SpreadMAYO: Unlocking Greater Parallelism for Faster MAYO Signatures on GPUs

Zhuoyu Xie, Tian Zhou, Fangyu Zheng, *Member, IEEE,* Yijing Ning, Yi Bian, Wangchen Dai, Jiankuo Dong, Yuewu Wang, Jingqiang Lin, *Senior Member, IEEE,* and Jiwu Jing

***Abstract*—Quantum computing threatens public-key cryptography, motivating post-quantum signature standardization. MAYO, a multivariate Round 3 candidate in NIST's additional signature process, offers compact keys and signatures. However, prior GPU implementations do not fully exploit parallelism within each MAYO operation: equation-slice partitioning serializes independent matrix work, while signing requires frequent synchronization among threads during linear-system construction and elimination. We present SpreadMAYO, a high-throughput GPU implementation of MAYO Round 3 that exposes greater parallelism and reduces synchronization through finer task partitioning and a revised signing pipeline. SpreadMAYO distributes matrix contractions across rows, output positions, and 64-bit limbs while reusing operands in registers. Each limb packs sixteen $\mathrm{GF}(16)$ coefficients; native 32-bit operations process its halves without lookup tables. For signing, threads privately accumulate columns with deferred polynomial reduction, transpose packed columns into rows using warp shuffles, and eliminate rows below each pivot in parallel. Additionally, we employ a hybrid AES design to accommodate different workloads and fuse key expansion with matrix computations to reduce global-memory traffic. On an NVIDIA GeForce RTX 4090, SpreadMAYO outperforms the Round 3 adapted GPU baseline by 2.33–3.71× in key generation, 4.40–6.93× in signing, and 3.42–5.13× in verification, reaching up to 10.75× in repeated-key server signing.**

***Index Terms*—Digital signatures, finite-field arithmetic, GPU acceleration, MAYO, post-quantum cryptography.**

I. Introduction

Quantum computing threatens RSA and elliptic-curve signatures through algorithms for factoring and discrete logarithms [1]. In August 2024, NIST finalized ML-KEM for key encapsulation and ML-DSA and SLH-DSA for digital signatures [2]–[4]. Its additional signature process, initiated in 2022, seeks to diversify security assumptions beyond structured lattices and support short signatures and fast verification [5].

MAYO is a multivariate signature scheme based on Oil and Vinegar (OV) maps, with security relying on the OV and multi-target whipped MQ problems [6], [7]. Its whipping construction combines a smaller underlying quadratic map and oil space to provide compact public keys and signatures. MAYO advanced to the third round of NIST's Additional Digital Signature standardization process in May 2026 [8]; throughout this paper, MAYO refers to the Round 3 specification.

Large authentication and signing services require high aggregate throughput, motivating GPU acceleration. MAYO's matrix products and linear systems over $\mathbb{F}_{16}$ expose parallel work across equations, matrix rows, and output positions. Efficient execution balances operand reuse with the registers and shared memory needed to run many instances concurrently.

A. Related Work

Prior MAYO implementations explore different field representations and computation strategies on CPUs and dedicated hardware. Beullens et al. [9] optimize AVX2 and Cortex-M4 implementations using nibble packing and matrix multiplication techniques. Samyuktha et al. [10] adopt a bitsliced representation in Jasmin, while Lee et al. [11] develop a Round 3 NEON implementation with interleaved AES expansion and a transposed Gaussian elimination solver. Hirner et al. [12] investigate FPGA and ASIC designs using specialized memory organization and data transposition. These studies establish the importance of field representation and matrix organization. GPU implementations must also distribute work across threads under register and shared memory constraints.

Prior GPU studies accelerate lattice-based schemes [13]–[18], code-based schemes [19], [20], and hash-based signatures [21], [22]. GPU implementations of homomorphic cryptography optimize arithmetic [23]–[25] and exploit on-chip tiling and memory reuse [26], [27]. These workloads differ from MAYO's matrix operations. Choi et al. [28] implement MAYO on GPUs and analyze its bottlenecks.

Chen et al. [29] subsequently propose Reuse Matrix Multiplication Evaluation (RMME), which reorders matrix contractions to reuse loaded operands within threads and reduce redundant global memory reads. Their design combines RMME with Horner evaluation, kernel fusion, and asynchronous execution across CUDA streams. KeyGen is fused into one kernel, whereas Sign and Verify separate key expansion from the main computation. The main computation kernels use one warp per instance and partition matrix work primarily across equation byte slices, leaving matrix rows and output positions inside sequential thread loops. Multiplication over GF(16) uses shared memory lookup tables, while Gaussian elimination updates target rows cooperatively with repeated warp synchronization. These choices improve operand reuse but leave opportunities to distribute matrix work more widely across threads and eliminate target rows in parallel.

Zhuoyu Xie and Tian Zhou contributed equally to this work. (Corresponding author: Fangyu Zheng.)

Zhuoyu Xie, Fangyu Zheng, Yi Bian, Yuewu Wang, and Jiwu Jing are with the School of Cryptology, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: zhengfangyu@ucas.ac.cn).

Tian Zhou, Yijing Ning, Jiankuo Dong, and Jingqiang Lin are with the School of Cyber Science and Technology, University of Science and Technology of China, Hefei 230027, China.

Wangchen Dai is with Sun Yat-sen University, Shenzhen 518107, China.

B. Design Challenges

MAYO exposes substantial parallelism across equations, matrix rows, and output positions. Exploiting this parallelism on GPUs requires balancing operand reuse, thread concurrency, and the use of registers and shared memory. Existing GPU implementations face three architectural challenges:

- Existing GPU implementations process two $\mathbb{F}_{16}$ coefficients per byte through nibble extraction, lookup tables in shared memory, and repacking [29]. This approach underutilizes native 32-bit integer arithmetic and incurs irregular shared memory accesses.
- Partitioning matrix contractions along the equation dimension exposes only $m/2$ tasks, each corresponding to a byte slice, while keeping matrix rows and output positions inside sequential thread loops [29]. Finer partitioning can increase concurrency but risks redundant operand loads and reduced data reuse.
- Constructing the signing system $Ax = y$ requires polynomial shifts $E_{ij}M_j$ along columns, whereas Gaussian elimination operates across matrix rows. Existing implementations reduce shifted contributions separately during construction and sequentially update target rows during elimination, requiring repeated polynomial reductions and synchronization [29].

C. Contributions

To address these challenges, we present SpreadMAYO, a high-throughput GPU implementation of MAYO Round 3 that jointly optimizes finite-field arithmetic, matrix parallelism, and kernel execution. Our contributions are as follows:

1) **Packed field arithmetic and fine-grained matrix partitioning.** We pack sixteen $\mathbb{F}_{16}$ coefficients into each 64-bit limb and multiply its two 32-bit halves independently by a common field scalar using native integer operations without lookup tables. We assign matrix rows, output positions, and equation limbs to each instance's 4-warp (128-thread) CTA while reusing operands in registers.
2) **Highly parallel linear system construction and solving.** Threads construct the signing matrix A by accumulating private columns in registers with deferred polynomial reduction, then transpose packed columns into rows using warp shuffles without intermediate memory staging. Gaussian elimination updates independent target rows in parallel, reducing the synchronization overhead of prior work [29].
3) **Memory-efficient data layout and customized symmetric primitives.** We introduce a padded 64-bit limb layout for aligned memory accesses and combine an 8-block barrel AES-128-CTR engine [30], [31] for large public matrices with a compact T-table AES engine in shared memory [29] for the smaller map Λ. We further optimize warp SHAKE256 by packing Keccak rotation offsets into register constants to eliminate stack spills.
4) **Fused kernel design for end-to-end acceleration.** We design fused 4-warp kernels that integrate key expansion with matrix computations and reuse storage across stages to reduce global memory traffic. Multi-stream execution supports batches with distinct keys, while repeated signing under one key reuses precomputed static state. On an NVIDIA GeForce RTX 4090, SpreadMAYO achieves overall end-to-end throughput speedups of 2.33–$3.71\times$ for key generation, 4.40–$6.93\times$ for signing, and 3.42–$5.13\times$ for verification over a Round 3 adaptation of Chen et al. [29]. With key expansion performed offline in both implementations, repeated signing under one key reaches up to $10.75\times$ over the corresponding baseline.

The remainder of this paper is organized as follows. Section II provides background. Sections III–V present the arithmetic, data layout, and fused kernel designs. Section VI evaluates performance, and Section VII concludes.

II. Background

This section introduces the finite-field arithmetic and notation, the core operations and computational structure of the MAYO signature scheme, and the GPU architectural mechanisms exploited in the subsequent sections.

A. Notation and Parameters

MAYO operates over the binary finite field $\mathbb{F}_{16}$ of 16 elements, defined as the polynomial quotient ring $\mathbb{Z}_2[x]/(x^4+x+1)$ [7]. Each element of $\mathbb{F}_{16}$ occupies a 4-bit nibble representing $\alpha_3x^3+\alpha_2x^2+\alpha_1x+\alpha_0$ with $\alpha_h \in \{0,1\}$. Because $\mathbb{F}_{16}$ has characteristic two, addition and subtraction are identical and reduce to bitwise XOR, while multiplication corresponds to polynomial multiplication modulo x^4+x+1.

We denote the set of bytes by $\mathbb{B}=\{0,\dots,255\}$, byte strings of length ℓ by $\mathbb{B}^\ell$, and concatenation by $\|$. Vector and matrix indices start at zero, $[a:b]$ denotes the half-open slice from a to b, and $[A \mid B]$ denotes horizontal concatenation. For $B \in \mathbb{F}_{16}^{d\times d}$, the operator $\mathrm{Upper}(B)$ preserves the diagonal $B[i,i]$, folds each lower-triangular entry into its symmetric counterpart by setting $\mathrm{Upper}(B)[i,j]=B[i,j]+B[j,i]$ for $i<j$, and zeros all entries below the diagonal. Over $\mathbb{F}_{16}$, any $w \in \mathbb{F}_{16}^d$ satisfies $w^TBw = w^T\,\mathrm{Upper}(B)w$, allowing quadratic forms to be stored and evaluated in upper-triangular form with roughly half the coefficients.

Table I summarizes the four Round 3 parameter sets targeting NIST security levels 1, 3, and 5 [7]: n is the number of variables per quadratic polynomial, m is the number of polynomials in the public map, o is the dimension of the secret oil subspace, and k is the whipping factor. They determine the vinegar dimension $v=n-o$ and total oil unknowns ko, with byte lengths $b_v=\lceil v/2\rceil$ and $b_r=\lceil ko/2\rceil$ at two nibbles/byte.

B. The MAYO Signature Scheme

1) MAYO Overview: MAYO is a multivariate signature scheme built upon the Unbalanced Oil and Vinegar (UOV) trapdoor [6], [32]. Its public map $\mathcal{P}(w) = (w^TP_0w,\dots,w^TP_{m-1}w)$ consists of m upper-triangular quadratic forms $P_a \in \mathbb{F}_{16}^{n\times n}$ whose n variables are partitioned

TABLE I
MAYO ROUND 3 PARAMETER SETS AND MATRIX DIMENSIONS.

