# 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 170.7× to 294.2× 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 implementation for the ML-DSA and for GPU design of 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 School of Computer Science, Fudan University, Shanghai, China.

H. Yang is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics.

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 170.7× to 202.4× on the A100 and 208.6× to 294.2× on the 4090 GPU across the three parameter sets. In comparison to the AVX2 optimized implementation, our implementation is faster by factors ranging from 37.4× to 42.4× on the A100 and 45.6× to 61.7× 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

We denote $n$ as a power-of-two and $q$ as a prime satisfying $q \equiv 1 \pmod{2n}$. Let $\mathbb{Z}$ represent the set of integers, and let $R = \mathbb{Z}[X]/(X^n + 1)$ be 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 restrict our attention to integer intervals. For instance, the range $[0, n]$ encompasses elements from the set $\{0, 1, ..., n\}$.

Elements in $R$ (or $R_q$) are represented as polynomials, typically denoted using bold, italicized lowercase letters like $\boldsymbol{f}$. In contrast, vectors are indicated using bold, upright lowercase letters such as $\mathbf{x}$. Every polynomial element, whether $\boldsymbol{f} \in R$ or $\boldsymbol{f} \in R_q$, can be exclusively expressed as $\boldsymbol{f} = \sum_{i=0}^{n-1} f_i X^i$, where $f_i$ belongs to $\mathbb{Z}$ (or $\mathbb{Z}_q$) for all $i \in [0, n)$. Matrices over $R$ or $R_q$ are denoted using uppercase boldface letters, e.g., $\mathbf{M}$. For a polynomial $\boldsymbol{f} \in R$, its $\ell_\infty$-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 $\ell_\infty$-norm is defined as $||\mathbf{x}||_\infty = \max\{||\boldsymbol{x}_i||_\infty\}$.

### B. Polynomial Multiplication

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

**Number-Theoretic Transform (NTT).** The NTT is an efficient strategy for polynomial multiplications within the ring $R_q$. Representing $\zeta$ as the primitive $2n$-th root of unity, this method starts by transforming the polynomials to the NTT domain, producing $\hat{\boldsymbol{a}}$ and $\hat{\boldsymbol{c}}$ such that $\hat{a}_j = \sum_{i=0}^{n-1} a_i \zeta^{(2i+1)j} \pmod{q}$ and analogously for $\hat{c}_j$, with $i, j \in [0, n)$. The multiplication is then reduced to point-wise multiplication by $\hat{b}_j = \hat{a}_j \circ \hat{c}_j$. Following this, $\hat{b}$ is reverted to the standard domain using $b_i = \frac{1}{n} \sum_{i=0}^{n-1} \hat{b}_j \zeta^{(2i+1)j} \pmod{q}$. Employing NTT, the complexity is reduced from $O(n^2)$ to $O(n \log n)$.

**Sparse Ternary Polynomial Multiplication (STPM).** Let $\boldsymbol{d}$ be a polynomial of degree $2n-2$ such that $\boldsymbol{d} = \boldsymbol{a} \cdot \boldsymbol{c} = \sum_{i=0}^{2n-2} d_i X^i$. We then get $d_j = \sum_{i=0}^{j} c_i a_{j-i}$ for $j \in [0, n)$ and $d_j = \sum_{i=j-n+1}^{n-1} c_j a_{j-i}$ for $j \in [n, 2n-1)$. Subsequently, $b_j = d_j - d_{j+n}$ for $j \in [0, n-1)$, and $b_{n-1} = d_{n-1}$. Introducing a condition where $a_{i-n} = -a_i$ for $i \in (0, n)$, the earlier formula restructures to $b_j = \sum_{i=0}^{n-1} c_i a_{j-i}$ for $j \in [0, n)$. If $\boldsymbol{c}$ is recognized as a sparse ternary polynomial with $\tau$ non-zero coefficients equaling 1 or $-1$, the multiplication computation complexity diminishes. Specifically, multiplications can be substituted with conditional structures, as detailed in Algorithm 1.

**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** $i \in [0, n)$ **do**
  $w_i := 0$, $v_i := a_i$, $v_{i-n} := -a_i$
**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**
      $w_j := w_j - v_{j-i}$
**for** $i \in [0, n)$ **do**
  $u_i := w_i \pmod{q}$
**return** $u = \sum_{i=0}^{n-1} u_i \cdot x^i$

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

$\mu \in \{0,1\}^{512} := \mathcal{H}(tr \| M)$
$rnd := \{0\}^{256}$ ▷ Deterministic variant
$\rho' \in \{0,1\}^{512} := \mathcal{H}(K \| rnd \| \mu)$
$\hat{\mathbf{A}} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
$\kappa := 0$; $(\mathbf{z}, \mathbf{h}) := \perp$
**while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} \in S_{\gamma_1}^\ell := \mathsf{ExpandMask}(\rho', \kappa)$
  $\mathbf{w} := \hat{\mathbf{A}} \cdot \mathbf{y}$
  $\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$
  $\tilde{c} := (\tilde{c}_1, \tilde{c}_2) \in \{0,1\}^{256} \times \{0,1\}^{2\lambda - 256} := \mathcal{H}(\mu \| \mathbf{w}_1)$
  $\boldsymbol{c} \in B_\tau := \mathsf{SampleInBall}(\tilde{c}_1)$
  $\mathbf{z} := \mathbf{y} + \boldsymbol{c} \cdot \mathbf{s}_1$
  $\mathbf{r}_0 := \mathsf{LowBits}_q(\mathbf{w} - \boldsymbol{c} \cdot \mathbf{s}_2, 2\gamma_2)$
  **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **or** $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\mathbf{z}, \mathbf{h}) := \perp$
  **else**
    $\mathbf{h} := \mathsf{MakeHint}_q(-\boldsymbol{c} \cdot \mathbf{t}_0, \mathbf{w} - \boldsymbol{c} \cdot \mathbf{s}_2 + \boldsymbol{c} \cdot \mathbf{t}_0, 2\gamma_2)$
    **if** $\|\boldsymbol{c} \cdot \mathbf{t}_0\|_\infty \geq \gamma_2$ or number of 1's in $\mathbf{h}$ exceeds $\omega$ **then**
      $(\mathbf{z}, \mathbf{h}) := \perp$
  $\kappa := \kappa + \ell$

