Back to the Bits: Revisiting MAYO's Data Representation for GPU and Tensor-Core Acceleration

JunHyeok Choi, DongCheon Kim and Seog Chung Seo*

Kookmin University, Seoul, Republic of Korea
c981126@kookmin.ac.kr, kindongsy@kookmin.ac.kr, scseo@kookmin.ac.kr

Abstract. MAYO, a multivariate quadratic signature scheme with small public keys and signatures, is a third-round candidate in the additional signature standardization process of the National Institute of Standards and Technology. Cloud and application servers receiving many independent signing requests need to batch them on Graphics Processing Units (GPUs). MAYO's nibble-sliced representation suits CPU shuffle instructions or multiplication tables, but on GPUs it costs a memory access or several instructions per $\mathbb{F}_{16}$ multiplication. The existing GPU implementation cuMAYO (CHES 2026) uses the second-round nibble-sliced representation and looks up a shared-memory table per $\mathbb{F}_{16}$ multiplication. Our GPU implementation keeps all keys and intermediate values in the bitsliced representation yet matches the official Known Answer Test vectors. Its bitsliced multiply-accumulate circuit computes matrix products 32 coefficients at a time with only word-wide AND and XOR. One thread block handles one instance; KeyGen, Sign, and Verify each run as a single kernel from the expansion of the compact key to the output; and representation conversion is confined to reading and writing specification-defined byte strings. For same-key batches, an additional tensor path decomposes $\mathbb{F}_{16}$ matrix products into binary ones and computes the large Sign and Verify matrix products on Tensor Cores. To our knowledge, the tensor path is the first use of Tensor Cores for MAYO, and ours is the first GPU implementation of the third-round specification. On an RTX 4090, the bitsliced implementation achieves 2.75× to 6.73× higher throughput and 6.99× to 34.35× lower single-shot latency than cuMAYO for all three operations of the four parameter sets. In same-key batches, the tensor path, including key preparation, reaches 13.75× to 30.62× the throughput of cuMAYO on distinct-key batches.

Keywords: MAYO · Post-Quantum Cryptography · GPU · Bitsliced Arithmetic · Tensor Cores

1 Introduction

In 2022, the National Institute of Standards and Technology (NIST) selected the signature algorithms CRYSTALS-Dilithium and Falcon (lattice-based) and SPHINCS+ (hash-based) in the NIST Post-Quantum Cryptography (PQC) Standardization Process [AAC+22], and in 2024 standardized CRYSTALS-Dilithium and SPHINCS+ as ML-DSA and SLH-DSA in Federal Information Processing Standards (FIPS) 204 and 205 [Nat24a, Nat24b]. As two of the three rely on structured lattices, NIST called in 2022 for additional general-purpose signatures, primarily those not based on structured lattices, to diversify the hardness assumptions of its standards [Nat22]. This additional signature standardization process

*Corresponding author

entered its third round in May 2026 and now has eight candidates: four multivariate (Unbalanced Oil and Vinegar (UOV), MAYO, QR-UOV, and SNOVA), three MPC-in-the-Head (FAEST, MQOM, and SDitH), and one isogeny-based (SQIsign) [ABC+26, Nat26].

MAYO is a multivariate quadratic signature scheme in the UOV family [Beu22, BCC+26], a family whose largest drawback is a large public key [ABC+26]. It shrinks the public key with a low-dimensional secret subspace (the oil space), offsetting the resulting shortage of unknowns by whipping, a k-fold expansion of the public map in signing and verification [BCC+26]. Noting smaller public keys than traditional UOV with retained signing and verification efficiency, the NIST second-round status report found MAYO reasonable to consider as a general-purpose signature scheme [ABC+26]. Because MAYO keys are compact representations with short seeds, key generation, signing, and verification must expand the public matrices P_1 and P_2 with AES-128 in counter mode (CTR), where AES is the Advanced Encryption Standard [BCC+26], concentrating the computational cost on AES-128 CTR and on the large $\mathbb{F}_{16}$ matrix products over these matrices. In the MAYO specification, the reference implementation without the AES New Instructions (AES-NI) spends 72% to 79% of verification cycles on public-key expansion [BCC+26].

Signature algorithms serve Internet protocols and applications such as Transport Layer Security (TLS) [Res26], Certificate Transparency [LMS21], the Domain Name System (DNS) Security Extensions (DNSSEC) [AAL+05], code signing [NMTA22], and firmware updates [MTBM21], some of which send servers many key generation, signing, and verification requests. A Certificate Transparency log verifies the signature chain of each submitted certificate and signs a timestamp [LMS21]. The DNS provider Cloudflare signs DNSSEC responses on the fly at its edge servers [Guð15], reporting 56.9 billion signed DNS record sets per day as early as 2016 [Gra16]. Depending on the application, a remote signing service issues a 10-minute short-term certificate with a fresh key pair per signature transaction [Swi26]. Keys also differ: such a service uses a fresh key per request and verifying a certificate chain uses the public keys of multiple signers, whereas a Certificate Transparency log signs all timestamps with one log key [LMS21].

Such concentrated requests demand high server throughput, so Graphics Processing Unit (GPU) batching has been studied for signatures such as Dilithium [SYD+24, SYLZ25], Falcon [LZS+24, LWS+26], and SPHINCS+ [WDC+25]. MAYO needs the same.