Parameter	NIST level	n	m	o	k	$v = n - o$	ko
MAYO-1	1	88	80	8	10	80	80
MAYO-2	1	86	64	13	5	73	65
MAYO-3	3	118	108	10	11	108	110
MAYO-5	5	154	142	12	12	142	144

Algorithm 1 MAYO KeyGen

Output: Compact keys (csk, cpk)
$seed_{sk} \xleftarrow{\$} \mathbb{B}^{sk_seed_bytes}$
$(seed_{pk}, O) \leftarrow \mathrm{SHAKE256}(seed_{sk})$
$(P_1, P_2) \leftarrow \text{AES-128-CTR}(seed_{pk})$
for $a = 0, \ldots, m-1$ **do**
  $B_a \leftarrow P_{1,a}O + P_{2,a}$; $P_{3,a} \leftarrow \mathrm{Upper}(O^T B_a)$
$csk \leftarrow seed_{sk}$; $cpk \leftarrow seed_{pk} \parallel P_3$
return (csk, cpk)

into $v = n - o$ *vinegar* variables and o *oil* variables, constructed so that $\mathcal{P}$ vanishes identically on a secret o-dimensional *oil space* $\mathcal{O} \subset \mathbb{F}_{16}^n$. Whereas classical UOV requires $o \geq m$ and incurs large $\Theta(mo^2)$ public keys, MAYO selects a much smaller oil dimension $o \ll m$ and publicly *whips* $\mathcal{P}$ across k input vectors $s_0, \ldots, s_{k-1} \in \mathbb{F}_{16}^n$ to obtain $ko \geq m$ total oil unknowns. During signing, fixing random vinegar vectors cancels all quadratic oil terms via the trapdoor on $\mathcal{O}$ and reduces signature generation to solving an $m \times ko$ linear system over $\mathbb{F}_{16}$. Algorithms 1–3 summarize these operations; see [6], [7] for formal security proofs.

2) Key Generation: A secret matrix $O \in \mathbb{F}_{16}^{v \times o}$ expanded from $seed_{sk}$ defines the oil space $\mathcal{O} = \{(Ox) \parallel x \mid x \in \mathbb{F}_{16}^o\}$. Each public matrix $P_a \in \mathbb{F}_{16}^{n \times n}$ is partitioned into vinegar and oil blocks as $P_a = \left(\begin{smallmatrix} P_{1,a} & P_{2,a} \\ 0 & P_{3,a} \end{smallmatrix}\right)$, with upper-triangular $P_{1,a} \in \mathbb{F}_{16}^{v \times v}$, $P_{3,a} \in \mathbb{F}_{16}^{o \times o}$, and $P_{2,a} \in \mathbb{F}_{16}^{v \times o}$. Expanding $P_{1,a}$ and $P_{2,a}$ from $seed_{pk}$ via AES-128-CTR, KeyGen (Algorithm 1) enforces $w^T P_a w = 0$ on $\mathcal{O}$ via

$$B_a = P_{1,a}O + P_{2,a}, \qquad P_{3,a} = \mathrm{Upper}(O^T B_a). \tag{1}$$

The compact keys store $csk = seed_{sk}$ and $cpk = seed_{pk} \parallel P_3$. Public-key expansion (ExpandPK) regenerates P_1, P_2 from $seed_{pk}$ for verification and signing; secret-key expansion (ExpandSK) precomputes the bilinear matrices

$$L_a = (P_{1,a} + P_{1,a}^T)O + P_{2,a}. \tag{2}$$

Computationally, both key generation and secret-key expansion are dominated by AES-128-CTR expansion of (P_1, P_2) and batched matrix products with O across all m equations.

3) Verification: MAYO identifies each vector in $\mathbb{F}_{16}^m$ with a polynomial in $\mathbb{F}_{16}[z]/(f(z))$, where $f(z) = z^m + \sum_{d=0}^{3} c_d z^d$ has fixed coefficients $c_d \in \mathbb{F}_{16}$. Multiplication by z modulo $f(z)$ defines a linear operator $E \in \mathbb{F}_{16}^{m \times m}$ that shifts coefficients upward by one degree and folds the degree-m overflow into degrees $0, \ldots, 3$ via c_d. Writing $E_{ij} = E^{\ell(i,j)}$, the symmetric exponent table $\ell(i,j)$ maps the $k(k+1)/2$ unordered pairs (i,j) to distinct shift powers $0, \ldots, k(k+1)/2 - 1$ [7].

Algorithm 2 MAYO Sign

Input: Compact secret key csk, message M
Output: Signature sig or failure $\perp$
$(seed_{pk}, O) \leftarrow \mathrm{SHAKE256}(csk)$
$(P_1, P_2) \leftarrow \text{AES-128-CTR}(seed_{pk})$
$L_a \leftarrow (P_{1,a} + P_{1,a}^T)O + P_{2,a}$ $(0 \leq a < m)$ // ExpandSK
$d \leftarrow \mathrm{SHAKE256}(M, digest_bytes)$
$\rho \leftarrow 0^{R_bytes}$ or $\rho \xleftarrow{\$} \mathbb{B}^{R_bytes}$
$salt \leftarrow \mathrm{SHAKE256}(d \parallel \rho \parallel csk, salt_bytes)$
$(t, K_\Lambda, IV_\Lambda) \leftarrow \mathrm{SHAKE256}(d \parallel salt, m/2 + 32)$
$(\Lambda_v \mid \Lambda_o) \leftarrow \text{AES-128-CTR}(K_\Lambda, IV_\Lambda, nm/2)$
for $ctr = 0, \ldots, 255$ **do**
  $V_{\mathrm{bytes}} \leftarrow \mathrm{SHAKE256}(d \parallel salt \parallel csk \parallel ctr, kb_v + b_r)$
  $v_i \leftarrow V_{\mathrm{bytes}}[ib_v : (i+1)b_v]$ $(0 \leq i < k)$
  $r \leftarrow V_{\mathrm{bytes}}[kb_v : kb_v + b_r]$
  $M_i[a, :] \leftarrow v_i^T L_a$ $(0 \leq i < k,\ 0 \leq a < m)$
  $M_0 \leftarrow M_0 + \Lambda_v O + \Lambda_o$
  $A_i \leftarrow \sum_{j=0}^{k-1} E_{ij} M_j$ $(0 \leq i < k)$; $A \leftarrow [A_0 \mid \cdots \mid A_{k-1}]$
  $y \leftarrow t + \sum_{i \leq j} E_{ij} u_{ij} + \sum_i E_{i0} \Lambda_v v_i$
  $x \leftarrow \mathrm{SampleSolution}(A, y, r)$
  if $x \neq \perp$ **then break**
if $x = \perp$ **then return** $\perp$
$x_i \leftarrow x[io : (i+1)o]$; $s_i \leftarrow (v_i + Ox_i) \parallel x_i$ $(0 \leq i < k)$
return $s_0 \parallel \cdots \parallel s_{k-1} \parallel salt$

Algorithm 3 MAYO Verify

Input: Compact public key cpk, message M, signature sig
Output: 0 (valid) or -1 (invalid)
if $|sig| \neq sig_bytes$ **then return** -1
$(seed_{pk}, P_3) \leftarrow cpk$
$(P_1, P_2) \leftarrow \text{AES-128-CTR}(seed_{pk})$ // ExpandPK
$(s, salt) \leftarrow sig$; $s_i \leftarrow s[in : (i+1)n]$ $(0 \leq i < k)$
$d \leftarrow \mathrm{SHAKE256}(M, digest_bytes)$
$(t, K_\Lambda, IV_\Lambda) \leftarrow \mathrm{SHAKE256}(d \parallel salt, m/2 + 32)$
$\Lambda \leftarrow \text{AES-128-CTR}(K_\Lambda, IV_\Lambda, nm/2)$
$y \leftarrow \sum_{i \leq j} E_{ij} u_{ij} + \sum_i E_{i0} \Lambda s_i$
return 0 if $y = t$, otherwise -1

Given $s_0, \ldots, s_{k-1} \in \mathbb{F}_{16}^n$ and $salt$, verification (Algorithm 3) evaluates the quadratic forms across all m matrices,

$$u_{ij} = \begin{cases} (s_i^T P_a s_i)_{a=0}^{m-1}, & i = j, \\ (s_i^T P_a s_j + s_j^T P_a s_i)_{a=0}^{m-1}, & i < j, \end{cases} \tag{3}$$

derives the target $t \in \mathbb{F}_{16}^m$ and linear map $\Lambda \in \mathbb{F}_{16}^{m \times n}$ from the salted message digest, and checks whether [7]

$$\sum_{i \leq j} E_{ij} u_{ij} + \sum_i E_{i0} \Lambda s_i = t. \tag{4}$$

Verification is dominated by expanding (P_1, P_2, Λ) and evaluating the $k(k+1)/2$ bilinear forms u_{ij} over $\mathbb{F}_{16}^{n \times n}$.

4) Signing: To sign a message M (Algorithm 2), the signer expands $\Lambda = (\Lambda_v \mid \Lambda_o)$ with $\Lambda_v \in \mathbb{F}_{16}^{m \times v}$ and $\Lambda_o \in \mathbb{F}_{16}^{m \times o}$, samples vinegar vectors $v_0, \ldots, v_{k-1} \in \mathbb{F}_{16}^v$ and an offset $r \in \mathbb{F}_{16}^{ko}$, and parameterizes each candidate vector as $s_i = (v_i + Ox_i) \parallel x_i$ in the unknown oil vectors $x_i \in \mathbb{F}_{16}^o$. Substituting s_i into Equation (4), the trapdoor in Equation (1) cancels all terms quadratic in x_i. Let $M_i \in \mathbb{F}_{16}^{m \times o}$ have rows $M_i[a, :] = v_i^T L_a$, with $\Lambda_v O + \Lambda_o$ added to M_0. Define $A_i = \sum_{j=0}^{k-1} E_{ij} M_j$ and $A = [A_0 \mid \cdots \mid A_{k-1}]$, and let u_{ij} denote the vinegar quadratic evaluations over $P_{1,a}$. Equation (4) then yields an $m \times ko$ linear system over $\mathbb{F}_{16}$:

$$y = t + \sum_{i \leq j} E_{ij} u_{ij} + \sum_i E_{i0} \Lambda_v v_i, \qquad Ax = y. \tag{5}$$

$\mathrm{SampleSolution}(A, y, r)$ applies Gaussian elimination to the shifted system $A\delta = y - Ar$, returning $x = r + \delta$ if A has full row rank m and $\perp$ otherwise. On failure, the signer increments ctr, resamples (v_i, r), and retries; on success, it forms $s_i = (v_i + Ox_i) \parallel x_i$. Across the signing pipeline, while the matrix contractions $v_i^T L_a$ and u_{ij} expose abundant equation-level parallelism, accumulating the polynomial shifts $E_{ij}M_j$ across columns and resolving sequential pivot dependencies in Gaussian elimination constitute the critical execution bottlenecks on GPUs.

