for $\boldsymbol{c} \cdot \mathbf{s}_2$. The A100 GPU shows slightly reduced acceleration effect for ML-DSA-65, primarily attributable to parameter-induced influences on hardware task scheduling. For other parameter sets, the performance enhancement remains between 33.6% and 42.3%. Considering the vertical parallelism, which is $r = 4, 5, 7$ for $\boldsymbol{c} \cdot \mathbf{s}_1$ and $r = 4, 6, 8$ for $\boldsymbol{c} \cdot \mathbf{s}_2$ across the three parameter sets, the execution time for $\boldsymbol{c} \cdot \mathbf{s}_2$ is longer than that for $\boldsymbol{c} \cdot \mathbf{s}_1$, by approximately 4 μs on the 4090 GPU. Unlike NTT, the main performance dominant for the DPSTPM is the variable τ, which affects both the computational complexity and the entire computation.

Speedups through Caching. Table IV presents kernel profiling metrics for the computation $(\mathbf{w}_1, \mathbf{w}_0) = \mathbf{A} \cdot \mathbf{y}$, comparing pre- and post-optimization results. The enhancement is primarily due to our refined memory configuration and access strategies. Our approach leverages various classes of on-chip memory based on specific access properties, leading to substantial increases in both L1 and L2 cache hit rates by 2876.22% and 102.31%, respectively. This leads to more efficient memory accesses, reduced execution time, optimized processor resource usage, and improved pipeline scheduling. As our method primarily uses addition and logical operations, there is an increase in the FMA and ALU pipeline utilization. For the ML-DSA-44 parameter set, execution time is reduced from 69.86 μs to 56.64 μs, yielding a 1.23× speedup. Similar enhancements are observed for ML-DSA-65 and ML-DSA-87, with speedups of 1.30× and 1.43×, respectively.

Performance of Sampler Variants. Table V presents execution times for three sparse polynomial samplers, benchmarked on both 4090 and A100 GPU platforms. Specifically, method I is the original method where entire $\boldsymbol{c}$ is stored, method II represents the variant which stores $\boldsymbol{cp}$, whereas method III directly samples positions. The first two methods are incompatible with parallel computing, primarily due to the data conflicts and competition. Comparing across both GPU platforms, method I slightly outperforms II, though III consistently demonstrates superior performance. However,

TABLE VI
THROUGHPUT COMPARISON BETWEEN C, AVX2 IMPLEMENTATIONS, THE GPU APPROACH OF [11], AND OUR IMPLEMENTATION. MEASUREMENTS ARE GIVEN IN OP/S. FOR SPEEDUP METRICS, ONLY THE STREAMING METHOD IS CONSIDERED: THE FIRST LINE INDICATES SPEEDUP RELATIVE TO THE C BASELINE, WHILE THE SECOND DENOTES IMPROVEMENT OVER [11]. THE SAMPLING TECHNIQUE EMPLOYED IS METHOD II.

Parameter Set	CPU		Dilithium [11]		This Work (ML-DSA)						
	Ref	AVX2	A100	4090	CPU[1]	A100			4090		
			Streaming	Streaming	Ref	Batching	Streaming	Speedup	Batching	Streaming	Speedup
ML-DSA-44	3,105	14,679	765,855	984,803	3,499	771,943	884,683	**284.9**×	1,049,238	1,080,871	**348.1**×
					+12.7%			**+15.5%**			**+9.8%**
ML-DSA-65	2,267	9,121	513,468	649,498	2,471	538,742	615,811	**271.6**×	736,109	884,195	**390.0**×
					+9.0%			**+19.9%**			**+36.1%**
ML-DSA-87	1,618	7,137	396,894	488,006	1832	448,515	540,285	**333.9**×	554,651	785,277	**485.3**×
					+13.2%			**+36.1%**			**+60.9%**

[1]Performance of the original C implementation integrated with our DPSTPM technique and optimized norm checking order.

TABLE VII
THROUGHPUT COMPARISON OF DILITHIUM IMPLEMENTATIONS ACROSS DIFFERENT PLATFORMS. MEASUREMENTS ARE REPRESENTED IN OP/S. THE THROUGHPUT IS DERIVED FROM THE CYCLE COUNTS, TIMES, AND FREQUENCIES PRESENTED IN THE RESPECTIVE PUBLICATIONS.

Related Work	Dilithium2	Dilithium3	Dilithium5	Platform
[7]	23,256	15,873	10,526	UltraScale+ FPGA
[8]	3,448	2,167	1,977	Artix-7 FPGA
[10]	2,310	1,377	1,044	ARM Cortex-A72
[12]	33,965	14,875	20,396	AGX Xavier GPU

method I necessitates an additional step to generate ***cp*** in subsequent kernels. This particularity highlights the superior compatibility of method II within our implemented framework.