Beullens et al. optimized MAYO for Advanced Vector Extensions 2 (AVX2) and Arm Cortex-M4, Hirner et al. and Sayari et al. for hardware accelerators [BCC+24, HSK+24, SMA+24], and Samyuktha et al. presented a bitsliced Jasmin implementation for central processing units (CPUs) amenable to formal verification [SBR24]. On GPUs, cuMAYO by Chen et al. [CHJ+26] (CHES 2026) targets throughput for KeyGen, Sign, and Verify. It uses the second-round nibble-sliced representation and combines Reuse Matrix Multiplication Evaluation, which reorders matrix products to reuse operands and cut global memory reads, with warp-level kernels and asynchronous multi-stream execution. However, each $\mathbb{F}_{16}$ multiplication looks up a shared-memory table per coefficient, a memory access [CHJ+26, Sys26].

On GPUs, the cost of the $\mathbb{F}_{16}$ matrix products of MAYO depends on how field-element vectors are represented in memory, which has changed across specification rounds: the first round stored the bits of each degree separately (the bitsliced representation), whereas later rounds pack the four bits of an element into a nibble (the nibble-sliced representation) [BCC+25, BCC+26]. According to Nibbling MAYO, the nibble-sliced representation enables fast $\mathbb{F}_{16}$ arithmetic with AVX2 and Arm Neon shuffle instructions, and on the Cortex-M4 the authors kept its performance with the Method of the Four Russians (M4R), which replaces multiplications with precomputed lookup tables [BCC+24].

GPUs, however, lack an instruction like the AVX2 byte shuffle that looks up a 16-entry multiplication table with a nibble index at once; the byte-permute instruction `prmt` of Parallel Thread Execution (PTX) only selects four of the eight bytes in two 32-bit registers.

The word-wide bitwise logic operations of the bitsliced representation, by contrast, run on GPUs: one AND or XOR on a 32-bit word processes the same-degree bits of 32 coefficients. Matrix products consist of multiply-accumulates (MACs), each multiplying 32 coefficients by a scalar and accumulating the products. Excluding per-scalar setup, a MAC takes 35 primitive operations in the bitsliced representation versus 60 in the reference code path of the nibble-sliced representation, a gap that accumulates over all MAYO matrix products.

Since the Volta generation in 2017, NVIDIA GPUs have included Tensor Cores, units dedicated to matrix products [NVI17, NVI20] and introduced to accelerate this central operation of neural network training and inference [NVI17]; later generations added INT8, INT4, and binary (1-bit) precision for inference and FP64 for scientific computing [NVI20]. Prior work has also recast cryptographic operations as Tensor Core matrix products in module-lattice-based PQC including Kyber [WZF+22, ZZF+24, ZZX+25], fully homomorphic encryption [FWX+23, JDW+25, FZZ+25, SFY+26], and sieving algorithms for lattice attacks [DSvW21]. Tensor Cores, however, lack $\mathbb{F}_{16}$ multiplication, so they cannot compute MAYO's matrix products directly. Moreover, one operand of the large matrix products in Sign and Verify is expanded from the key. If we call each operation in a batch an instance, a batch with a different key per instance multiplies a different matrix for each, while a batch of many requests under one key shares one matrix across all instances.

We handle vectors of $\mathbb{F}_{16}$ elements on the GPU in the bitsliced representation and distinguish distinct-key batches, where each instance uses a different key, from same-key batches, where all instances use the same key. For distinct-key batches, our *bitsliced implementation* computes the matrix products of each instance with word-wide bitwise logic operations; for same-key batches, our *tensor path* groups the large Sign and Verify matrix products across the batch and computes them on Tensor Cores.

Contributions. We make four contributions.

- **Bitsliced processing throughout, with outputs identical to the specification.** Both paths keep $\mathbb{F}_{16}$ vectors bitsliced except when reading and writing byte strings such as the compact public key, signatures, and SHAKE256 outputs. We compute the AES-128-CTR-based seed expansion with bitsliced AES [AP21], whose rounds use logic operations instead of lookup tables, and transpose its output directly to $\mathbb{F}_{16}$ bit planes rather than bytes. The Upper operation, E-accumulation, and assembly of the coefficient matrix A also work on bit planes, and solving the linear system runs pivot search, elimination, and back-substitution in the column bitsliced layout with warp votes and bit-plane word operations. The outputs remain identical to the official Known Answer Test (KAT) vectors.

- **Bitsliced multiply-accumulate circuit and matrix product execution.** The bitsliced implementation computes $\mathbb{F}_{16}$ matrix products for distinct-key batches with a bitsliced MAC circuit handling 32 coefficients at once: four byte sign-replication instructions extract four bit masks from a scalar stored in bit planes, and only word-wide AND and XOR perform the multiplication and accumulation, without table lookups. Around it, we choose the thread mapping, mask and input reuse, and traversal order per matrix shape, and load a group's four bit-plane words with one 128-bit instruction.