C. GPU Architecture

NVIDIA GPUs schedule cooperative thread arrays (CTAs) on streaming multiprocessors (SMs), with each CTA partitioned into warps of 32 threads [33]. Each SM provides native 32-bit integer operations. Threads within a warp can exchange register values through shuffle instructions, while warps within a CTA commonly exchange data through shared memory and coordinate with barriers. Separate kernels exchange intermediate data through global memory. To hide latency from memory accesses and instruction dependencies, an SM schedules other warps that are ready to execute. Register and shared memory allocations constrain the number of resident CTAs and warps on each SM, limiting the concurrency available to hide latency.

III. Packed Arithmetic and Matrix Parallelism

The efficiency of MAYO's matrix computations depends on both the representation of field elements and the assignment of work to GPU threads. SpreadMAYO jointly organizes packed arithmetic, matrix contractions, and the construction and solution of the signing linear system to expose parallelism within each instance, reuse operands, and reduce synchronization.

A. Packed GF(16) Arithmetic

Many MAYO products multiply the m equation coefficients at a matrix coordinate by the same $\mathbb{F}_{16}$ scalar. The byte lookup implementation [29] extracts two nibbles, reads their products from shared memory tables, and repacks each byte, repeating unpacking and repacking while incurring bank conflicts under divergent lookups. Prior MAYO implementations use bit-plane bitslicing [10] or nibble packing [9], [34].

To address the above limitations, we propose a packed arithmetic design that retains contiguous four-bit coefficients in 64-bit limbs, simplifying compact key serialization, register transposition, and row elimination. For sixteen coefficients $a_0, \ldots, a_{15} \in \mathbb{F}_{16}$, a limb stores

$$u = \mathrm{pack}(a_0, \ldots, a_{15}) = \sum_{j=0}^{15} a_j 2^{4j}. \tag{6}$$

Packed multiplication by $b \in \mathbb{F}_{16}$ conditionally XORs the limb into an accumulator for each bit of b. Shifts, masks, and XORs implement multiplication by x modulo $x^4 + x + 1$: clearing each nibble's x^3 bit before shifting prevents carries between nibbles; those bits determine the reduction mask.

Although 64-bit limbs simplify storage, GPU execution units use a native 32-bit integer datapath, requiring multiple instructions for 64-bit shifts and multiplication. Because the nibble boundaries are isolated, no carry crosses the 32-bit boundary either. Our split32 formulation evaluates the two 8-nibble halves of $u = u_{\mathrm{lo}} + 2^{32}u_{\mathrm{hi}}$ independently:

$$\mathrm{Mul}_{64}(u, b) = \mathrm{Mul}_{32}(u_{\mathrm{lo}}, b) + 2^{32}\,\mathrm{Mul}_{32}(u_{\mathrm{hi}}, b). \tag{7}$$

The same thread processes both halves using native 32-bit shifts, integer multiply-add, and ternary logic instructions.

TABLE II
$\mathbb{F}_{16}$ MULTIPLICATION-ACCUMULATION FOR 16 COEFFICIENTS AND ONE SCALAR ON THE RTX 4090.

Method	SASS ops / 16 elems	Inst. (10^6)	Time (μs)	Speedup
u8 [29]	16 LDS+61 SHF+18 IMAD+54 LOP3	633.43	592.80	1.00×
u64	12 SHF+18 IMAD+18 LOP3	212.93	244.72	2.42×
split32	12 SHF+12 IMAD+16 LOP3	179.38	156.58	3.79×

Table II compares per-limb SASS operations, dynamic warp instruction counts, and execution times. The microbenchmark uses 512 4-warp CTAs, each thread evaluating four independent dependency chains of 512 packed 16-element multiplications. Both packed methods eliminate shared memory table loads. Split32 reduces per-limb SASS operations from 48 to 40 and executed warp instructions by 15.8%, achieving $1.56\times$ and $3.79\times$ speedups over u64 and u8 lookup.

Across the matrix contractions in KeyGen, Sign, and Verify, each matrix entry (r, c) stores its m equation coefficients in $N_{\mathrm{limb}} = \lceil m/16 \rceil$ contiguous 64-bit limbs. We denote limb $\ell \in [0, N_{\mathrm{limb}})$ at matrix coordinate (r, c) by $P_1[r, c, \ell]$; Section IV-A details the padded stride and key conversion.

B. Fine-Grained Matrix Partitioning

SpreadMAYO partitions matrix computations in KeyGen, Sign, and Verify across rows, output positions, and 64-bit limbs to balance operand reuse within threads with parallelism across threads. We illustrate this approach using KeyGen, which computes $B_a = P_{1,a}O + P_{2,a}$ and $P_{3,a} = \mathrm{Upper}(O^T B_a)$ across m equations. A coarse slice-level partitioning (Figure 1(a), as adopted in [29]) assigns one thread to a byte slice $q \in [0, m/2)$ and computes the upper triangles of two $P_{3,a}$ matrices by traversing all v vinegar rows sequentially, forming the current B row in registers, and immediately updating all $o(o+1)/2$ upper-triangular outputs for its equation pair. Here, q indexes bytes containing two field coefficients, whereas ℓ indexes 64-bit limbs containing sixteen coefficients. While this avoids materializing B, it confines both the traversal of v rows and all $o(o+1)/2$ outputs inside each thread's local loop.

To unlock parallelism across both matrix dimensions and equation limbs, we decouple the two matrix products at B and partition work at the granularity of individual rows and 64-bit limbs. As shown in Figure 1(b), a thread assigned to (r, ℓ) initializes o packed accumulators from $P_2[r, :, \ell]$, traverses upper-triangular columns $c \geq r$, and reuses each loaded limb $P_1[r, c, \ell]$ across all o oil scalars $O[c, :]$ before storing $B[r, :, \ell]$ (Figure 1(c)). Because threads within the same warp are assigned to several consecutive rows r, having each thread start its inner loop at its own diagonal $c = r$

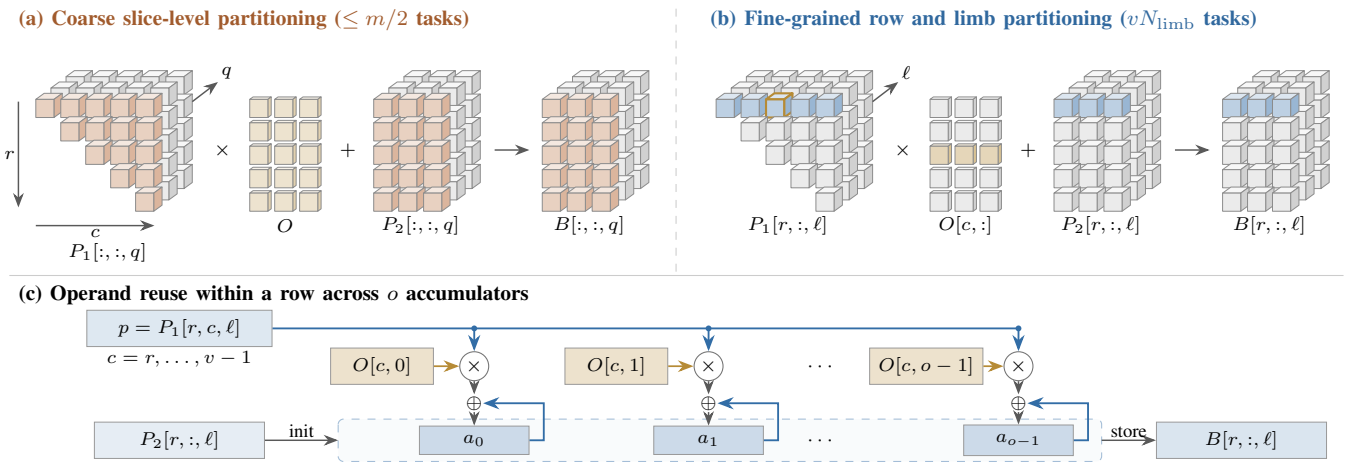

Fig. 1. Thread mapping and operand reuse for $B = P_1O + P_2$ in KeyGen. (a) Coarse partitioning assigns tasks to equation byte slices q spanning all matrix rows. (b) Fine-grained partitioning assigns tasks to individual rows r and 64-bit limbs ℓ. (c) Expanded view of one (r,ℓ) task in (b), showing reuse of each loaded limb across o register accumulators initialized from $P_2[r,:,\ell]$, producing $B[r,:,\ell]$.

would cause lanes in the same warp to read different rows $O[c,:]$ from shared memory in the same cycle. We therefore align the loop starting index of all lanes in a warp to the minimum row index $r_{\min}$ held by that warp and predicate off updates while $c < r$, ensuring that all 32 lanes access the exact same shared memory address $O[c,:]$ at every iteration via conflict-free warp broadcast. While retaining temporal locality in registers across the o oil columns as in RMME [29], this fine-grained (r,ℓ) assignment restricts each thread to a single row and limb, allowing the remaining CTA threads to evaluate distinct rows and limbs concurrently. In the second stage, we distribute the upper-triangular positions of P_3 across threads: each thread assigned to (i,j,ℓ) sums $B[r,j,\ell]O[r,i]$ over vinegar rows and, when $i \neq j$, also sums $B[r,i,\ell]O[r,j]$, directly computing $\text{Upper}(O^TB)$ without forming the lower triangle before writing the completed limb to the compact key.

The two stages expose vN_{limb} work items for B and $o(o+1)N_{\text{limb}}/2$ for P_3, whereas coarse slice-level partitioning exposes at most $m/2$ equation pairs across all parameter sets. For example, on MAYO-2, the P_3 computation expands from 32 slice tasks to 364 (i,j,ℓ) tasks. Threads process these work items through strided loops, allowing the work item count to exceed the CTA size; each B item holds o packed accumulators, while each P_3 item holds only one. This finer partitioning requires writing voN_{limb} intermediate 64-bit words to global memory for downstream threads to read. Section V-A describes storage reuse that avoids an additional allocation. Sections V-B and V-C describe the corresponding matrix partitioning for Sign and Verify.

C. Column Accumulation and Deferred Reduction

Constructing the signing matrix $A = [A_0 \mid \cdots \mid A_{k-1}]$ from the bilinear blocks $A_i = \sum_{j=0}^{k-1} E_{ij}M_j$ (Algorithm 2) is governed by the column-wise structure of the whipping operator $E_{ij} = E^{\ell(i,j)}$, which shifts the m equation coefficients of a column upward by $\ell(i,j)$ polynomial degrees modulo $f(z)$. Specifically, multiplying a column polynomial of degree below m by z^ℓ shifts its top ℓ coefficients into degrees $m, m+1, \ldots, m+\ell-1$; we refer to these terms at degrees $\geq m$ as *overflow coefficients*, which must be folded back into degrees $< m$ via $z^m \equiv c_3z^3 + c_2z^2 + c_1z + c_0 \pmod{f(z)}$. If A is assembled directly in row-major shared memory across threads [29], the loop iterates over all $N_{\text{pair}} = k(k+1)/2$ unordered pairs (i,j), shifts each contribution, immediately folds its $\ell(i,j)$ overflow coefficients modulo $f(z)$ into shared memory, and executes a warp barrier after every pair, repeatedly reducing contributions that land on the same overflow degree $\geq m$.