C. Overall Performance

Table VI summarizes throughput results of the basic software implementation, GPU implementations from related work of Dilithium [11], and our solutions. We utilize the officially open-source version of ML-DSA[1] to serve as our CPU baseline, containing both a C reference and an AVX2-optimized implementation. The study [11] represents the most recent and highest-performing GPU implementation of Dilithium. For a comprehensive assessment, we executed performance comparison on identical GPU platforms.

We show around 10% speedups on the C implementation with our DPSTPM technique and optimized norm checking order. For GPU performance evaluation, we provide two distinct methodologies include batching and streaming. The batching method processes 10,000 signing tasks in a single batch, while the streaming method initiates 10 CUDA streams, each processing 1,000 signing tasks. The results show a consistent superior performance of the streaming method over batching, primarily due to its ability to hide memory transfer latency. On the 4090 GPU, the streaming approach achieves a 41.6% acceleration relative to batching for the ML-DSA-87. Similarly, on the A100 GPU, it consistently surpasses batching, showing enhancements ranging from 14.3% to 20.5%.

Using the server-grade A100 GPU, our framework exhibits speedups of 284.9×, 271.6×, and 333.9× for the three ML-DSA parameter sets against the C baseline. Similarly, on the consumer-oriented 4090 GPU, our solution achieves accelerations of 348.1×, 390.0×, and 485.3×, respectively. Compared to the AVX2-optimized method, we achieve speedups ranging from 60.3× to 75.7× on the A100 GPU and 73.6× to 110.0× on the 4090 GPU. While CPU designs can benefit from multi-threaded executions, GPU architectures show superior capability in processing computational tasks concurrently. This capacity enables GPUs to function as efficient co-processors, thereby alleviating the computational burden on CPUs and allowing them to prioritize scheduling tasks.

Furthermore, when benchmarked against the generalized GPU design in [11], our implementation demonstrates improvements of 9.8%, 36.1%, and 60.9% for the three ML-DSA parameter sets on the 4090 GPU. Additionally, we observe enhancements of 15.5%, 19.9%, and 36.1% on the A100 platform. These significant improvements are attributable to our refined signing procedure combined with server-oriented architectural considerations.

Table VII delineates the throughput results derived from several related studies [7], [8], [10], [12] across various computational platforms. The research elucidated in [10] focuses on ARM processors, leveraging the acceleration capabilities of NEON. The work [12] delineates GPU implementations, specifically tailored for the AGX Xavier GPU. Concurrently, the studies [7], [8] concentrate on FPGA-based hardware designs. While there is potential to enhance throughput by allocating more hardware area, GPU implementations often emerge as the preferred choice in server-grade scenarios requiring high-performance and real-time solutions.

VI. CONCLUSION

In this work, we address a significant gap in GPU-optimized implementations for ML-DSA in server environments. Our tailored, server-centric design, enhanced with novel theoretical optimizations, achieves substantial performance gains. The demonstrated speedups against both CPU benchmarks and existing methods show the effectiveness of our approach. Our work contributes to post-quantum cryptographic deployments and underscores the potential of specialized GPU designs in cryptographic performance. As a future direction, we aim to study the acceleration of other standard post-quantum digital signatures and explore the integration of our design in various protocols to facilitate a seamless post-quantum migration.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China (No. 2022YFB2701601), the National Natural

[1]https://github.com/pq-crystals/dilithium/tree/standard

Science Foundation of China (No. 62132008), the General Project of State Key Laboratory of Cryptography (No. MMK-FKT202227), the Technical Standard Project of Shanghai Scientific and Technological Committee (No. 21DZ2200500), and the Shanghai Collaborative Innovation Fund (No. XTCX-KJ-2023-54).

REFERENCES

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," SIAM J. Comput., vol. 26, no. 5, pp. 1484–1509, 1997.

[2] NIST, "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," 2016.

[3] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, C. Miller, D. Moody, and R. Peralta, "Status report on the third round of the NIST post-quantum cryptography standardization process," US Department of Commerce, NIST, 2022.

[4] N. I. of Standards and Technology, "Fips 204 (initial public draft): Module-lattice-based digital signature standard," 2023.

[5] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium: A lattice-based digital signature scheme," IACR Trans. Cryptogr. Hardw. Embed. Syst., vol. 2018, no. 1, pp. 238–268, 2018.

[6] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and Stehlé, "CRYSTALS-Dilithium: Algorithm specifications and supporting documentation," Submission to the NIST's post-quantum cryptography standardization process, 2020.

[7] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-performance hardware implementation of crystals-dilithium," in International Conference on Field-Programmable Technology, (IC)FPT 2021, Auckland, New Zealand, December 6-10, 2021. IEEE, 2021, pp. 1–10.

[8] C. Zhao, N. Zhang, H. Wang, B. Yang, W. Zhu, Z. Li, M. Zhu, S. Yin, S. Wei, and L. Liu, "A compact and high-performance hardware architecture for CRYSTALS-Dilithium," IACR Trans. Cryptogr. Hardw. Embed. Syst., vol. 2022, no. 1, pp. 270–295, 2022.