### C. ML-DSA and Parallel STPM

ML-DSA is a digital signature scheme, verified as strongly unforgeable in the QROM based on the decisional Module-LWE and the SelfTargetMSIS assumptions [4]. This scheme emerges from the CRYSTALS-Dilithium, a proposal submitted to the NIST PQC standardization project. The signing procedure is summarized in Algorithm 2, with three parameter sets detailed 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 | $n$ | $q$ | $d$ | $\tau$ | $\gamma_1$ | $\gamma_2$ | $(k,\ell)$ | $\eta$ | $\beta$ | $\omega$ | $\lambda$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

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

- $\boldsymbol{c}$ is a ternary polynomial where $c_i$ ranges within $\{-1, 0, 1\}$, with only $\tau$ 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 $[-2^{d-1}, 2^{d-1}]$.

Due to these properties, the conventional approach of using NTT to compute $\boldsymbol{c} \cdot \mathbf{s}_1$, $\boldsymbol{c} \cdot \mathbf{s}_2$ and $\boldsymbol{c} \cdot \mathbf{t}_0$ becomes inefficient. This method transforms smaller integers in $[-\eta, \eta]$ and $[-2^{d-1}, 2^{d-1}]$ to $\mathbb{Z}_q$, where $\eta$ and $2^{d-1}$ are significantly smaller than $q$, resulting in increased memory usage.

To optimize based on the aforementioned property, a parallel STMP 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. Initially, the vector $\mathbf{a}$ is packed into an array $\boldsymbol{v} := \{v_i\}$. Subsequently, parallel computation results are stored in the array $\boldsymbol{w} := \{w_j\}$, which is derived by summing elements in $\boldsymbol{v}$ according to the values of $c_i$. Each polynomial multiplication result is then extracted by decomposing $\boldsymbol{w}$. Here, $U$ denotes the upper limit of $a_i$, $M$ represents the boundary for $\tau$ multiplicative additions, and $\gamma$ 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 inherent parallel capabilities to enhance the speed of the signing procedure. While CPUs excel at managing the logical flow of general-purpose programs, GPUs have been specially designed for tasks that demand intense parallel processing. This design principle renders GPUs exceptionally effective for computationally demanding tasks, offloading much of the burden traditionally borne by CPUs.

Within this architecture, instructions are executed in streams by threads. These threads can be organized into blocks during processing. 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, a block is partitioned into warps for single-instruction-multiple-thread (SIMT) execution. Each warp comprises a set of 32 threads with sequential thread 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 (RF), L1 cache, shared memory (SMEM), and constant caches, where the RF boasts the fastest access speed. Beyond these are larger regions with higher IO latency shared across all SMs, such as the L2 Cache, global memory (GMEM), local memory, texture, and constant memory. Among them, only the RF, SMEM, and GMEM support read-write operations, while texture and constant memory are cached in L1 and constant caches, respectively.

**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**
    $u_i^{(r-1-j)} := (t \bmod M) - \tau U (\mathrm{mod} q)$
    $t := \lfloor t/M \rfloor$
**return** $\mathbf{u} = [\boldsymbol{u}^{(0)}, \cdots, \boldsymbol{u}^{(r-1)}]^T$

## III. THEORETICAL OPTIMIZATIONS TO ML-DSA SIGNING PROCEDURE

In this section, we detail theoretical improvements to theML-DSA signing procedure. After outlining the signing architecture, we present our depth-prior optimization, strategies for branch elimination, and a refined order for norm checks, all aimed at enhancing signing efficiency.

### A. Signing Architecture Overview

To offer a lucid understanding of the process, we succinctly summarize the arithmetic involved. We decompose and reconstruct the operations intrinsic to the ML-DSA signing procedure considering the associativity, and the detailed graphical representation is shown in Fig. 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, the CRH is invoked twice with different inputs, including the concatenated form $tr||M$ and $K||rnd||\mu$, each producing the first 64 bytes to be designated as the element $\mu$ and $\rho'$, respectively.
- ExpandA: This function adopts the rejection sampling mechanism to sample uniform polynomials and obtain the matrix $\mathbf{A} \in R_q^{k \times \ell}$. The random streams are generated from the seed $\rho$ and obtained via SHAKE-128. In this mechanism, sequences of 3 bytes are sequentially extracted and compared against $q$. Those byte sequences

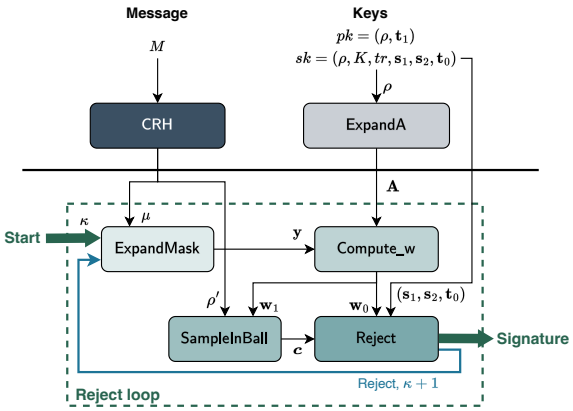



Fig. 1. Structure of the ML-DSA signing procedure.