Each column of A is accumulated privately in registers (Fig. 2(a)). Within a 4-warp CTA, thread t processes columns $c = t + 128q < C$, where $C = 32\lceil ko/32 \rceil$ and $q = 0, 1, \ldots$. Padded tasks with $c \geq ko$ skip input loads and retain zero accumulators for transposition.

Because block A_i depends only on the k source blocks $M_0, \ldots, M_{k-1}$, precomputing the $k \times k$ shift lookup table $\ell(i,j)$ allows a thread assigned to a column in block i to iterate directly over $j \in \{0,\ldots,k-1\}$ in $k \in \{10,5,11,12\}$ steps rather than scanning all $N_{\text{pair}} \in \{55,15,66,78\}$ pairs, while ensuring that adjacent threads in a warp read the same source block M_j in lockstep. Rather than reducing modulo $f(z)$ at every step, the thread accumulates all k shifted vectors into an extended private register buffer of degree $< m + N_{\text{pair}} - 1$ and reduces the sum modulo $f(z)$ only once after the loop. For a fixed column, let $g_\ell(z)$ denote the unshifted input at shift ℓ. By linearity of reduction,

$$\sum_{\ell=0}^{N_{\text{pair}}-1} \left(z^\ell g_\ell(z) \bmod f(z)\right) = \left(\sum_{\ell=0}^{N_{\text{pair}}-1} z^\ell g_\ell(z)\right) \bmod f(z). \quad (8)$$

All shifted contributions landing on the same overflow degree $d \in [m, m+N_{\text{pair}}-2]$ are combined by bitwise XOR prior to a single reduction pass.

To quantify the reduction savings across all ko columns of A, observe that at shift $\ell \in [1, N_{\text{pair}})$, a diagonal pair with $i = j$ sets $w_\ell = 1$ and updates o columns of A_i, whereas an off-diagonal pair with $i < j$ sets $w_\ell = 2$ and updates $2o$ columns across A_i and A_j, each producing ℓ

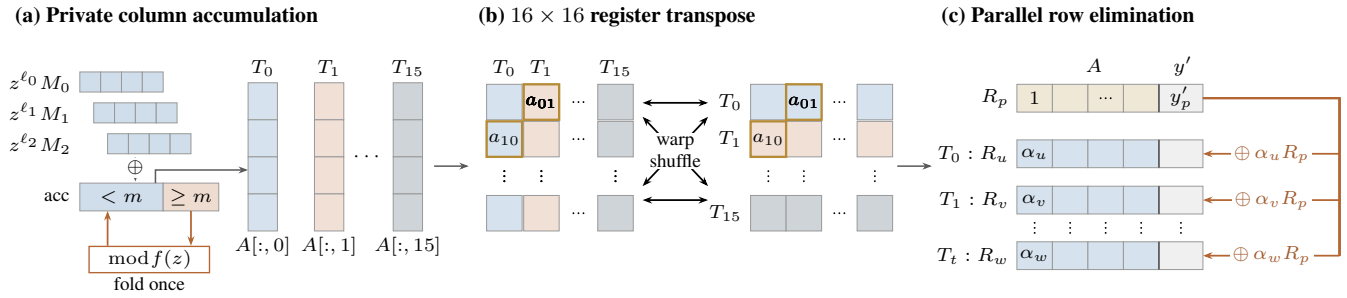

Fig. 2. Signing system construction and solving: (a) private column accumulation with deferred reduction; (b) register transpose via warp shuffles; (c) parallel row elimination in shared memory. In (a,b), colours indicate thread ownership; outlined entries in (b) mark exchanged coefficients.

TABLE III
OVERFLOW COEFFICIENTS FOLDED ACROSS ALL ko COLUMNS OF A. PARENTHESES SHOW REDUCTIONS RELATIVE TO $N_{\text{per-pair}}$.

Param.	(m, k, o)	N_{pair}	$N_{\text{per-pair}}$ $\Theta(ok^4)$	N_{deferred} $\Theta(ok^3)$
MAYO-1	$(80, 10, 8)$	55	21,984	4,320 (−80.3%)
MAYO-2	$(64, 5, 13)$	15	2,236	910 (−59.3%)
MAYO-3	$(108, 11, 10)$	66	39,420	7,150 (−81.9%)
MAYO-5	$(142, 12, 12)$	78	65,508	11,088 (−83.1%)

overflow coefficients per updated column. Immediate per-pair reduction therefore folds $N_{\text{per-pair}}$ overflow coefficients across the matrix, whereas deferred reduction folds the $N_{\text{pair}} - 1$ accumulated overflow degrees exactly once for each of the ko columns, denoted N_{deferred}:

$$N_{\text{per-pair}} = o \sum_{\ell=1}^{N_{\text{pair}}-1} w_\ell \ell = \Theta(ok^4),$$
$$N_{\text{deferred}} = ko(N_{\text{pair}} - 1) = \Theta(ok^3). \tag{9}$$

Because all four parameter sets satisfy $N_{\text{pair}}+1 < m$, folding any overflow degree $\leq m + N_{\text{pair}} - 2$ with the cubic tail of $f(z)$ produces only degrees $< m$ in a single pass. As summarized in Table III, deferred reduction reduces the total number of folded overflow coefficients across A by 59.3%–83.1%. Our implementation sizes the private column buffer to $\lceil (m + N_{\text{pair}} - 1)/16 \rceil \in \{9, 5, 11, 14\}$ 64-bit limbs, updates its words via compile-time constant predication rather than dynamic array indexing, and folds overflow limbs with split32 arithmetic, keeping the entire column accumulator in registers with zero local memory spills.

D. Parallel Gaussian Elimination

Whereas polynomial shift accumulation is independent across the ko columns of A, forward Gaussian elimination distributes the rows below each normalized pivot across threads. In the byte layout of [29], all 32 lanes cooperate to update one target row and process target rows sequentially, executing a warp barrier after each nonzero update and incurring $O(m^2)$ barriers over the elimination. Recent CPU work on MAYO Round 3 likewise highlights the benefits of matrix transposition and echelon-form row updates [11]. Packing sixteen row coefficients into each 64-bit limb lets one thread update the $\lceil (ko + 1)/16 \rceil \in \{6, 5, 7, 10\}$ limbs of a target row, enabling parallel row updates with $O(m)$ CTA barriers.

Algorithm 4 Register transpose and shared memory writeback

Input: Thread $t \in [0, 128)$; reduced column limbs $R_c[\ell]$, zero for $c \geq ko$
Output: Packed rows of A in shared memory
M_s: low s nibbles of each $2s$-nibble group

$C \leftarrow 32\lceil ko/32 \rceil$, $j \leftarrow t \bmod 16$
for $c = t, t + 128, \ldots < C$ **do**
  for $\ell = 0, \ldots, \lceil m/16 \rceil - 1$ **do**
    $w \leftarrow R_c[\ell]$
    for $s \in \{1, 2, 4, 8\}$ **do**
      $d \leftarrow 4s$; $p \leftarrow \textsc{ShuffleXor}(\texttt{0xffffffff}, w, s, 16)$
      $w \leftarrow \begin{cases} w \oplus \big(((p \gg d) \oplus w)\ \&\ M_s\big), & j\ \&\ s \neq 0, \\ w \oplus \big((((w \gg d) \oplus p)\ \&\ M_s) \ll d\big), & \text{otherwise} \end{cases}$
    end for
    $r \leftarrow 16\ell + j$, $b \leftarrow \lfloor c/16 \rfloor$
    if $r < m$ and $b < \lceil ko/16 \rceil$ **then**
      $A[r, b] \leftarrow w$
    end if
  end for
end for
CTA barrier

Algorithm 4 converts packed columns into packed rows through a 16×16 nibble transpose within each 16-lane half of a warp (Fig. 2(b)), keeping all exchanges in registers before writing A to shared memory for augmentation with $y' = y - Ar$.

At each pivot column, one thread searches for a nonzero pivot. When one is found, its inverse is retrieved from a packed constant, and the pivot row is swapped into place and normalized across its limbs. Following CTA synchronization, threads iterate over the remaining rows below the pivot via strided loops, applying $R_u \leftarrow R_u \oplus \alpha_u R_p$ in parallel (Fig. 2(c)) with a constant number of CTA barriers per successful pivot step. If A lacks full row rank, signing restarts with fresh samples; otherwise, the first warp performs back-substitution to recover $x = r + \delta$. Consequently, private column accumulation, register transposition, and parallel row elimination form a unified on-chip pipeline without intermediate global memory traffic.

IV. Data Layout and Symmetric Primitives

MAYO generates compact byte streams through AES-128-CTR and SHAKE256, whereas the matrix datapath developed in Section III operates on packed 64-bit limbs. We bridge these representations through aligned matrix storage and sym-

TABLE IV
MATRIX STRIDES AND AES WORKLOADS (16-BYTE BLOCKS).

Param.	Compact $b = m/2$ (B)	Padded stride $s = 8N_{\text{limb}}$ (B)	Public (P_1, P_2) (AES-128 blks)	Auxiliary Λ (AES-128 blks)
MAYO-1	40	40 (5×8)	9,700 (155 KB)	220 (3.5 KB)
MAYO-2	32	32 (4×8)	7,300 (117 KB)	172 (2.8 KB)
MAYO-3	54	56 (7×8)	23,511 (376 KB)	399 (6.4 KB)
MAYO-5	71	72 (9×8)	52,616 (842 KB)	684 (10.9 KB)

metric primitive implementations designed for the resource constraints of fused GPU kernels.

A. Padded 64-Bit Limb Layout

Following the MAYO specification [7], matrices P_1, P_2, and L use a position-major layout: the m equation coefficients at each coordinate (r, c) are contiguous. SpreadMAYO groups these coefficients into $N_{\text{limb}} = \lceil m/16 \rceil$ 64-bit limbs instead of accessing individual bytes [29]. When compact vector lengths are not multiples of eight bytes, consecutive matrix entries are misaligned. We zero-pad the final limb of each vector to a uniform stride $s = 8N_{\text{limb}}$, ensuring aligned 64-bit loads and stores throughout matrix traversal. Table IV lists the compact and padded sizes for each parameter set. Public matrix expansion produces the padded layout directly; P_3 serialization removes padding to preserve the compact key format.