- **Tensor path for same-key batches.** The tensor path treats each bit plane as a matrix over $\mathbb{F}_2$, turning an $\mathbb{F}_{16}$ matrix product into a sum of 16 binary matrix products between planes on the binary matrix multiply-accumulate instructions of Tensor Cores. Concatenating same-key instances along the batch axis, it computes the Sign and Verify matrix products that share the key-side operand as large binary matrix products over the batch, and builds the key-side input once per key by

Table 1: MAYO parameter sets (round 3) and NIST security levels.

Parameter set	NIST level	n	m	o	k	d
MAYO_1	1	88	80	8	10	0
MAYO_2	1	86	64	13	5	1
MAYO_3	3	118	108	10	11	2
MAYO_5	5	154	142	12	12	2

transposing the key expanded by the bitsliced implementation. To our knowledge, this is the first application of Tensor Cores to MAYO.

- **First GPU implementation of the third-round specification, and its evaluation.** To our knowledge, ours is the first GPU implementation of the third-round MAYO specification [BCC+26]. On an RTX 4090, against cuMAYO [CHJ+26] (second round), the bitsliced implementation achieves 2.75× to 6.73× higher throughput and 6.99× to 34.35× lower single-shot latency in distinct-key batches for KeyGen, Sign, and Verify of all four parameter sets, and the tensor path achieves 13.75× to 30.62× higher Sign and Verify throughput in same-key batches than cuMAYO on distinct-key batches. The bitsliced implementation also achieves 30.58× to 56.78× the throughput of the third-round reference implementation's single-core AVX2 path. On the embedded Jetson AGX Orin and Xavier, it also outperforms cuMAYO in throughput and single-shot latency for every operation and parameter set.

2 Background

2.1 Notation and Element Representation

All arithmetic is over the finite field $\mathbb{F}_{16} = \mathbb{F}_2[x]/(x^4 + x + 1)$, whose element $a = a_0 + a_1x + a_2x^2 + a_3x^3$ has bits $a_t \in \mathbb{F}_2$, called bit t of a; addition XORs bits t pairwise. After the number m of multivariate quadratic polynomials in the public key, we call $\mathbf{b} \in \mathbb{F}_{16}^m$ an m-vector. Whereas the MAYO specification keeps one coefficient matrix per polynomial, we merge the m matrices entrywise into one matrix of m-vectors. With coefficients $\mathbf{b}[i]$ $(0 \le i < m)$, we write $\mathbf{b} = \sum_{t=0}^{3} \mathbf{b}_t x^t$, where the degree-$t$ plane $\mathbf{b}_t \in \mathbb{F}_2^m$ holds bit t of every coefficient: $\mathbf{b}_t[i]$ is bit t of $\mathbf{b}[i]$. Matrix products are built from the MAC acc $\oplus= a \cdot \mathbf{b}$ of a scalar a and an m-vector $\mathbf{b}$, which multiplies each coefficient by a without mixing coefficients.

Matrices split likewise into $X = \sum_{t=0}^{3} X_t x^t$, where the $\mathbb{F}_2$ matrix X_t holds bit t of every entry. A product XY splits into seven partial products $D_t = \bigoplus_{i+j=t} X_i Y_j$ $(t = 0, \ldots, 6)$, and degree reduction with $x^4 = x + 1$, $x^5 = x^2 + x$, and $x^6 = x^3 + x^2$ gives

$$XY = (D_0 \oplus D_4) + (D_1 \oplus D_4 \oplus D_5)\, x + (D_2 \oplus D_5 \oplus D_6)\, x^2 + (D_3 \oplus D_6)\, x^3.$$

The MAC follows the same formula with the bits a_i and the planes $\mathbf{b}_j$ in place of X_i and Y_j. For a square matrix X, $\mathrm{Upper}(X)$ is upper-triangular, keeping the diagonal and holding $X[i][j] + X[j][i]$ at each (i, j) with $i < j$.

2.2 The MAYO Signature Scheme

MAYO is a multivariate quadratic signature scheme of the UOV family over $\mathbb{F}_{16}$ [Beu22, BCC+26]. Its public key is a quadratic map $P : \mathbb{F}_{16}^n \to \mathbb{F}_{16}^m$ of m multivariate quadratic polynomials in n variables, and its secret key is the oil space, an o-dimensional subspace $O \subset \mathbb{F}_{16}^n$ on which P vanishes. In the classic UOV signature scheme, a signature for a

Algorithm 1 MAYO KeyGen, Sign, and Verify, rewritten from [BCC+26, Alg. 4–8]

procedure KeyGen
  $\mathsf{seed_{sk}} \xleftarrow{\$} \mathcal{B}^{\mathsf{sk_seed_bytes}}$
  $\mathsf{seed_{pk}}, O \leftarrow \mathsf{SHAKE256}(\mathsf{seed_{sk}})$
  $P_1, P_2 \leftarrow \mathsf{ExpandP}(\mathsf{seed_{pk}})$
  $P_3 \leftarrow \mathrm{Upper}\big(O^{\mathsf{T}}(P_1 O + P_2)\big)$
  return $\mathsf{csk} = \mathsf{seed_{sk}}$, $\mathsf{cpk} = \mathsf{seed_{pk}} \| P_3$