**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:** $\boldsymbol{u} = \boldsymbol{c} \cdot \boldsymbol{a} \in R_q$

**for** $i \in [0, n)$ **do**
  $w_i := 0$
  $v_i := a_i$
  $v_{i-n} := -a_i$
**for** $j \in [0, n)$ **do**
  **for** $i \in [0, n)$ **do**
    **if** $c_i = 1$ **then**
      $w_j := w_j + v_{j-i}$
    **if** $c_i = -1$ **then**
      $w_j := w_j - v_{j-i}$
**for** $i \in [0, n)$ **do**
  $u_i := w_i \pmod q$
**return** $u = \sum_{i=0}^{n-1} u_i \cdot x^i$

that are numerically inferior to $q$ are retained as polynomial coefficients. The derived matrix is interpreted within the NTT domain to enable fast polynomial multiplication.

- ExpandMask: This function employs a similar principle to ExpandA but a different SHAKE-256 to generate a random polynomial vector $\mathbf{y} \in S^{\ell}_{\gamma_1}$. Since the bound $\gamma_1$ is a power of 2, the rejection will not happen and all coefficients are sampled one by one. The input is the concatenation of $\rho'$ and a nonce $\kappa$ that introduces randomness to each round, and the output coefficients are generated by taking $2\gamma_1$ bits as a positive integer and then subtracting $\gamma_1 - 1$ as each result.
- Compute_w: This process computes the inner-product of the matrix $\mathbf{A}$ and the vector $\mathbf{y}$ to deriving $\mathbf{w} \in R^k_q$, and decomposes to the high-order and low-order vectors $\mathbf{w}_1$ and $\mathbf{w}_0$. Moreover, a serialization process is invoked on $\mathbf{w}_1$ in anticipation of the subsequent hashing stage.
- SampleInBall: This process commences with employing SHAKE-256 to absorb $\mu||\mathbf{w}_1$ to yield $\tilde{c}$, and then re-input to SHAKE-256 to obtain a random stream to generate $\boldsymbol{c}$, which has $\tau$ nonzero coefficients that equals 1 or $-1$. The initial $\tau$ bits serve as sign determinants, and the rejection sampling is invoked to synthesize $\tau$ distinct positions within the domain of $[0, n)$ for nonzero values.
- Reject: This process involves $sk$-related arithmetic computations and checks the resultant signature to ensure correct verification and avoid disclosure of secret information. It requires to compute $\mathbf{z} := \mathbf{y} + \boldsymbol{c} \cdot \mathbf{s}_1$, $\mathbf{r}_0 := \mathsf{LowBits}_q(\mathbf{w} - \boldsymbol{c} \cdot \mathbf{s}_2)$, 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 also have bound on the hamming weight. Any deviation from these stipulations results in an abortive process and the commencement of a new iteration.

**Potential for Acceleration.** Despite intensive research, the acceleration of the ML-DSA scheme is still possible, especially in some specific scenarios. Below we discuss the potential of acceleration.

1) *Server-oriented design.* In scenarios where a server operates as the subject, two primary strategies emerge to optimize computational efficiency amidst a high volume of 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 polynomial multiplication techniques.* During the Reject procedure, one can capitalize on the characteristic that $\boldsymbol{c}$ is a sparse ternary polynomial. Opting for a general NTT approach might overlook this property, leading to inefficiencies. Leveraging specialized multiplication techniques tailored for sparse polynomials can not only diminish computational complexity but also eliminate the need for domain conversions.
3) *Earlier rejection.* In the traditional execution of ML-DSA, a comprehensive polynomial evaluation result must be computed before any verification can commence. By refining the computational pattern, certain coefficients can be determined in advance, enabling earlier inspection of checkpoints 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.

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*

Within the ML-DSA scheme, one often encounters computational waste when the norms of the resulting values exceed the set bounds. Conventional methods like NTT-based [4], [6] or PSPM-based polynomial multiplication [13] also grapple with this inefficiency. The applied width-prior approach mandates a full polynomial arithmetic operation evaluation before any bounds check can be performed. Motivated by the prospect of obtaining coefficient values to be checked more rapidly, we delve into 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 foundation of sparse ternary polynomial multiplication detailed in Algorithm 1, we introduce a depth-prior method depicted in Algorithm 4. The original methodology computes the accumulation operation on $w_j$ based on $c_i$

value, requiring all $w_j$ values to be determined. In contrast, our approach commences from the $w_j$ index. For every $w_j$, the polynomial $\boldsymbol{c}$ is traversed, accumulating to $w_j$ according to each $c_i$ value. Consequently, some $w_j$ values is obtained more swiftly, facilitating earlier checks, which in turn minimizes potential unnecessary operations.

**Vertical Parallelism.** Given the bounded coefficient values of the input polynomial $\boldsymbol{a}$ and a constant $\tau$, each $w_j$ possesses an upper limit linearly related to the coefficient bounds of $\boldsymbol{a}$. If the infinity norm is relatively compact, especially when compared to machine word size, we can employ packing techniques. This enables the bundling of multiple polynomials into a singular machine word unit, facilitating 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:

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

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:

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

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, $w_j$ will conform to either 32-bit or 64-bit dimensions, allowing for streamlined representation in software. After accumulation, we subtract the extraneous $\tau U$, 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, which involves the simultaneous computation of multiple $v_i$ values. This form of parallelism is similar to NTT, wherein 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.

**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.

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

- *Flexibility*: DPSTPM's parallelism boasts inherent adaptability due to its minimal constraints on parallelism degree. Conversely, NTT entails stringent requirements relating to 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*: On several lightweight platforms, multiplication operations can be burdensome and potentially variable in execution time, introducing a risk of side-channel attacks. Our PSTPM eliminates multiplication operations, relying solely on lightweight addition, subtraction, and bit-shifting. Additionally, branching operations solely pertain to the public polynomial $\boldsymbol{c}$, sidestepping energy analysis concerns.