The linear solver instead uses rows in shared memory, with each limb packing sixteen adjacent column coefficients of one equation after register transposition. Appending the right-hand side $y - Ar$ gives $\lceil (ko + 1)/16 \rceil$ useful limbs per row: 6, 5, 7, and 10 for MAYO-1/2/3/5. Padding to odd strides reserves 7, 5, 7, and 11 limbs per row, respectively. Sampling retains the specification's compact byte boundaries for vinegar vectors v_i and the solver offset r, masking unused high nibbles during decoding [7], [34].

B. Hybrid AES-128-CTR Expansion

MAYO invokes AES-128-CTR [35] for two substantially different workloads: expanding the large public matrices (P_1, P_2) and generating the much smaller linear map Λ derived from the message. As shown in Table IV, the former processes thousands to tens of thousands of counter blocks per instance, whereas the latter requires only a few hundred. The choice of AES engine must therefore account for both expansion throughput and resource usage in the fused kernels.

For public matrix expansion, we adapt the barrel ShiftRows representation of Adomnicai and Peyrin [30] and its GPU mapping by Lee et al. [31]. Each thread holds eight AES blocks in 32 state words, with ShiftRows implemented as register rotations. The Boolean S-box network uses 71 PTX $\mathrm{lop3.b32}$ instructions per invocation [36] to evaluate logic functions of three inputs in registers, avoiding S-box table lookups during the encryption rounds. The generated coefficients are then written into the padded matrix layout. For MAYO-1/2, the compact output already matches the limb stride, allowing direct aligned stores. For MAYO-3/5, threads stage compact outputs in shared memory and cooperatively write aligned 64-bit limbs with zero padding.

For the smaller map Λ, we retain a compact T-table AES engine in shared memory [29], [37]. On MAYO-3/5, barrel AES requires additional shared memory for output staging. This storage remains reserved throughout the fused kernel, potentially reducing CTA residency during subsequent computation stages. As shown in Section VI-E, applying barrel AES to Λ accelerates standalone expansion but reduces end-to-end Sign throughput on these two parameter sets. Combining the two engines therefore balances expansion throughput with the shared memory requirements of the complete kernel.

C. Warp SHAKE256 Hashing

Throughout KeyGen, Sign, and Verify, SHAKE256 [38] derives the public matrix seed $seed_{pk}$, secret matrix O, message digest, salt, target t, and candidate samples (V, r). We build on the warp Keccak-$f[1600]$ mapping of Shen et al. [39], in which 25 lanes of a warp each hold one 64-bit state word in registers and exchange state words via warp shuffles across the 24 permutation rounds without shared memory state arrays. However, directly inlining this standalone routine into fused MAYO kernels under high register pressure causes local memory stack spills when lanes index per-lane rotation offset arrays. SpreadMAYO eliminates these spills by placing the 24 round constants in constant memory and encoding the 25 rotation offsets of the ρ step in four packed 64-bit constants; each lane selects a constant and extracts its offset once into a register before the 24-round loop. Moreover, rather than staging partial final squeeze blocks through shared memory before copying bytes to destination buffers [39], lanes write requested squeeze tail bytes directly from registers to the target buffer while preserving padding and domain separation.

V. FUSED KERNEL ARCHITECTURE

SpreadMAYO integrates the preceding arithmetic, data layouts, and symmetric primitives into fused KeyGen, Sign, and Verify pipelines. As illustrated in Fig. 3, the main computation for each instance executes within a single CTA, with intermediate data retained between stages and storage reused when lifetimes do not overlap. This section describes the resulting computation flows and batch execution modes.

A. Fused Key Generation

KeyGen fuses seed expansion, public matrix generation, and the two matrix stages of Section III-B into a single kernel. First, warp SHAKE256 expands $seed_{sk}$ into $seed_{pk}$ and the secret matrix O, staging O in shared memory so that threads computing both $B_a = P_{1,a}O + P_{2,a}$ and $P_{3,a} = \mathrm{Upper}(O^T B_a)$ can reuse it without global loads. Next, the 8-block barrel AES engine expands P_1 and P_2 directly into the padded 64-bit limb layout in global memory. During the first matrix stage, each (r, ℓ) thread initializes its o register accumulators from row r of P_2, accumulates the upper-triangular products with P_1 and O, and overwrites the consumed P_2 region in place with the completed row of B. A single CTA

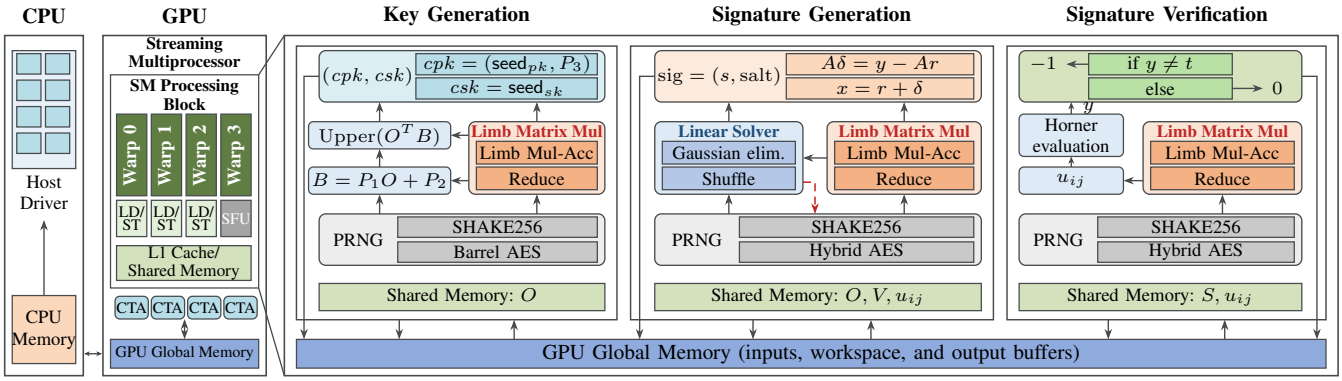

Fig. 3. Fused execution architecture of SpreadMAYO. Shared memory stores intermediate state within each CTA; final outputs are written to global memory. In Verify, matrix and linear contributions accumulate into u_{ij} before Horner evaluation yields y.

barrier then separates these writes from the independent P_3 reductions, whose threads read B and O to serialize the upper-triangular coefficients of P_3 directly into the compact public key cpk while stripping trailing padding bytes. Reusing the P_2 workspace keeps B available to other threads without a separate allocation or launch.

B. Fused Signing

Sign from a compact secret key fuses key preparation with the matrix contractions needed to assemble A and y (Algorithm 2). Let $V \in \mathbb{F}_{16}^{k\times v}$ stack the vinegar vectors. The Round 3 reference software avoids explicitly constructing L when signing from a compact secret key [7], [34]. SpreadMAYO instead constructs L to use the same downstream $M_i = v_i^T L$ computation in both signing modes and reuse L across retries. Because constructing L and $P_1 V^T$ traverses the same public matrix P_1, SpreadMAYO defers L until V is available on the first signing attempt. Each thread assigned to (r, ℓ) uses $P_1[c, r, \ell]$ for $L[r, :]$ when $c < r$, and $P_1[r, c, \ell]$ for $(P_1 V^T)[r, :]$ when $c \geq r$. For $c > r$, the same loaded limb also updates $L[r, :]$; the diagonal contributes only to $P_1 V^T$. The joint traversal avoids redundant upper-triangular reads and stores L in the consumed P_2 buffer.

Linear terms are folded into these contractions rather than evaluated separately. Since $E_{j0} = E_{0j}$, the contribution $\Lambda_v v_j$ shares the polynomial shift of u_{0j}. Initializing the first accumulator of each $P_1 V^T$ row with $\Lambda_v[a, r]$ folds this contribution into the row products. Threads immediately multiply their completed row accumulators by vinegar coefficients and combine the resulting symmetric pairs u_{ij} in shared memory using pairs of 32-bit `atomicXor` instructions, avoiding global intermediates for $P_1 V^T$ and $V(P_1 V^T)$:

$$u_{0j} \leftarrow u_{0j} + \Lambda_v v_j \ (0 \leq j < k), \quad y = t + \sum_{i \leq j} E_{ij} u_{ij}. \tag{10}$$

Contributions from different rows and warps update the same accumulators, so shared atomic XORs combine them safely before the CTA barrier. The quadratic and linear contributions to y thus share one Horner evaluation over decreasing $\ell(i, j)$.

The bilinear blocks $M_i = v_i^T L$ similarly share one traversal of L, rather than reloading it for each of the k blocks. Since $oN_{\mathrm{limb}} \leq 108 < 128$, one CTA assigns a thread to each (b, ℓ) pair. The thread initializes its first accumulator from $\Lambda_o[:, b]$, loads each limb $L[r, b, \ell]$ once, and updates all k accumulators alongside $\Lambda_v[:, r] O[r, b]$. This folds $\Lambda_v O + \Lambda_o$ into M_0 while reducing global reads of L by a factor of k:

$$\begin{aligned} M_0[a, :] &\leftarrow v_0^T L_a + (\Lambda_v O + \Lambda_o)[a, :], \\ M_i[a, :] &\leftarrow v_i^T L_a \quad (1 \leq i < k). \end{aligned} \tag{11}$$

Because the shared memory buffer reserved for the linear solver $[A \mid y - Ar]$ is idle during matrix contractions and after Gaussian elimination, Sign time-multiplexes this single buffer across execution phases: prior to assembling A, the first part of the buffer caches O and the decoded vinegar vectors V while the second part simultaneously holds the symmetric pair accumulators u_{ij}; after Gaussian elimination completes, the same buffer is reused to assemble the output signature s, eliminating global memory buffers for V, $P_1 V^T$, $V(P_1 V^T)$, and s with zero additional static shared memory. If the linear system $Ax = y$ lacks full row rank, the CTA increments ctr and restarts sampling within the same resident kernel, reusing the live key and message state $salt, O, P_1, L, t, \Lambda$. Each retry resamples (V, r), recomputes u_{ij} and M_i, rebuilds (A, y), and solves the new system.

C. Fused Verification

Verification (Algorithm 3) evaluates u_{ij} and Λs_i across the full $n \times n$ public matrices P_a and checks Equation (4), requiring neither A nor Gaussian elimination. Partitioning this evaluation on GPUs faces a tension between thread concurrency and intermediate memory footprint: a coarse slice-level loop over all n rows and columns per byte slice [29] avoids global intermediates but exposes only $m/2 \leq 71$ byte-slice tasks for one warp, with $O(n^2 k)$ work per task, whereas evaluating row and limb contractions of $S(P_a S^T)$ in two stages, where $S \in \mathbb{F}_{16}^{k\times n}$ stacks $s_0^T, \ldots, s_{k-1}^T$, exposes nN_{limb} parallel tasks across the CTA threads but stores all $n \times k$ vectors of $P_a S^T$, or $8nkN_{\mathrm{limb}}$ bytes per instance, in global memory.