procedure Verify($\mathsf{cpk}, \mathsf{M}, \mathsf{sig}$)
  $P_1, P_2 \leftarrow \mathsf{ExpandP}(\mathsf{seed_{pk}})$
  $\mathsf{salt}, S \leftarrow \mathsf{Decode}(\mathsf{sig})$
  $t, \Lambda \leftarrow \mathsf{H}(\mathsf{M}, \mathsf{salt})$
  $PS \leftarrow PS^{\mathsf{T}}$
  $U \leftarrow \mathrm{Upper}(S \cdot PS)$
  $y \leftarrow \mathsf{EAcc}(U) + \mathsf{LinTerm}(\Lambda, S)$
  return $y = t$

procedure Sign($\mathsf{csk}, \mathsf{M}, \mathsf{R}$)
  $\mathsf{seed_{pk}}, O \leftarrow \mathsf{SHAKE256}(\mathsf{seed_{sk}})$
  $P_1, P_2 \leftarrow \mathsf{ExpandP}(\mathsf{seed_{pk}})$
  $L \leftarrow (P_1 + P_1^{\mathsf{T}})O + P_2$
  $\mathsf{salt} \leftarrow \mathsf{SHAKE256}(\mathsf{M}, \mathsf{R}, \mathsf{seed_{sk}})$
  $t, \Lambda \leftarrow \mathsf{H}(\mathsf{M}, \mathsf{salt})$
  for ctr from 0 to 255 **do**
    $V_{\mathrm{dec}}, \mathbf{r} \leftarrow \mathsf{SHAKE256}(\mathsf{M}, \mathsf{salt}, \mathsf{seed_{sk}}, \mathsf{ctr})$
    $M \leftarrow V_{\mathrm{dec}} L$
    $P_v \leftarrow P_1 V_{\mathrm{dec}}^{\mathsf{T}}$
    $U \leftarrow \mathrm{Upper}(V_{\mathrm{dec}} P_v)$
    $y \leftarrow t + \mathsf{EAcc}(U) + \mathsf{LinTerm}(\Lambda, V_{\mathrm{dec}})$
    $A \leftarrow \mathsf{BuildA}(M, \Lambda, O)$
    $\mathbf{x} \leftarrow \mathsf{SampleSolution}(A, y, \mathbf{r})$
    break if $\mathbf{x} \neq \perp$
  return $(\mathbf{v}_i + O\mathbf{x}_i,\ \mathbf{x}_i)_{i \in [k]} \| \mathsf{salt}$

target vector $t \in \mathbb{F}_{16}^m$ is a preimage s with $P(s) = t$. The signer sets $s = (\mathbf{v} + Ox,\, x)$ with the secret $v \times o$ matrix O ($v = n - o$), a random vinegar vector $\mathbf{v} \in \mathbb{F}_{16}^v$, and the unknown solution vector $x \in \mathbb{F}_{16}^o$; as P vanishes on the oil space, the terms quadratic in x cancel, and finding a preimage reduces to solving a linear system.

UOV needs $o \geq m$ to have a solution for every target vector, which makes the public key large. MAYO instead takes $o < m$ and, in signing and verification, extends the public map to k input blocks, a technique called whipping. The whipped map $P^* : (\mathbb{F}_{16}^n)^k \to \mathbb{F}_{16}^m$ multiplies P evaluated on each block pair (i, j) by the entry $\mathbf{Z}[i, j]$ of the symmetric $k \times k$ whipping schedule $\mathbf{Z}$ and sums the resulting m-vectors [BCC+26], a summation we call E-accumulation. The entries of $\mathbf{Z}$ are powers of z modulo a public irreducible polynomial $f(z)$ of degree m, with the m elements of an m-vector read as the coefficients of degrees 0 to $m-1$ of the remainder; multiplying by z shifts each coefficient up one position and folds the degree-m coefficient back into the lower positions by $f(z)$. With the vinegar values fixed, $P^*(S) = t$ is a linear system $Ax = y$ in ko oil unknowns, $A \in \mathbb{F}_{16}^{m \times ko}$, which the signer brings to echelon form (EF) by Gaussian elimination and solves by back-substitution, a step we call solving the linear system. As the slack $d = ko - m$ is only 0 to 2, A may be rank-deficient, and the signer then retries with new randomness. Against claw-finding attacks that test the same candidate signature against many messages, the third-round specification adds a linear term to the verification equation, $P^*(S) + \sum_i \mathbf{Z}[i, 0]\, \Lambda s_i = t$, where the linear-term matrix $\Lambda \in \mathbb{F}_{16}^{m \times n}$ is derived from a hash for each message. Table 1 lists the four parameter sets and their NIST security levels.