### C. Methods for Branch Elimination

In the aforementioned algorithm, one remaining concern pertains to 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 but not limited to loop unrolling and constant folding. Additionally, the unpredictability introduced by these statements disrupts the optimal alignment of pipeline scheduling due to the indeterminacy of the instructions. Further complications arise within multi-threaded parallel systems, where differing execution paths can culminate in challenges such as thread divergence. Given the aggregation of these factors, there is a consequent degradation in performance and an impediment to achieving optimal execution, thus highlighting the imperative to devise strategies that effectively address and 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 categorically

**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**
    $u_j^{(r-1-i)} := (t \bmod M) - \tau U (\mathrm{mod} q)$
    $t := \lfloor t/M \rfloor$
**return** $\mathbf{u} = [u^{(0)}, \cdots, u^{(r-1)}]^T$

delineated into two predominant pathways, contingent upon the disparate 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 statement 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 advocate for an alteration in the structure of $\boldsymbol{c}$. Instead of preserving the complete $\boldsymbol{c}$ during sampling, we only conserve the respective $\tau$ 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 enables the unification of the accumulation phase expression as $w_j := w_j + v_{j+cp_i}$, successfully obviating the need for branching. A detailed representation of this refined approach is delineated in Algorithm 6.

### D. Rejection-Prioritized Norm Checking Order

In the signing procedure of Dilithium four distinct conditions must be met to ensure a valid signature, as delineated 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 adhering to these bounds are $e^{-256\cdot\beta l/\gamma_1}$ and $e^{-256\cdot\beta k/\gamma_2}$, respectively. Given that the cumulative probability of both conditions being met is $e^{-256\cdot\beta(l/\gamma_1 + k/\gamma_2)}$, it becomes evident, based on the parameter values in Table I, that these two conditions primarily dictate the restarts during the rejection process, rather than the conditions encompassing $||\boldsymbol{c} \cdot \mathbf{t}_0||_\infty$ and hints. Furthermore, since $k/\gamma_2$ typically surpasses $l/\gamma_1$, it can be inferred that $\mathbf{r}_0$ is more susceptible to rejection than $\mathbf{z}$. Drawing from this analysis, we advocate for a rejection-prioritized norm checking sequence. Within the PSTPM computation, we commence with the calculation of $\boldsymbol{c} \cdot \mathbf{s}_2$, 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 conditions with elevated rejection probabilities to be checked at the outset, effectively curtailing redundant execution procedures.

## 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 pivots around the proposed optimized signing of ML-DSA. In contrast to [23], which employs 32 threads, and [11], which alternates between 32 and 128 threads, our approach dedicates a block of 128 threads to each task. This allocation confers significant benefits, including optimized memory access, minimized IO latency, and a potential SM occupancy of 100%. This stands in stark contrast to the 33.3% theoretical occupancy achieved with just 32 threads. Adhering to the operation decomposition delineated in Sec III-A, we implement the associated kernels. Furthermore, we incorporate a Pack kernel to facilitate the preprocessing of secret key components. As our focus is on a server-oriented architecture, only the message-centric kernels remain online, relegating the ExpandA and Pack kernels to offline operations.

To handle multiple signing tasks, we incorporate batch processing within the implemented kernels, facilitating simultaneous task execution. Concurrently, we adopt the memory pool design in [11], ensuring both secure and efficient memory access. We also harness the dynamic scheduling mechanism delineated in [11] to guarantee optimal hardware resource allocation. This mechanism addresses the potential decline in occupancy due to varying repetitions across different signing tasks. With this, we ensure consistent high occupancy and maximize hardware utilization of our implementation.

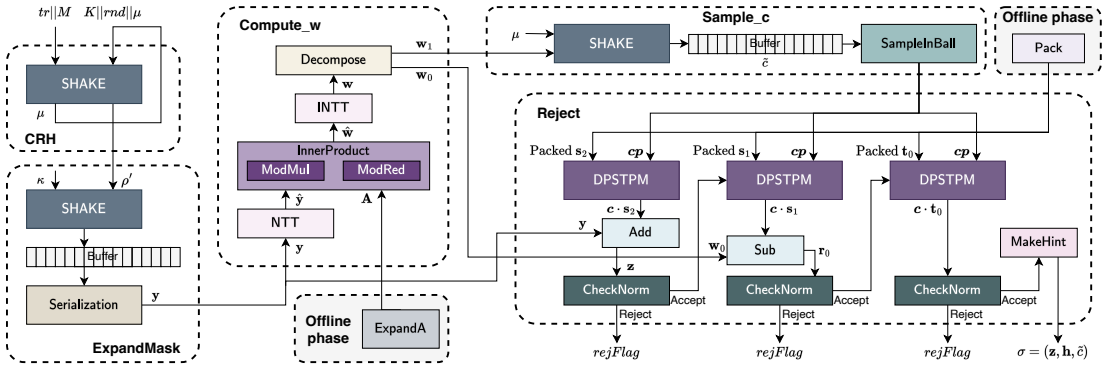



Fig. 2. Architectural design of the ML-DSA signing procedure, segmented by the implemented kernels and comprised of the associated inline functions.

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 |

## B. Implementation Details

In pursuit of a coherent and streamlined design, we deconstruct the six stages in ML-DSA signing to extract and identify their underlying arithmetic structures. The detailed breakdown of each resultant step and their encompassing functions is presented in Table II. Stemming from this analytical deconstruction, we have subsequently derived a comprehensive structural design, as illustrated in Fig. 2.