[9] S. Ricci, L. Malina, P. Jedlicka, D. Smékal, J. Hajny, P. Cíbik, P. Dzurenda, and P. Dobias, "Implementing crystals-dilithium signature scheme on fpgas," in ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021, D. Reinhardt and T. Müller, Eds. ACM, 2021, pp. 1:1–1:11.

[10] H. Becker, V. Hwang, M. J. Kannwischer, B. Yang, and S. Yang, "Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1," IACR Trans. Cryptogr. Hardw. Embed. Syst., vol. 2022, no. 1, pp. 221–244, 2022.

[11] S. Shen, H. Yang, W. Dai, Z. Liu, and Y. Zhao, "High-throughput GPU implementation of Dilithium post-quantum digital signature," CoRR, vol. abs/2211.12265, 2022.

[12] S. C. Seo and S. An, "Parallel implementation of crystals-dilithium for effective signing and verification in autonomous driving environment," ICT Express, vol. 9, no. 1, pp. 100–105, 2023.

[13] J. Zheng, F. He, S. Shen, C. Xue, and Y. Zhao, "Parallel small polynomial multiplication for dilithium: A faster design and implementation," in Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022. ACM, 2022, pp. 304–317.

[14] D. O. C. Greconici, M. J. Kannwischer, and A. Sprenkels, "Compact Dilithium implementations on Cortex-M3 and Cortex-M4," IACR Trans. Cryptogr. Hardw. Embed. Syst., vol. 2021, no. 1, pp. 1–24, 2021.

[15] P. Ravi, S. S. Gupta, A. Chattopadhyay, and S. Bhasin, "Improving speed of Dilithium's signing procedure," in Smart Card Research and Advanced Applications, ser. Lecture Notes in Computer Science, 2020, pp. 57–73.

[16] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: FrodoKEM, NewHope, and Kyber," IEEE Trans. Parallel Distributed Syst., vol. 32, no. 3, pp. 575–586, 2021.

[17] T. Ono, S. Bian, and T. Sato, "Automatic parallelism tuning for Module Learning with Errors based post-quantum key exchanges on GPUs," in IEEE International Symposium on Circuits and Systems, ISCAS 2021, Daegu, South Korea, May 22-28, 2021. IEEE, 2021, pp. 1–5.

[18] L. Wan, F. Zheng, and J. Lin, "TESLAC: accelerating lattice-based cryptography with AI accelerator," in Security and Privacy in Communication Networks - 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6-9, 2021, Proceedings, Part I, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, J. García-Alfaro, S. Li, R. Poovendran, H. Debar, and M. Yung, Eds., vol. 398. Springer, 2021, pp. 249–269.

[19] Y. Gao, J. Xu, and H. Wang, "cuNH: Efficient GPU implementations of post-quantum KEM NewHope," IEEE Trans. Parallel Distributed Syst., vol. 33, no. 3, pp. 551–568, 2022.

[20] W. Lee, H. Seo, Z. Zhang, and S. O. Hwang, "TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU," IEEE Access, vol. 10, pp. 20 616–20 632, 2022.

[21] L. Wan, F. Zheng, G. Fan, R. Wei, L. Gao, Y. Wang, J. Lin, and J. Dong, "A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator," in Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part III, ser. Lecture Notes in Computer Science, vol. 13556. Springer, 2022, pp. 514–534.

[22] S. Sun, R. Zhang, and H. Ma, "Efficient parallelism of post-quantum signature scheme SPHINCS," IEEE Trans. Parallel Distributed Syst., vol. 31, no. 11, pp. 2542–2555, 2020.

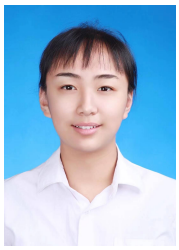
[23] W. Dai, B. Sunar, J. Schanck, W. Whyte, and Z. Zhang, "NTRU modular lattice signature scheme on CUDA GPUs," in 2016 International Conference on High Performance Computing & Simulation (HPCS), 2016, pp. 501–508.

[24] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in Advances in Cryptology - CRYPTO 1986, ser. Lecture Notes in Computer Science, vol. 263, 1986, pp. 311–323.

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

Shiyu Shen received the PhD degree from School of Computer Science, Fudan university in 2024. Her research interests include lattice-based cryptography, homomorphic encryption, and cryptographic engineering. Her email address is crypto@sher1e.dev.

Hao Yang received the PhD degree from College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics in 2024. His research interests include homomorphic encryption, lattice-based cryptography, and cryptographic engineering. His email address is crypto@d4rk.dev.



Wenqian Li , born in 2001. Master candidate in the School of Computer Science, Fudan University. Her main research interests include post-quantum cryptography and cryptographic engineering.

Yunlei Zhao received his PhD at Fudan University in 2004. He is now a distinguished professor at Fudan university. His main research interests include post-quantum cryptography, cryptographic protocols, theory of computing. His email address is ylzhao@fudan.edu.cn.