Algorithm 1 restates MAYO.CompactKeyGen, MAYO.Sign, and MAYO.Verify of the specification, abbreviated as KeyGen, Sign, and Verify, in matrix-product notation. Sign absorbs MAYO.ExpandSK to start from the compact secret key, Verify absorbs MAYO.ExpandPK, byte-string decoding and serialization are omitted, and hashed M is its SHAKE256 digest. The public map consists of the upper-triangular public matrices P_1 ($v \times v$) and P_3 ($o \times o$) and the public matrix P_2 ($v \times o$), forming the $n \times n$ upper-triangular matrix P. In a field of characteristic 2, subtraction is addition. $\mathsf{ExpandP}$ generates P_1 and P_2 with AES-128 CTR keyed by $\mathsf{seed_{pk}}$; from the message and the salt, H derives t with SHAKE256 and $\Lambda = [\Lambda_v\ \Lambda_o]$ with AES-128 CTR. The rows of the decoded vinegar-value matrix V_{dec} ($k \times v$, Sign) and the signature matrix S ($k \times n$, Verify) are the vinegar vectors $\mathbf{v}_i^{\mathsf{T}}$ and the signature blocks $\mathbf{s}_i^{\mathsf{T}}$, respectively. $U[i, j]$ is P evaluated on the block pair (i, j), and $\mathsf{EAcc}(U) = \sum_{i \leq j} \mathbf{Z}[i, j]\, U[i, j]$ is the E-accumulation. $\mathsf{LinTerm}$ is the linear term above, with $\mathbf{v}_i$ and Λ_v for $\mathbf{s}_i$ and Λ in Sign. BuildA adds $\Lambda_v O + \Lambda_o$ to the first row M_0 of M and then, for each block pair $i \leq j$, adds $\mathbf{Z}[i, j] M_j$ to column block i of the coefficient matrix A and, if $i \neq j$, $\mathbf{Z}[i, j] M_i$ to column block j. $\mathsf{SampleSolution}$ solves the linear system.

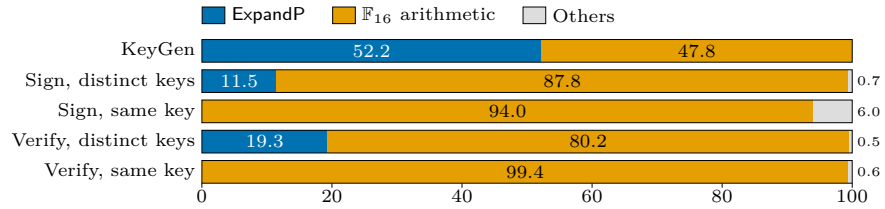


Figure 1: Per-step share of execution time (%) of the GPU port for MAYO_3. $\mathbb{F}_{16}$ arithmetic covers Upper, EAcc, LinTerm, BuildA, and the matrix products; Others covers SHAKE256, H, SampleSolution, and the rest.

We ported the third-round reference implementation to the GPU with one thread per instance and call it the GPU port. Figure 1 shows its per-step execution-time shares for the representative MAYO_3 on an RTX 4090 under Windows Subsystem for Linux 2 (WSL2) at a batch size of 4,096. Each thread accumulated cycle-counter differences at every step boundary, and we report the median share over three measurements. Each instance of a distinct-key batch expands its own compact key, whereas a same-key batch reads one key expanded before the measurement. ExpandP was the largest step of KeyGen at 52.2%. In distinct-key batches, $\mathbb{F}_{16}$ arithmetic took 87.8% of Sign and 80.2% of Verify, followed by ExpandP at 11.5% and 19.3%; in same-key batches, $\mathbb{F}_{16}$ arithmetic took 94.0% and 99.4%, and the matrix products alone 92.2% of Sign. $\mathbb{F}_{16}$ arithmetic was also the largest step of Sign and Verify for the other parameter sets, except distinct-key Verify of MAYO_2, led by ExpandP at 64.0%. We therefore target the $\mathbb{F}_{16}$ arithmetic and ExpandP of distinct-key batches and the matrix products of same-key batches, where key expansion drops out.

2.3 GPU Execution and Memory Model

In the CUDA programming model [NVI26a], a kernel launch starts thread blocks, each running to completion on one streaming multiprocessor (SM). Threads of a block exchange values through shared memory and order accesses with barriers; blocks run in unspecified order, and an ordinary kernel launch cannot use grid-wide synchronization. Threads run in warps of 32 that execute the same instruction together, the Single Instruction, Multiple Threads (SIMT) model, and divergent branches in a warp run one after another. Threads (lanes) of a warp exchange register values with shuffle instructions and combine predicates with warp vote instructions. The integer unit executes AND, XOR, and population count, which counts the 1 bits, on a 32-bit word in one instruction each.

Registers are private to a thread, and shared memory is on-chip SM memory valid only while its block runs. Global memory is large but has long latency, goes through the L2 cache, and persists for later kernels to read. Since a block holds its resources until it finishes, the resident block count, the most blocks an SM holds at once, is set by whichever per-SM limit on registers, shared memory, warps, or blocks is hit first. Many resident warps let the SM hide latency by running other warps while one waits for memory.

Tensor Cores are dedicated matrix-product units in the SM. With the binary matrix multiply-accumulate (BMMA) instruction, a warp's 32 threads jointly multiply two binary matrices and accumulate the product, with input and result fragments spread over their registers. Its AND variant, available from the Ampere generation onward [NVI20, NVI26b], ANDs corresponding elements and adds the population count of the result to an integer accumulator. The K dimension of a matrix product is the one summed out.