Below we delineate our GPU implementation pertaining to the inline functions. Some details of the pseudocode are presented in Algorithm 7. The inline functions can be stratified into two distinct categories. The first category comprises functions that operate singularly on one operand, eschewing requisite for data or thread interaction. Every thread processes a single element, invoking the functions to derive the outcomes. The functions within this category include:

- ModRed: We employ a variant of Barrett Reduction [6], [24] for modular reduction, where the division is supplanted by more efficient multiplication and bit-shifting. Contrary to the original method [24], we compute $t$ in a more effective strategy using only addition and bit-shifting, with stricter bounds on both the input and output.
- MontMul: We leverage Montgomery Reduction [25] to implement a modular multiplication. This function accepts two integers within the Montgomery domain and yields a result in the range $(-q, q)$. Given that the product of two 32-bit integers surpasses the register size, we resort to CUDA PTX assembly instructions to optimize register usage and minimize the instruction count.
- Decompose: Given that this function encompasses two cases corresponding to distinct values of $\gamma_2$, we employ macro definitions to encapsulate these cases. The computation of the high-order and low-order elements adheres to the definitions in [6]. Results are derived using a approximately equal method that ensures constant-time execution and cost-efficiency.
- CheckNorm: The function evaluates the value of $a$ relative to the threshold $B$. Given the potential negativity, we derive $|a|$ in constant time using the sign bit. By masking $2a$ with the sign bit and subtracting it from $a$, we achieve the absolute value. The comparison result is discerned from the sign bit following subtraction from $B$.
- MakeHint: This function takes the pair $(a_1, a_0)$ as input, evaluates three specified conditions, and subsequently yields the corresponding 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.

**Algorithm 7** Details of the implementation for the first group 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` $a_h, x, \zeta$
  `mul.lo.s32` $a_l, x, \zeta$ ▷ $a \leftarrow x \cdot \zeta$
  `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

TABLE III
PARAMETER CONFIGURATION FOR THE DPSTPM IMPLEMENTATION.

| Scheme | Operation | $\tau$ | $U$ | $2\tau U$ | $M$ | $r$ |
|---|---|---|---|---|---|---|
| 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 |

- NTT: Contrary to [11] that employs both radix-2 and radix-8 approaches, we exclusively leverage radix-2 utilizing 128 threads. In our implementation, temporary values are stored in SMEM while twiddle factors are cached in constant memory. Each thread processes two elements separated by a distance of $2^{8-i}$ at level $i$, where $i \in [1, 8]$. To circumvent the performance degradation due to bank conflicts arising from stride SMEM accesses, we strategically pad the SMEM bank units, ensuring conflict-free access within identical read or write cycles.
- DPSTPM: In our implementation, we focus on the evaluation phase of the unified DPSTPM. Accepting the packed vector element $v$ and position array $\boldsymbol{cp}$ as inputs, the function yields evaluation results. We deploy 128 threads and accomplish the evaluation in two distinct rounds. During the $k$th round, the $j$th thread manages the element $w_{128k+j}$ and iteratively conducts a $\tau$-loop, sequentially adding the element $v_{128k+j+cp_i}$ 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, resulting in coalesced memory access, which maximizes access speed.

### C. Optimized Reject Kernel

Throughout the entire reject process, our implemented kernel receives the challenge $\boldsymbol{cp}$, the secret key elements, and other pertinent polynomials as inputs, conducts arithmetic operations over the ring $R$, and checks the norms to derive a valid signature. Within this implementation, we incorporate three previously discussed optimizations. Initially, we employ the DPSTPM algorithm illustrated in Section III-B. Subsequently, we introduce a merging technique that leverages the early evaluation approach. Finally, we embrace an order that facilitates easier rejection by prioritizing the norm-checking of $\mathbf{r}_0$. The conventional signing process demands numerous NTT and INTT computations. Transitioning to the PSTPM approach reduces the number of multiplications and facilitates the concurrent 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 that merges DPSTPM with early evaluation, aimed at expediting the rejection of invalid signatures, which is compatible with all three evaluation processes. The underlying principle is splitting the evaluation phase into two rounds. Upon the completion of each round, a sequential unpacking is executed to derive each half results. This is followed by arithmetic operations and a subsequent assessment to determine if the norms exceed their bounds. 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. By our design parameters, tid $\in [0, 128)$ and lid $\in [0, 32)$. The procedure commences with generating a table base on $\mathbf{s}_2$ for PSTPM evaluation. Each thread subsequently undertakes the

**Algorithm 8** GPU implementation of depth-prior PSTPM 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
$s\_cp \leftarrow cp$, $s\_f := 0$, $reg := 0$
$tmp := pack\_s_2$[tid]
$s\_table$[tid $+ N$] $:= tmp$ ▷ Prepare PSTPM table
$s\_table$[tid $+ N + 128$] $:= tmp$
$s\_table$[tid], $s\_table$[tid $+ 2N$] $:= Mask\_s_2 - tmp$
$s\_table$[tid $+ 128$], $s\_table$[tid $+ 2N + 128$] $:= Mask\_s_2 - tmp$
**for** $j \in [0, \tau)$ **do** ▷ First round DPSTPM evaluation
  idx := tid $+ s\_cp[j]$
  $reg = reg + s\_table$[idx]
__syncthreads()
**for** $i \in [0, k)$ **do**
  $res := (reg \& Mask\_P) - \tau \cdot \eta$
  $reg := reg \gg Bit\_P$
  $res := \mathbf{w}_0[i]$[tid] $- res \bmod q$
  $rejFlag :=$ __any_sync(CheckNorm$(\gamma_2 - \beta, res)$)
  **if** lid $= 0$ *and* $rejFlag = 1$ **then**
    $s\_f := 1$
  __syncthreads()
  **if** $s\_f$ **then return**
  $s\_s_2[i]$[tid] $:= res$ ▷ Write results to shared memory
**for** $j \in [0, \tau)$ **do** ▷ Second round DPSTPM evaluation
  idx := tid $+ s\_cp[j]$
  $reg = reg + s\_table$[idx $+ 128$]
__syncthreads()
**for** $i \in [0, k)$ **do**
  $res := (reg \& Mask\_P) - \tau \cdot \eta$
  $reg := reg \gg Bit\_P$
  $res := \mathbf{w}_0[i]$[tid $+ 128$] $- res \bmod q$
  $rejFlag :=$ __any_sync(CheckNorm$(\gamma_2 - \beta, res)$)
  **if** lid $= 0$ *and* $rejFlag = 1$ **then**
    $s\_f := 1$
  __syncthreads()
  **if** $s\_f$ **then return**
  $s\_s_2[i]$[tid $+ 128$] $:= res$ ▷ Write results to shared memory