Using the same on-the-fly shared memory accumulation of u_{ij} as in Section V-B, SpreadMAYO achieves parallel row contraction across the full $n \times n$ public map in Verify without materializing $P_a S^T$ in global memory. Each thread

assigned to (r, ℓ) decodes the signature S into shared memory, maintains k register accumulators for row r of $P_a S^T$ with the first accumulator initialized to $\Lambda[a, r]$ to fold Λs_i into u_{0i} as in Equation (10), and traverses column c of P_1, P_2 when $r < v$ or P_3 when $r \geq v$. Immediately upon completing row r in registers, the thread multiplies its k row accumulators by $S[i, r]$ and accumulates the products directly into the $k(k+1)/2$ vectors u_{ij} in shared memory using pairs of 32-bit atomicXor instructions. After a single CTA barrier, the first warp evaluates the Horner recurrence over u_{ij} and compares the result against t.

D. Batch Execution Pipelines

At the grid level, SpreadMAYO maps each cryptographic instance in a batch of size B to one 4-warp CTA. The SMs schedule these CTAs as resources become available. Each CTA covers all $M_i = v_i^T L$ outputs in one pass; larger contractions and row updates use strided loops. This choice balances parallelism against register demand per CTA and idle threads during SHAKE256, pivot search, and back-substitution. To eliminate runtime allocation overhead across large batches, each instance occupies one row of a preallocated pitched memory pool aligned to 128-byte cache line boundaries, with B and L overwriting P_2 in place across distinct phases.

SpreadMAYO provides two batch execution pipelines tailored to distinct deployment scenarios:

- **Multi-key batch processing:** When a service processes independent requests under distinct compact keys as evaluated by Chen et al. [29], callers invoke the fused compact-key kernels. Each CTA expands its instance's compact seed on the fly and immediately consumes the generated matrices within the same kernel, computing L jointly with $P_1 V^T$ on the first signing attempt without separate expansion launches.
- **Repeated-key server processing:** For batches of distinct messages signed under one secret key, SpreadMAYO separates static key preparation from online signing, following prior GPU signing designs [16], [17]. A standalone $\mathrm{ExpandSK}$ kernel precomputes $(P_1, L, O, \mathsf{csk})$ once in global memory, allowing the shared state to be reused through the GPU's L2 cache. Each signing CTA uses private scratch space, derives its own message state, and generates samples (V, r) for each signing attempt. Only static key material is shared across instances. This reduces the memory footprint of each instance and avoids repeated key expansion.

To hide PCIe transfer latency, SpreadMAYO uses asynchronous batching, as explored in GPU cryptographic services [40]. The host partitions the pitched memory pool into disjoint row subbatches and dispatches them across multiple nonblocking CUDA streams, overlapping host-device data transfers with kernel execution.

VI. Evaluation

A. Experimental Setup

Our GPU baseline adapts Chen et al.'s [29] open source Round 2 implementation to MAYO Round 3 [7]. We update parameters, the reduction polynomial and whipping schedules, add the linear map Λ derived from the message to Sign and Verify, handle odd vector lengths in MAYO-2, and correct retry and synchronization errors. The byte layout, shared memory lookup arithmetic, matrix evaluation, solver, and instance mapping are retained: one warp for KeyGen, ExpandSK, Sign, and Verify, and four for ExpandPK. Both implementations match the CPU reference byte for byte on official Round 3 known answer tests and reject tampered or mismatched verification inputs.

All GPU experiments run on an NVIDIA GeForce RTX 4090 (SM89) using CUDA Toolkit 12.9 (nvcc 12.9.86), with kernels compiled using -O3. Full batch evaluations and complete API ablations process $B = 20{,}000$ instances across ten nonblocking streams at $2{,}000$ instances per stream; standalone component benchmarks and Nsight Compute profiling use a single stream of $B = 2{,}000$ instances unless otherwise noted. Sign and Verify start from compact keys unless stated otherwise. Throughput in ops/s denotes the number of complete KeyGen, Sign, or Verify operations processed per second. End-to-end timings include host-device transfers, key expansion, kernel execution, and synchronization; CUDA event timings exclude transfers but include coordination between streams. After warmup, we average twenty measurements for each of four input groups and report the median group time. Speedups in Tables V and VI divide the reported throughput of Ours by that of the GPU baseline; ablation comparisons use paired group time ratios. Main throughput and ablation results are measured in separate runs.

B. End-to-End Throughput

Table V compares end-to-end full batch throughput on the RTX 4090 against the official MAYO Round 3 reference and GFNI/AVX-512 CPU results [7] reported for an Intel Xeon Gold 6338 at 2.0 GHz and against the GPU baseline of Chen et al. [29]. The CPU throughputs are converted from the reported cycle counts at 2.0 GHz, including key expansion for Sign and Verify. Compared with the portable C reference and GFNI/AVX-512 implementations, SpreadMAYO improves throughput by approximately $4{,}046$–$5{,}418\times$ and 40.8–$69.4\times$ for KeyGen, $3{,}607$–$4{,}793\times$ and 59.7–$90.9\times$ for Sign, and $2{,}661$–$4{,}608\times$ and 42.6–$71.3\times$ for Verify, respectively. Relative to the GPU baseline of Chen et al. [29], SpreadMAYO achieves speedups of 2.33–$3.71\times$ in KeyGen, 4.40–$6.93\times$ in Sign, and 3.42–$5.13\times$ in Verify. Separate CUDA event timings excluding host-device transfers show similar gains. Moreover, our 128-byte aligned pool layout increases global memory usage by only 1.0%–5.0% across KeyGen and Sign compared with [29], while all SpreadMAYO production kernels compile with zero stack frame and zero spills to local memory.

Figure 4 evaluates single-stream end-to-end throughput as the batch size doubles from $B = 16$ to $16{,}384$. By combining fine-grained parallelism within each 4-warp CTA with key expansion fused into Sign and Verify, SpreadMAYO provides a clear throughput advantage at small batches. As the batch size increases, throughput gradually approaches its peak.

Table VI evaluates repeated-key signing with offline key expansion in both implementations. We extend the adapted

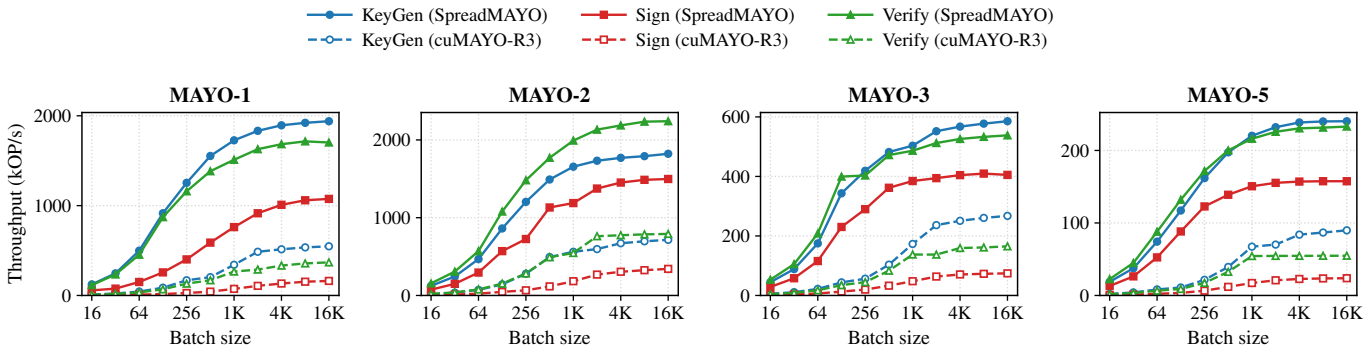

Fig. 4. Single-stream end-to-end throughput scaling across batch sizes $B = 16$ to 16,384 on the RTX 4090.

TABLE V
END-TO-END MAYO ROUND 3 THROUGHPUT (OPS/S); SPEEDUP OVER THE ADAPTED GPU BASELINE.

Param.	Op.	Ref [7]	GFNI/ AVX-512 [7]	Ref [29]*	Ours	Speedup
MAYO-1	KeyGen	387	30,207	565,430	2,096,600	3.71×
MAYO-1	Sign	228	12,020	163,058	1,092,863	6.70×
MAYO-1	Verify	460	27,054	375,750	1,927,770	5.13×
MAYO-2	KeyGen	429	37,124	760,209	2,160,239	2.84×
MAYO-2	Sign	424	19,777	347,899	1,529,431	4.40×
MAYO-2	Verify	659	47,744	878,821	3,036,563	3.46×
MAYO-3	KeyGen	147	13,393	271,756	632,267	2.33×
MAYO-3	Sign	89	5,930	74,607	410,779	5.51×
MAYO-3	Verify	196	13,152	169,963	580,457	3.42×
MAYO-5	KeyGen	62	6,142	91,131	250,846	2.75×
MAYO-5	Sign	39	2,657	22,879	158,609	6.93×
MAYO-5	Verify	92	5,745	56,475	244,820	4.34×

*Adapted from Chen et al.'s [29] Round 2 GPU implementation to MAYO Round 3. Speedup compares throughput (Ours/Ref [29]).

cuMAYO baseline to reuse expanded key material across signing requests. Compared with compact-key signing, SpreadMAYO improves online throughput by 1.92–3.82× while reducing the batch memory pool by 19.1–35.9×, to 117–470 MiB for $B = 20{,}000$ messages. Against the corresponding server-oriented cuMAYO baseline, SpreadMAYO achieves 3.14–10.75× higher signing throughput. The generated signatures match compact-key signing byte for byte.

C. Arithmetic and Matrix Parallelism

To understand how our architectural optimizations contribute to these end-to-end gains, Fig. 5 shows cumulative optimizations to packed arithmetic and matrix/solver parallelism (+Parallel), hybrid symmetric primitives (+Sym), and on-chip accumulation with kernel fusion (+Fusion). Uninstrumented API times determine the bar heights. Thread 0 records `clock64` at synchronized stage boundaries. Category cycles are summed across CTAs, kernels, and input groups, including signing retries, then normalized by their total and multiplied by the uninstrumented API time. Timed regions include synchronization waits; transfer and launch overheads in the API total are distributed proportionally. The segments therefore estimate stage contributions rather than independently measured stage latencies. In Base KeyGen, fused B/P_3 computation and output packing are timed together under Primary Matrix, leaving Reduce & Build at zero. Combined matrix stages dominate the estimated Base stage breakdown, and +Parallel yields speedups of 1.26–1.95× for KeyGen, 3.19–4.75× for Sign, and 1.44–2.55× for Verify.

The +Parallel stage combines arithmetic, matrix partitioning, and solver optimizations; Table II, the component tests, and the solver control in Section VI-D provide complementary evidence for these mechanisms. Separately, retaining the final arithmetic, symmetric primitives, and on-chip accumulation with key expansion in separate kernels achieves up to 6.74× Sign and 3.78× Verify GPU speedups over [29]. Standalone benchmarks with prepared inputs accelerate KeyGen P_3, Sign L, Sign y, and Verify y matrix components by 2.46–2.92×, 4.53–5.82×, 1.76–7.30×, and 1.71–7.19×, respectively. In this layout, split32 (Table II) raises full batch end-to-end throughput over u64 by +6.5%–+13.1% in KeyGen, +15.3%–+22.4% in Sign, and +7.6%–+12.4% in Verify.