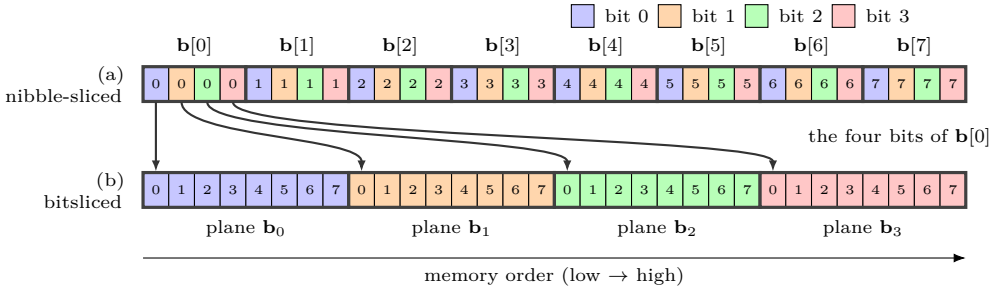


Figure 2: The same eight coefficients (32 bits) in both representations; cell numbers are coefficient indices i, and colors are bits t.

3 Choosing the Representation: Nibble-Sliced versus Bitsliced

We compare the two storage representations of an m-vector by MAC operation count and memory layout and select one for the GPU. Figure 2 shows both memory layouts of the same eight coefficients. The nibble-sliced representation stores each coefficient $\mathbf{b}[i]$ as one contiguous nibble, coefficient after coefficient; the bitsliced representation stores the bit planes $\mathbf{b}_0, \ldots, \mathbf{b}_3$ separately, spreading each coefficient over four planes and keeping same-degree bits adjacent. The MAYO specification has used the nibble-sliced representation since the second round, including the third [BCC+24, BCC+25, BCC+26].

3.1 Operation Count of the Multiply-Accumulate

We compare arithmetic cost by the operation count of one MAC over four words of equal width, which hold 32 coefficients either way: a 32-bit word holds eight nibble-sliced coefficients or the same-degree bits of 32 bitsliced coefficients, and we call these four bit-plane words a group. We count primitive operations, before the compiler merges instructions, and only the per-word part of the MAC, excluding the setup that expands a scalar into multiplication-table entries or masks, shared by all words it multiplies.

The reference code path for the nibble-sliced representation first computes four multiplication-table entries $\tau_t = a\,x^t$ $(t = 0, \ldots, 3)$ from the scalar a. For each word w, it shifts bit t of every nibble to the least significant position of the nibble and masks the other bits with AND, giving a temporary vector e_t with only bit t of each coefficient, which splits the vector into bit planes every time. Integer multiplication by τ_t puts τ_t in each nibble of e_t whose bit is 1 and 0 elsewhere, and XORing the four products into the accumulator completes the MAC for the word (Figure 3(a)). A word thus takes four ANDs, three shifts, four integer multiplications, and four XORs, so four words take 60 operations: 16 ANDs, twelve shifts, 16 integer multiplications, and 16 XORs. The reference implementation uses 64-bit words, but the per-word count is independent of word width.

Already split into planes, the bitsliced representation needs neither this splitting nor integer multiplication, and one AND or XOR covers all 32 coefficients of a group. Splitting the scalar into bits gives $a \cdot \mathbf{b} = \sum_{t=0}^{3} a_t\,(x^t \cdot \mathbf{b})$, and since $x^4 = x + 1$ makes each output plane of $x^t \cdot \mathbf{b}$ an XOR combination of input planes, no separate degree reduction step is required. For the bit-plane words $i_0, \ldots, i_3$ of a group, three shared intermediate values $\alpha = i_0 \oplus i_3$, $\beta = i_3 \oplus i_2$, and $\gamma = i_2 \oplus i_1$ produce all four combinations. Each bit a_t becomes a mask m_t replicated across the word. As the scalar matrix is also stored in bit planes, a register word holds the bits of eight scalars, one byte per plane, with the current scalar's bits in the most significant position of each byte; four byte sign-replication instructions, each replicating the most significant bit of one byte across the word, extract $m_0, \ldots, m_3$,

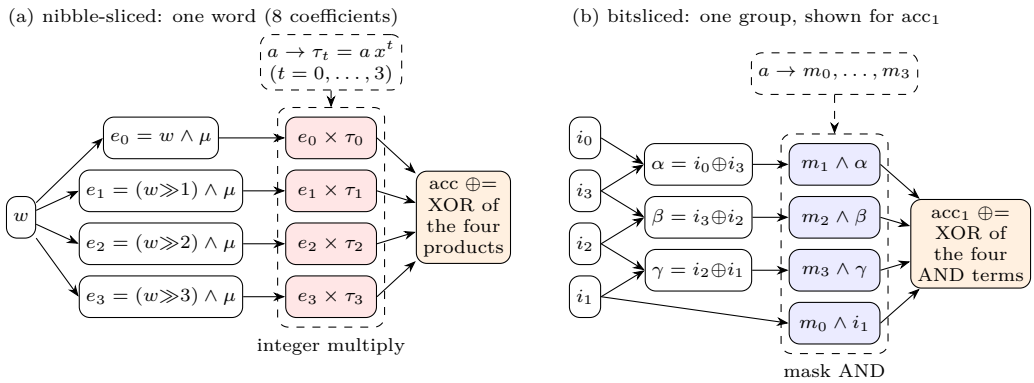


Figure 3: MAC acc $\oplus= a \cdot \mathbf{b}$ on 32 coefficients. (a) One of four nibble-sliced words, where $\mu =$ `0x11111111` keeps the lowest bit of each nibble; four words take 60 operations (16 AND, twelve shift, 16 integer-multiply, 16 XOR). (b) Bitsliced circuit for one group: 35 operations (16 AND, 19 XOR). Both counts exclude the once-per-scalar setup.

and a 1-bit shift moves to the next scalar. In Figure 3(b), the four accumulators take four ANDs and four XORs each and the shared values three more XORs, so the per-word part of the MAC for one group takes 35 operations: 16 ANDs and 19 XORs.