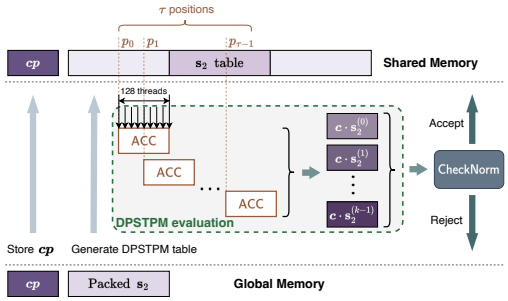



Fig. 3. Optimized memory sccess pattern and processing logic flow for the DPSTPM evaluation of $\boldsymbol{c}\cdot\mathbf{s}_2$.

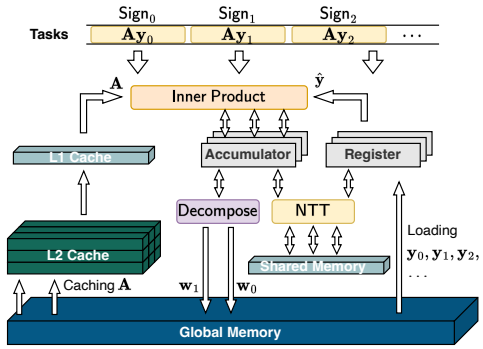



Fig. 4. Layered Caching Strategy for the computation of $\mathbf{A} \cdot \mathbf{y}$.

$\tau$-loop for a singular round of evaluation, and then unpacks the coefficient of the $i$th polynomial and subtract it from $\mathbf{w}_0[i][\mathsf{tid}]$ to check the norm. Finally, a synchronization of the thread states is conducted to determine if rejection is required, subsequently updating the rejection flag.

**Optimized Memory Access Pattern.** To achieve low latency data access across all evaluation processes, we allocate the array $\boldsymbol{cp}$ and the three DPSTPM tables to the SMEM, while frequently accessed accumulation results are designated to the RF. In contrast to the other evaluations, the result of $\boldsymbol{c}\cdot\mathbf{s}_2$ need to be temporarily stored for subsequent computations. Consequently, we transfer the it from the RF to the SMEM to maintain an equilibrium. This process is visually elucidated in Fig. 3. Such an approach presents dual advantages. Firstly, it conserves registers for other computations, thereby mitigating the risk of register overuse, which diminish kernel occupancy. Secondly, it ensures relatively fast memory access speeds for both the writing and retrieval of $\boldsymbol{c} \cdot \mathbf{s}_2$.

### D. Caching Key Component

In the basic design of the Compute_w kernel, a block comprising 128 threads is dedicated to manage computations of a signing task. For the calculations that involve the inner-product of matrix $\mathbf{A}$ and vector $\mathbf{y}$, each thread loads two coefficients of both entities from the GMEM, subsequently storing them in the block's specific memory segment. The thread then undertakes the multiplication task, accumulating the result accordingly, thereby accomplishing the computation. Notably, when the server functions as an entity and the key remains invariant, the matrix $\mathbf{A}$ is subject to consistent reloading across all active blocks. In the architecture of [11], the memory segment dedicated to each signing task in the memory pool reserves space for $\mathbf{A}$, prompting each signing task to load it autonomously. Such an approach fails to proffer computational advantages during compilation and results in pronounced memory access overhead.

Leveraging caching for the $\mathbf{A}$ matrix presents a potential solution to the issue. However, the substantial memory demands of $\mathbf{A}$ render the SMEM inadequate for storing it, particularly for larger parameter sets such as ML-DSA-65 and ML-DSA-87. Since excessive SMEM allocation can detrimentally affect SM occupancy and overall performance, we separate $\mathbf{A}$ from the memory segments allocated to tasks and allocate it within a discrete region in the GMEM. All blocks then access this matrix from this location. 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. 4. In this architecture, $\mathbf{A}$ adheres to the memory hierarchy, being cached in a stratified manner, transitioning from GMEM to L2 and then to L1. Each block independently loads its associated $\mathbf{y}_i$ to the RF to expedite access, given its recurrent use during computations. Simultaneously, we allocate $2k$ registers within each thread as accumulators to retain the summative results. A temporary space within SMEM is designated for data interchange for NTT and INTT. Subsequent to the computation and through decomposition, we derive $\mathbf{w}_0$ and $\mathbf{w}_1$, which are subsequently transferred back to GMEM.

### E. Sparse Polynomial Sampler and Adaptations

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

In the first version, we follow the original sampling procedure, preserving the entire $\boldsymbol{c}$. The random stream is generated from $\tilde{c}$, wherein the initial 8 bytes function as sign determinants while the remainder facilitate position generation. A notable limitation of this approach is the necessity to traverse $\boldsymbol{c}$ to obtain the $\tau$ positions in DPSTPM. Consequently, in our secondary variant, we directly archive the positions $\boldsymbol{cp}$ during the sampling mechanism, eschewing the retention of $\boldsymbol{c}$. Here, we establish $cp_j := -i$ when $c_i = 1$ and $cp_j := n - i$ when $c_i = -1$. Nonetheless, for both aforementioned versions, the random nature of the data inhibits parallel computations through multiple threads, owing to potential data conflicts and read/write competition.

Therefore, we introduce a third version that allow parallel computations. While positions can be generated in a conventional sequential manner, they can also be obtained concurrently. We construct a Boolean lookup table to meticulously monitor pre-existing positions, thereby ensuring the generation of $\tau$ distinct positions. This method eliminates the need for exhaustive comparisons or traversal processes, which is more efficient and can serve as a complementary solution given its intrinsic alterations in test vectors.