D. Linear System Construction and Solving

We next evaluate the signing linear system construction and Gaussian elimination optimizations in Sections III-C–III-D. Relative to a control with dynamic column indexing, separate accumulation for each output of M_i, and additional solver synchronization and scalar field operations, the final design improves end-to-end Sign throughput by +11.5% on MAYO-1, +8.1% on MAYO-2, and +8.7% on MAYO-3/5. Both configurations retain the final register transposition and shared memory layout. Nsight Compute profiling shows that the final Sign kernels have no stack frame or local memory load/store sectors, whereas the control uses a 112 B stack frame on MAYO-1/3/5. Global load sectors decrease by 30.0%–40.4% and executed Sign instructions by 5.4%–6.9%.

E. AES-128-CTR and SHAKE256 Primitives

To evaluate public matrix expansion in Section IV-B, we fix Λ to T-table AES in shared memory and compare our 8-block barrel AES engine against a shared memory T-table public expansion engine derived from Chen et al. [29]. Across all parameter sets, 8-block barrel AES consistently outperforms

TABLE VI
SIGNING THROUGHPUT (OPS/S) AND BATCH POOL MEMORY (MIB): COMPACT-KEY (BASE) VERSUS REPEATED-KEY (SERVER-ORIENTED).

	Ref [29]*			Ours			
Param.	Base	Server-Oriented	Pool	Base	Server-Oriented	Pool	Speedup
MAYO-1	162,701	378,993	3,115 → 147	1,085,946	2,089,746	3,174 → 166	5.51×
MAYO-2	346,990	1,241,677	2,344 → 108	1,521,269	3,895,718	2,393 → 117	3.14×
MAYO-3	74,682	178,513	7,441 → 254	405,449	1,317,989	7,813 → 284	7.38×
MAYO-5	22,928	56,259	16,504 → 430	158,364	604,697	16,875 → 470	10.75×

*Chen et al.'s [29] Round 2 GPU implementation adapted to Round 3. Speedup compares Server-Oriented throughput (Ours/Ref [29]).

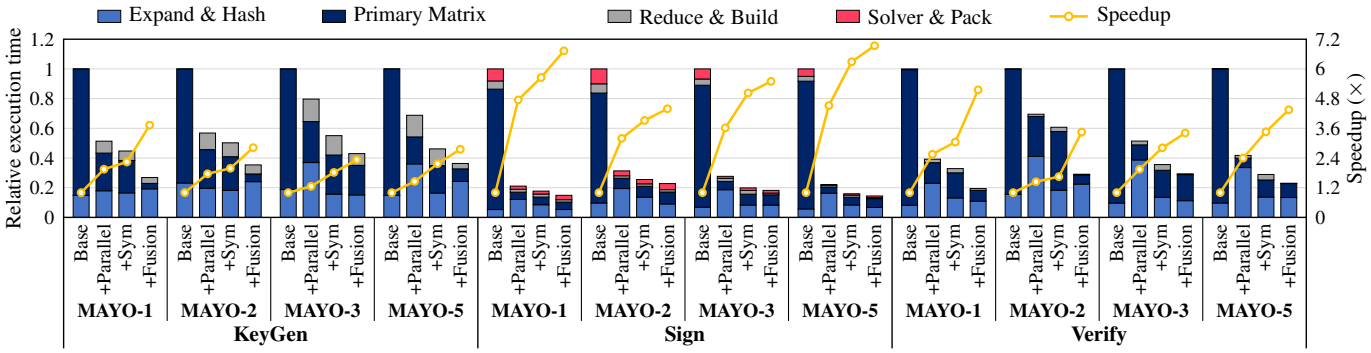

Fig. 5. Cumulative API time normalized to Base and measured speedup ($B = 20{,}000$, ten streams). Segments estimate time from CTA cycle shares measured with `clock64`; fused stages remain grouped. +Parallel adds packed arithmetic, finer matrix and solver parallelism; +Sym adds hybrid AES and warp SHAKE; +Fusion adds on-chip accumulation and kernel fusion.

T-table AES for (P_1, P_2), accelerating standalone public expansion by $+40.2\%$–$+40.9\%$ on MAYO-1/2 and $+109.7\%$–$+119.1\%$ on the padded parameter sets MAYO-3/5. End-to-end throughput improves by $+18.6\%$–$+44.4\%$ in KeyGen on MAYO-1/3/5, $+2.5\%$–$+9.3\%$ in Sign, and $+11.4\%$–$+32.5\%$ in Verify; MAYO-2 KeyGen is 2.3% slower.

Nsight Compute profiling of MAYO-3 padded public matrix expansion explains this gain. Although 8-block barrel AES and T-table AES execute nearly identical warp instruction counts of 1.224 billion vs. 1.207 billion, their ratios of L1TEX store sector bytes to useful compact output bytes are 1.102 and 9.722, with 1.60 vs. 0.74 eligible warps/active cycle: coalesced wide output stores from 8-block barrel AES reduce uncoalesced byte stores and raise warp issue efficiency. Conversely, using 8-block barrel AES to expand the smaller map Λ degrades end-to-end Sign throughput on MAYO-3 and MAYO-5 despite faster standalone expansion, because its shared output staging increases peak shared memory across the fused Sign kernel and reduces CTA residency during the matrix and solver phases.

For SHAKE256 hashing in Section IV-C, executing all preparation and sampling hashes on our warp Keccak core with round constants in constant memory and the 25 rotation offsets of ρ encoded in four packed 64-bit constants eliminates local memory stack spills across lanes. Compared with a mixed serial/warp SHAKE control, this reduces the per-thread Sign stack frame from 416 B to zero across the four parameter sets and reduces MAYO-2 Verify from 93 registers and 416 B stack to 80 registers and 0 B stack, improving end-to-end throughput by up to $+25.7\%$ in Sign and $+17.0\%$ in Verify.

F. On-Chip Accumulation and Kernel Fusion

We first evaluate the joint traversal of P_1 for constructing L and P_1V^T during the first signing attempt (Section V-B). Compared with two separate traversals under the same shared memory layout, this optimization improves Sign throughput by $+8.9\%$ and $+5.6\%$ on MAYO-1 and MAYO-2, and by $+15.7\%$ and $+10.2\%$ on MAYO-3 and MAYO-5, respectively.

Next, on-chip accumulation combines symmetric pair terms u_{ij} directly in shared memory using 32-bit `atomicXor` instructions instead of materializing row products in global memory (Sections V-B and V-C). This improves compact-key Sign throughput by $+1.0\%$–$+8.7\%$ and Verify throughput by up to $+16.4\%$, with MAYO-3 Verify approximately unchanged (-0.3%). For repeated-key signing, throughput improves by up to $+22.1\%$, with a 3.9% decrease on MAYO-2, while the online memory pool shrinks by 3.00–$6.41\times$.

Finally, we isolate kernel fusion by comparing fused Sign and Verify against execution with separate expansion kernels under the same padded limb layout (Section V-D). Fusing key expansion directly into the consumer CTA improves throughput by up to $+11.3\%$ in Sign and $+100.3\%$ in Verify. In MAYO-1 Verify, the expanded (P_1, P_2) matrices for 2,000 instances occupy 310.4 MB, exceeding the 72 MiB L2 cache. Keeping expansion and consumption within the same CTA reduces total DRAM reads from 315.03 MB to 5.80 MB and raises the L2 hit rate from 10.91% in the separate consumer to 99.40% in the fused kernel.

VII. CONCLUSION

We presented SpreadMAYO, a high-throughput GPU implementation of MAYO Round 3 that combines packed field

arithmetic, fine-grained matrix parallelism, and kernel fusion. By coordinating data representation, work partitioning, and storage reuse, the design exposes parallelism within each instance while reducing synchronization and intermediate global memory traffic. On an RTX 4090, SpreadMAYO achieves throughput speedups of 2.33–$3.71\times$ for KeyGen, 4.40–$6.93\times$ for Sign, and 3.42–$5.13\times$ for Verify over a Round 3 adaptation of Chen et al. [29]. With key expansion performed offline in both implementations, signing under one key reaches up to $10.75\times$. Future work will explore constant-time pivot handling in the signing solver to reduce the risk of timing leakage, and extend these design principles to other post-quantum schemes with similar finite-field matrix workloads.

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] National Institute of Standards and Technology, "Module-lattice-based key-encapsulation mechanism standard," U.S. Dept. of Commerce, FIPS Pub. 203, Aug. 2024.

[3] ——, "Module-lattice-based digital signature standard," U.S. Dept. of Commerce, FIPS Pub. 204, Aug. 2024.

[4] ——, "Stateless hash-based digital signature standard," U.S. Dept. of Commerce, FIPS Pub. 205, Aug. 2024.

[5] ——, "Post-quantum cryptography: Additional digital signature schemes," NIST CSRC Project, 2022, updated Oct. 2026. [Online]. Available: https://csrc.nist.gov/projects/pqc-dig-sig

[6] W. Beullens, "MAYO: Practical post-quantum signatures from Oil-and-Vinegar maps," in *Proc. Selected Areas Cryptogr. (SAC)*, ser. LNCS, vol. 13203. Springer, 2022, pp. 355–376.

[7] W. Beullens, F. Campos, S. Celi, B. Hess, and M. J. Kannwischer, "MAYO: Round 3 specification," Round 3 submission specification, 2026. [Online]. Available: https://pqmayo.org/assets/specs/mayo-round3.pdf

[8] G. Alagic, M. Bros, P. Ciadoux, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, H. Silberg, D. Smith-Tone, and N. Waller, "Status report on the second round of the additional digital signature schemes for the NIST post-quantum cryptography standardization process," Nat. Inst. Standards Technol., NIST Interagency/Internal Rep. (NISTIR) 8610, May 2026.

[9] W. Beullens, F. Campos, S. Celi, B. Hess, and M. J. Kannwischer, "Nibbling MAYO: Optimized implementations for AVX2 and Cortex-M4," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2024, no. 2, pp. 252–275, 2024.

[10] M. Samyuktha, P. Borkar, and C. Rebeiro, "High speed high assurance implementations of multivariate quadratic based signatures—extended abstract," in *Proc. Int. Conf. Secur., Privacy, Appl. Cryptogr. Eng. (SPACE)*, ser. LNCS, vol. 15351. Springer, 2025, pp. 196–200.

[11] M. Lee, M. Sim, S. Eum, and H. Seo, "Optimized NEON implementation of MAYO on Apple M1," Cryptology ePrint Archive, Paper 2026/2350, 2026. [Online]. Available: https://eprint.iacr.org/2026/2350

[12] F. Hirner, M. Streibl, F. Krieger, A. C. Mert, and S. S. Roy, "Whipping the multivariate-based MAYO signature scheme using hardware platforms," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*. ACM, 2024, pp. 3421–3435.