This 25-operation difference, the twelve shifts and 16 integer multiplications per MAC of the nibble-sliced representation against three extra XORs, does not alone determine overall performance, which also depends on memory access, padding, and thread mapping. Moreover, the nibble-indexed 16-entry table lookup behind the choice of the nibble-sliced representation in the specification [BCC+24] is not a single instruction on the GPU. Given the operation count and this table-lookup constraint, our implementation stores and computes m-vectors in the bitsliced representation.

3.2 Storage Layout and Padding

We cut each plane into 32-bit words, dividing an m-vector into $\lceil m/32 \rceil$ groups of 32 coefficients: $3, 2, 4, 5$ groups for MAYO_1, MAYO_2, MAYO_3, and MAYO_5. Group g stores its four words, one per plane, at consecutive addresses, and bit r of its degree-t word is $\mathbf{b}_t[32g + r]$, so a single 128-bit load fetches one group for the MAC circuit.

When m is not a multiple of 32, the upper $32\lceil m/32 \rceil - m$ bits of the last group ($16, 0, 20, 18$ bits in parameter-set order) hold no coefficient, so an m-vector occupies 48, 32, 64, and 80 bytes, no less than the 40, 32, 56, and 72 bytes of the reference implementation, whose nibble-sliced representation holds 16 coefficients per 64-bit word. AND and XOR do not mix bit positions, so padding bits never spread into coefficient bits, but E-accumulation reads positions that multiplication by z pushes to degree m or higher as coefficients and folds them back into the lower positions by $f(z)$. Since the byte strings of the specification also concatenate m-vectors without gaps, cutting them into groups puts coefficients of the next m-vector into the padding bits. Our implementation therefore clears the padding bits to 0 when filling m-vectors from the input, and AND and XOR keep them 0.

4 Bitsliced Implementation of MAYO

Our bitsliced GPU MAYO KeyGen, Sign, and Verify run one instance per thread block and keep m-vectors in bit planes from the AES output of key expansion to the solver.

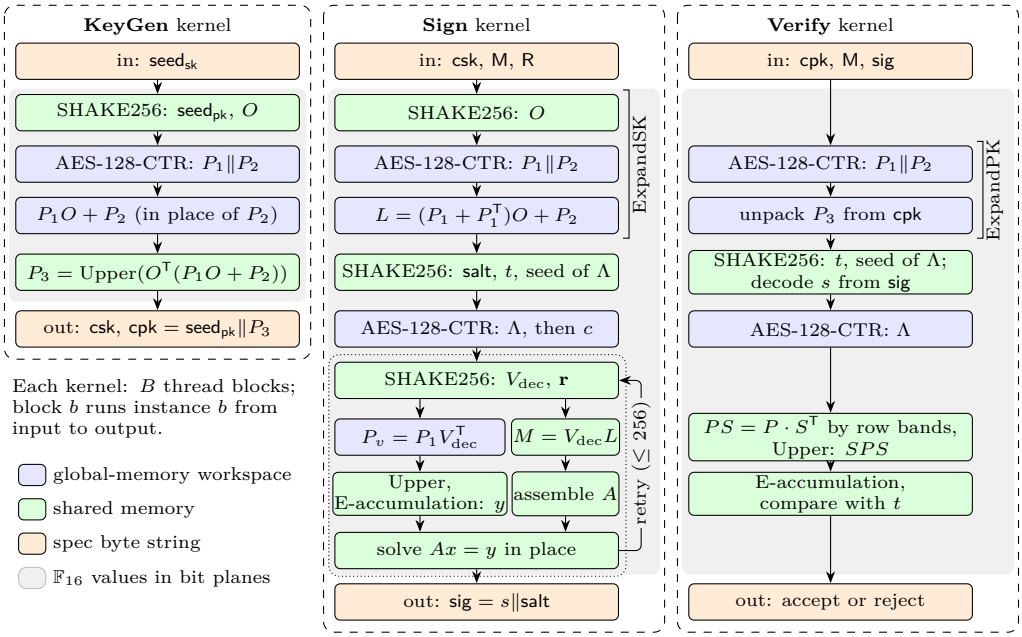


Figure 4: KeyGen, Sign, and Verify kernels of the bitsliced implementation.

4.1 Design: One Thread Block per Instance

Batch instances share no intermediate values or results, so an instance can map to a thread, a warp, or a thread block, depending on its working-set size and on how many threads can share its work. The working set is dominated by the expanded $P_1 \| P_2$, whose $v(v+1)/2+vo$ m-vectors take 116,800 bytes (MAYO_2) to 948,560 bytes (MAYO_5) in bitsliced form, exceeding the per-block shared memory of the target GPUs. We therefore keep $P_1 \| P_2$ and the large Sign products L and P_v in a per-instance global-memory workspace, and keep in shared memory the values later steps reread.