## V. Performance Evaluation

### A. Experimental Setup

The C/C++ source code is compiled utilizing g++ 12.2.0, whereas GPU implementation is compiled using CUDA 11.8. All compilations and executions are conducted on an Arch Linux system running kernel 5.15. For CPU benchmarks, we use an Intel(R) Core(TM) i9-12900KS CPU endowed with 16 cores. The performance evaluations are performed on a NVIDIA Tesla A100 80G PCIe and a NVIDIA GeForce RTX 4090. This provides a comprehensive spectrum of computational capabilities to bolster the robustness of our analysis. In the experiments, we batch processing 10,000 computational tasks, and the execution time is recorded and presented in microseconds ($\mu s$). For the overall efficiency, throughput is

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 ($\mu s$) | | 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%** |

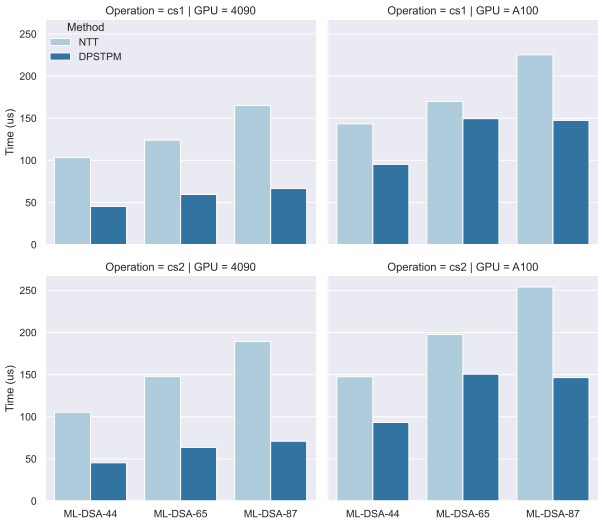



Fig. 5. Performance analysis of batch processing 10,000 executions for $\boldsymbol{c} \cdot \mathbf{s}_1$ and $\boldsymbol{c} \cdot \mathbf{s}_2$. Vertical parallelism values are $r = 4, 5, 7$ for $\boldsymbol{c} \cdot \mathbf{s}_1$, and $r = 4, 6, 8$ for $\boldsymbol{c} \cdot \mathbf{s}_2$, across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

TABLE V
EXECUTION TIME OF THREE SPARSE POLYNOMIAL SAMPLERS (MEASURED IN MICROSECONDS ($\mu s$)).

| 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 |

represented as operations per second (OP/s), elucidating the number of signatures successfully finalized within a second.

### B. Evaluation of Optimization Effectiveness

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

**Comparison of DPSTPM and NTT.** Given that the evaluation of $\boldsymbol{c} \cdot \mathbf{s}_1$ and $\boldsymbol{c} \cdot \mathbf{s}_2$ extensively harnesses our proposed parallel technique, with vertical parallelism denoted by $r = k$ or $l$ for the respective cases, we depict a comparative execution time between the NTT approach and our DPSTPM for these processes in Fig. 5, which are representative in the signing procedure. 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 to obtain the execution time. Our DPSTPM consistently surpasses the original NTT method, showcasing execution speed enhancements ranging from 52.0% to 59.7% for $\boldsymbol{c} \cdot \mathbf{s}_1$ and between 56.9% and 62.5% for $\boldsymbol{c} \cdot \mathbf{s}_2$. The A100 GPU reveals a marginally diminished acceleration effect for ML-DSA-65, primarily attributable to parameter-induced influences on hardware task scheduling. For the remaining parameter sets, the performance enhancement persists, lying in the vicinity of 33.6% to 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, it is discernible that the execution time for $\boldsymbol{c} \cdot \mathbf{s}_2$ is nominally more prolonged than that for $\boldsymbol{c} \cdot \mathbf{s}_1$, primarily by approximately 4 $\mu s$ on the 4090 GPU. In contrast to the NTT, the main performance dominant for the DPSTPM is the variable $\tau$, which affect both the computational complexity and the entire computation.

**Speedups through Caching.** An examination of the kernel profiling metrics, pre and post-optimization, for the computation $(\mathbf{w}_1, \mathbf{w}_0) = \mathbf{A} \cdot \mathbf{y}$ is tabulated in Table IV. The enhancement is largely attributed to our refined memory configuration and access strategies. Our approach leverages varying classes of on-chip memory commensurate with specific access properties, leading to a substantial increase in both L1 and L2 cache hit rates by 2876.22% and 102.31%, respectively. As a consequence, memory accesses witness heightened efficiency, leading to reduced execution time, optimized processor resource usage, and superior pipeline scheduling. Given that our method predominantly harnesses addition and logical operations, there is an increase in the FMA and ALU pipeline utilization. For the ML-DSA-44 parameter set, the execution time is reduced from 69.86 $\mu s$ to 56.64 $\mu s$, indicating a speedup factor of 1.23×. Similar enhancements are evident for the ML-DSA-65 and ML-DSA-87 configurations, showing speedups of 1.30× and 1.43×, respectively.