[13] T. Zhou, F. Zheng, G. Fan, L. Wan, W. Tang, Y. Song, Y. Bian, and J. Lin, "ConvKyber: Unleashing the power of AI accelerators for faster Kyber with novel iteration-based approaches," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2024, no. 2, pp. 25–63, 2024.

[14] T. Zhou, F. Zheng, Z. Xie, W. Tang, G. Fan, Y. Ning, Y. Bian, J. Lin, and J. Jing, "ML-Cube: Accelerating module-lattice-based cryptography using machine learning accelerators with a memory-less design," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2025, pp. 1829–1843.

[15] X. Ji, J. Dong, T. Deng, P. Zhang, J. Hua, and F. Xiao, "HI-Kyber: A novel high-performance implementation scheme of Kyber based on GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 6, pp. 877–891, 2024.

[16] S. Shen, H. Yang, W. Li, and Y. Zhao, "cuML-DSA: Optimized signing procedure and server-oriented GPU design for ML-DSA," *IEEE Trans. Dependable Secure Comput.*, vol. 22, no. 3, pp. 2295–2307, 2025.

[17] W. Li, H. Wei, S. Shen, H. Yang, W. Dai, and Y. Zhao, "cuFalcon: An adaptive parallel GPU implementation for high-performance Falcon acceleration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 37, no. 5, pp. 1153–1167, 2026.

[18] R. Dai, J. Dong, M. Qiu, Z. Dong, F. Xiao, and J. Lin, "GOLF: Unleashing GPU-driven acceleration for FALCON post-quantum cryptography," *IEEE Trans. Inf. Forensics Security*, vol. 20, pp. 9441–9453, 2025.

[19] J. Dong, Y. Hou, S. Wang, L. Sha, F. Xiao, Z. Dong, and J. Lin, "HIGH: Harnessing GPU parallelism for optimized HQC performance," *IEEE Trans. Comput.*, vol. 75, no. 10, pp. 3483–3497, Oct. 2026.

[20] Y. Ning, J. Dong, Y. Zhao, J. Lin, T. Zhou, J. Wang, and F. Xiao, "X^2O: Cross parallel optimization of the CROSS post-quantum scheme on GPU," *IEEE Trans. Inf. Forensics Security*, vol. 21, pp. 2992–3005, 2026.

[21] Z. Wang, X. Dong, H. Chen, Y. Kang, and Q. Wang, "CUSPX: Efficient GPU implementations of post-quantum signature SPHINCS+," *IEEE Trans. Comput.*, vol. 74, no. 1, pp. 15–28, 2025.

[22] Y. Ning, J. Dong, J. Lin, F. Zheng, Y. Fu, and F. Xiao, "GRASP: Accelerating hash-based PQC performance on GPU parallel architecture," *IEEE Trans. Comput.*, vol. 75, no. 4, pp. 1579–1592, 2026.

[23] F. Zheng, G. Fan, W. Tang, Y. Song, T. Zhou, Y. Zhao, J. Dong, J. Lin, S. Yan, and J. Jing, "GIF-FHE: A comprehensive implementation and evaluation of GPU-accelerated FHE with integer and floating-point computing power," *IEEE Trans. Parallel Distrib. Syst.*, vol. 36, no. 8, pp. 1524–1541, 2025.

[24] G. Fan, M. Zhang, F. Zheng, S. Fan, T. Zhou, X. Deng, W. Tang, L. Kong, Y. Song, and S. Yan, "WarpDrive: GPU-based fully homomorphic encryption acceleration leveraging tensor and CUDA cores," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2025, pp. 1187–1200.

[25] J. Dong, Q. Ye, S. Lu, L. Feng, Z. Dong, J. Lin, and F. Xiao, "HPS: Speeding up Paillier homomorphic cryptosystem on GPU," *IEEE Trans. Comput.*, 2026, Early Access.

[26] Y. Chen, F. Zheng, G. Fan, J. Dong, W. Tang, T. Zhou, J. Lin, and J. Jing, "BOLT-FHE: An efficient unified framework for GPU-based TFHE bootstrapping via on-chip local tiling strategies," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2026, no. 3, pp. 198–223, 2026.

[27] G. Fan, Y. Chen, L. Chen, L. Kong, C. Niu, D. Jiao, Y. Zhu, G. Yang, S. Fan, X. Deng, F. Zheng, J. Weng, M. Li, Y. Chang, S. Yan, and M. Zhang, "HyperDrive: Hierarchical exploitation of memory efficiency for GPU-based FHE acceleration," in *Proc. ACM/IEEE Int. Symp. Comput. Archit. (ISCA)*, 2026, pp. 1955–1969.

[28] J. Choi, D. Kim, and S. C. Seo, "Parallel implementation of MAYO on GPU," in *Proc. Int. Conf. Mobile Internet Secur. (MobiSec)*, 2024, art. P-20. [Online]. Available: https://di0zxmb8pwajl.cloudfront.net/kiisc/conference/mobisec2024/P-20.pdf

[29] H. Chen, J. Huang, S. Jin, R. C. C. Cheung, D. Chen, and W. Dai, "High-throughput GPU design and implementation of MAYO with matrix computation reordering," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2026, no. 4, pp. 496–521, 2026.

[30] A. Adomnicai and T. Peyrin, "Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM Cortex-M and RISC-V," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 1, pp. 402–425, 2021.

[31] W.-K. Lee, S. C. Seo, H. Seo, D. C. Kim, and S. O. Hwang, "Speed record of AES-CTR and AES-ECB bit-sliced implementation on GPUs," *IEEE Embedded Syst. Lett.*, vol. 16, no. 4, pp. 481–484, 2024.

[32] A. Kipnis, J. Patarin, and L. Goubin, "Unbalanced Oil and Vinegar signature schemes," in *Proc. EUROCRYPT*, ser. LNCS, vol. 1592. Springer, 1999, pp. 206–222.

[33] NVIDIA, "CUDA programming guide: Programming model," NVIDIA CUDA Documentation, 2026. [Online]. Available: https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html

[34] MAYO implementation team, "MAYO-C: Reference and optimized implementations," Software repository, 2026. [Online]. Available: https://github.com/PQCMayo/MAYO-C

[35] National Institute of Standards and Technology, "Advanced encryption standard (AES)," U.S. Dept. of Commerce, FIPS Pub. 197, May 2023.

[36] NVIDIA, "Parallel thread execution ISA: Logic and shift instructions," NVIDIA CUDA Documentation, 2026. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[37] W.-K. Lee, H. J. Seo, S. C. Seo, and S. O. Hwang, "Efficient implementation of AES-CTR and AES-ECB on GPUs with applications for

high-speed FrodoKEM and exhaustive key search," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 6, pp. 2962–2966, 2022.
[38] National Institute of Standards and Technology, "SHA-3 standard: Permutation-based hash and extendable-output functions," U.S. Dept. of Commerce, FIPS Pub. 202, Aug. 2015.
[39] S. Shen, H. Yang, W. Dai, H. Zhang, Z. Liu, and Y. Zhao, "High-throughput GPU implementation of Dilithium post-quantum digital signature," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 11, pp. 1964–1976, 2024.
[40] Y. Bian, F. Zheng, Y. Wang, L. Lei, Y. Ma, T. Zhou, J. Dong, G. Fan, and J. Jing, "AsyncGBP+: Bridging SSL/TLS and heterogeneous computing power with GPU-based providers," *IEEE Trans. Comput.*, vol. 74, no. 2, pp. 356–370, 2025.



Zhuoyu Xie received the B.E. degree in information security from the University of Science and Technology of China, Hefei, China, in 2025. He is currently pursuing a Ph.D. degree with the School of Cryptology, University of Chinese Academy of Sciences, Beijing, China. His research interest includes applied cryptography and high-performance computing.



Tian Zhou received the B.E. degree from Hangzhou Dianzi University and the Ph.D. degree from the University of Science and Technology of China. He is currently a postdoctoral researcher with the School of Cyber Science and Technology, University of Science and Technology of China. His research interests include applied cryptography.



Fangyu Zheng (Member, IEEE) received the B.E. degree in information security from the University of Science and Technology of China, Hefei, China, in 2011, and the Ph.D. degree in information security from the University of Chinese Academy of Sciences, Beijing, China, in 2016. He is an Associate Professor with the School of Cryptology, University of Chinese Academy of Sciences, focusing on applied cryptography and high-performance computing.



Yijing Ning received his B.E. degree from the University of Science and Technology of China in 2023, and is currently pursuing a Ph.D. degree with the School of Cyber Science and Technology, University of Science and Technology of China, focusing on cryptographic engineering, post-quantum cryptography, and high-performance computing.



Yi Bian received the B.E. degree from Shandong University, Jinan, China, in 2018, and the Ph.D. degree from the University of Chinese Academy of Sciences, Beijing, China, in 2025. He is currently an Assistant Research Fellow with the School of Cryptology, University of Chinese Academy of Sciences. His research interests include applied cryptography and high-performance computing.



Wangchen Dai received the Ph.D. degree in electronic engineering from the City University of Hong Kong in 2018. After that, he held appointments at Hardware Security Lab, Huawei Technologies Company Ltd., and the Department of CSSE, Shenzhen University. He is currently working as an Associate Professor with Sun Yat-sen University. His research interests include cryptographic hardware and embedded systems, fully homomorphic encryption, and reconfigurable computing.



Jiankuo Dong received the B.E. degree from the Xi'an Jiaotong University, and the Ph.D. degree from the University of Chinese Academy of Sciences in 2014 and 2019, respectively. He is currently a Research Professor with the School of Cyber Science and Technology, University of Science and Technology of China, Hefei, China. His research interests include cryptographic engineering, public key cryptography and applied cryptography.



Yuewu Wang received the Ph.D. degree from the Graduate University, Chinese Academy of Sciences, Beijing, China, in 2008. He is a Professor with the School of Cryptology, University of Chinese Academy of Sciences. His research interests include mobile security, network attack simulation, and information system security evaluation.



Jingqiang Lin (Senior Member, IEEE) received the M.S. and Ph.D. degrees from the University of Chinese Academy of Sciences, in 2004 and 2009, respectively. He is a Full Professor with the School of Cyber Science and Technology, University of Science and Technology of China, focusing on applied cryptography and system security.



Jiwu Jing received the B.E. degree from the Department of Electronics Engineering, Tsinghua University, Beijing, China, in 1987, and the M.Sc. and Ph.D. degrees from the Graduate School, Chinese Academy of Sciences, Beijing, in 1990 and 2003, respectively. He is currently a Professor with the School of Cryptology, University of Chinese Academy of Sciences. His main research interest is information security, including public key infrastructure, identity management, and fault tolerance.