**Performance of Sampler Variants.** In Table V, we delineate the execution time associated with three sparse polynomial samplers, benchmarked on the 4090 and A100 GPU platforms. Specifically, sampling method I corresponds to 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

TABLE VI
THROUGHPUT COMPARISON BETWEEN C, AVX2 IMPLEMENTATIONS, THE GPU APPROACH OF [11], AND OUR GPU 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 | A100 | | | 4090 | | |
| | | | Streaming | Streaming | Batching | Streaming | Speedup | Batching | Streaming | Speedup |
| ML-DSA-44 (Dilithium2) | 5,182 | 23,678 | 765,855 | 984,803 | 771,943 | 884,683 | **170.7×** **+15.5%** | 1,049,238 | 1,080,871 | **208.6×** **+9.8%** |
| ML-DSA-65 (Dilithium3) | 3,396 | 15,415 | 513,468 | 649,498 | 538,742 | 615,811 | **181.3×** **+19.9%** | 736,109 | 884,195 | **260.4×** **+36.1%** |
| ML-DSA-87 (Dilithium5) | 2,669 | 12,728 | 396,894 | 488,006 | 448,515 | 540,285 | **202.4×** **+36.1%** | 554,651 | 785,277 | **294.2×** **+60.9%** |

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 |

and competition. A comparative analysis across both GPU platforms reveals that I marginally outpaces II, though III consistently demonstrates superior performance. However, the method I necessitates an additional step to generate $\boldsymbol{cp}$ in subsequent kernels. This particularity accentuates the superior compatibility of method II within our implemented framework.

### C. Overall Performance

Table VI encapsulates the throughput results of the basic software implementation, and the GPU implementations from both the related work of Dilithium [11] and our solutions. Given the absence of an officially open-source implementation of ML-DSA, we utilize the open-source version of Dilithium[1] to serve as our CPU baseline. This repository contains both a C reference and an AVX2-optimized implementation. Notably, the primary difference is a minor adjustment in the length of some seeds, exerting a minimal influence on overall performance. The study [11] represents the most recent GPU implementation of Dilithium, and achieves best performance. For a comprehensive assessment, we executed the corresponding performance comparison on identical GPU platforms.

For the performance evaluation, we provide two distinct methodologies include batching and streaming, where we batch process 10,000 signing tasks in the first method and initiate 10 CUDA streams with each processing 1,000 signing tasks in the second method. The results show a consistent superior performance of the streaming method over batching, predominantly attributed to its advantage in hiding memory transfer latency. On the 4090 GPU, the streaming approach manifests a 41.6% acceleration relative to the batching approach for the ML-DSA-87. Similarly, on the A100 GPU, it consistently surpasses the batching technique, displaying enhancements ranging from 14.3% to 20.5%.

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

Utilizing the server-caliber A100 GPU, our framework exhibits speedups of 170.7×, 181.3×, and 202.4× for the three ML-DSA parameter sets against the C baseline. Analogously, employing the commercially oriented 4090 GPU, our solution achieves accelerations of 208.6×, 260.4×, and 294.2×, respectively. In juxtaposition with the AVX2-optimized methodology, we achieve a speedup ranging from 37.4× to 42.4× on the A100 GPU and 45.6× to 61.7× on the 4090 GPU. Although CPU designs can harness performance gains via multi-threaded executions, GPU architectures display superior adeptness in concurrently processing computational tasks. 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. Such marked improvements are ascribable to our refined signing procedure amalgamated with our server-attuned architectural considerations.

Table VII delineates the throughput results derived from several related studies [7], [8], [10], [12] spanning diverse computational platforms. The research elucidated in [10] is calibrated for 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] focus on FPGA-centric hardware designs. While there is potential to enhance throughput by allocating more hardware area, in server-grade scenarios emphasizing high-performance and real-time solutions, GPU implementations often stand out as a preferred choice.

## VI. CONCLUSION

In this work, we address a significant gap in GPU-optimized implementations for ML-DSA in server environments. By introducing a tailored, server-centric design enhanced with novel theoretical optimizations, we have achieved substantial performance gains. Demonstrated speedups against both CPU benchmarks and existing methods emphasize 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.

## 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. [Online]. Available: https://doi.org/10.1137/S0097539795293172

[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. [Online]. Available: https://doi.org/10.13154/tches.v2018.i1.238-268

[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," pp. 1–10, 2021. [Online]. Available: https://doi.org/10.1109/ICFPT52863.2021.9609917

[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. [Online]. Available: https://doi.org/10.46586/tches.v2022.i1.270-295

[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," pp. 1:1–1:11, 2021. [Online]. Available: https://doi.org/10.1145/3465481.3465756

[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. [Online]. Available: https://doi.org/10.46586/tches.v2022.i1.221-244

[11] S. Shen, H. Yang, W. Dai, Z. Liu, and Y. Zhao, "High-throughput gpu implementation of dilithium post-quantum digital signature," arXiv preprint arXiv: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. [Online]. Available: https://doi.org/10.1016/j.icte.2022.08.003

[13] J. Zheng, F. He, S. Shen, C. Xue, and Y. Zhao, "Parallel small polynomial multiplication for dilithium: A faster design and implementation," in Proceedings of the 38th Annual Computer Security Applications Conference, 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. [Online]. Available: https://doi.org/10.46586/tches.v2021.i1.1-24

[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. [Online]. Available: https://doi.org/10.1109/TPDS.2020.3025691

[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. [Online]. Available: https://doi.org/10.1109/ISCAS51556.2021.9401575

[18] L. Wan, F. Zheng, and J. Lin, TESLAC: Accelerating Lattice-Based Cryptography with AI Accelerator, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 2021, book section Chapter 13, 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. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3097277

[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. [Online]. Available: https://doi.org/10.1109/ACCESS.2022.3152217

[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. [Online]. Available: https://doi.org/10.1007/978-3-031-17143-7_25

[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. [Online]. Available: https://doi.org/10.1109/TPDS.2020.2995562

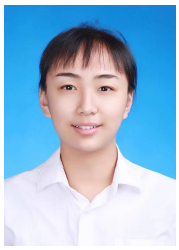
[23] S. C. Seo and S. W. An, "Parallel implementation of crystals-dilithium for effective signing and verification in autonomous driving environment," ICT Express, 2022.

[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** is a PhD candidate in School of Computer Science, Fudan university. Her research interests include lattice-based cryptography, homomorphic encryption, and cryptographic engineering. Her email address is shenshiyu21@m.fudan.edu.cn.

**Hao Yang** is a PhD candidate at College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics. 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.