An instance also has many independent tasks (1,920 when L of MAYO_1 Sign is split by groups). Once threads share them, each next step gathers values from other threads: E-accumulation and assembling A read M, and the solver reads A. The shared memory and barriers for this exchange exist only within a block, so each instance gets one thread block, the largest unit that can share it. Packing several thread- or warp-level instances into a block would multiply its shared memory by the instance count. The number of threads per instance is the block size; one warp per instance is the candidate block size 32.

cuMAYO, whose Sign and Verify take compact keys, splits key expansion (ExpandSK, ExpandPK) and the operation into two kernels [CHJ+26, Sys26]. We instead run each block from key expansion through the operation in one kernel; Figure 4 shows the steps of each kernel and whether each result stays in the workspace or shared memory. KeyGen, as in cuMAYO, is a single kernel with one block per instance. Kernel fusion removes only the kernel boundary: the expanded $P_1 \| P_2$ still goes through the workspace.

With one instance per block, the resident block count per SM, set by per-instance shared memory and the register total of a block, equals the number of concurrent instances per SM. A larger block adds resident warps but can reduce resident blocks through its register total, and its barriers synchronize more warps. To choose the block size, we compared the throughput of predefined candidate block sizes (multiples of the warp size) and resident-block targets per SM, each combination at the largest batch reached within the memory limit; a target makes the compiler fit the per-thread register cap to it. The batch size B is the number of operation instances processed at once. The default block

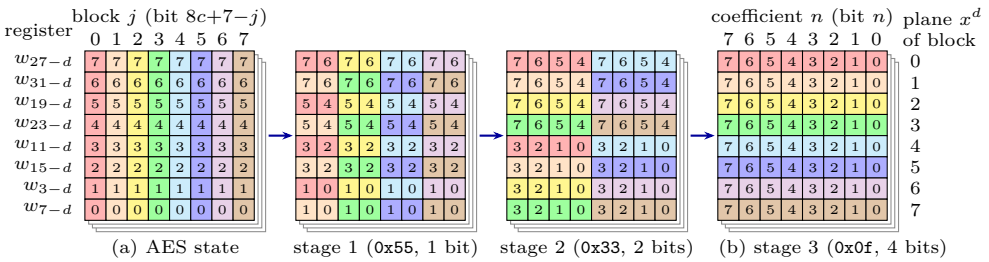


Figure 5: The 8×8 bit transpose moving the barrel-shiftrows AES state directly to $\mathbb{F}_{16}$ bit planes (plane x^d, byte position $c = 0$). Cell colors give the AES block index j, and cell numbers the coefficient index n.

size is 128, with 256 only for MAYO_3 KeyGen and MAYO_3 and MAYO_5 Sign; only MAYO_2 Verify specifies a target, of eight resident blocks per SM.

4.2 Feeding AES Output Directly into Bit Planes

AES-128 CTR for $P_1 \| P_2$ uses the barrel-shiftrows implementation of Adomnicai and Peyrin [AP21], whose round function is a table-free bitwise circuit, so no block loads round tables into shared memory and no bank access depends on byte values. Each thread computes eight AES blocks in a state of 32 32-bit words. Word w_{8r+7-b} holds bit b of the bytes in state row r; in it, the byte in state column c occupies the byte lane of bits $8c$ to $8c+7$, whose bit $8c+7-j$ belongs to block j. We replace its final transpose back to bytes with one into $\mathbb{F}_{16}$ bit planes, so the AES output lands directly at its positions in the bit planes.

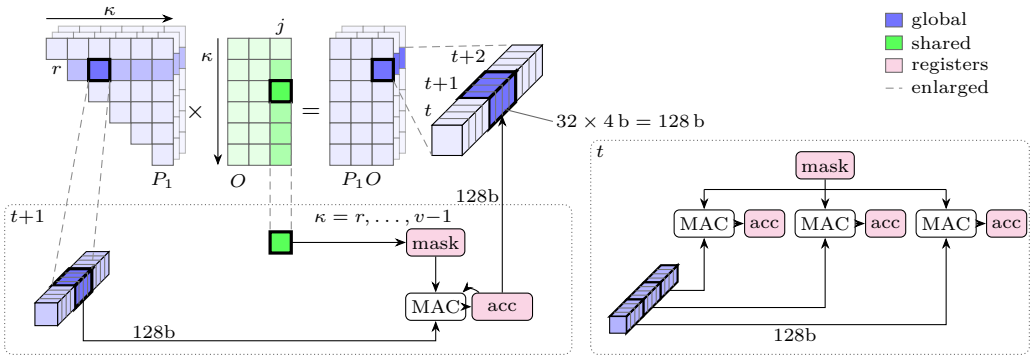
Byte $q = 4c + r$ of block j holds coefficients $2q$ and $2q+1$ in its low and high nibbles. For nibble position e, coefficient bit d is byte bit $4e + d$, which after the last round sits at bit $8c + 7 - j$ of word $w_{8r+7-d-4e}$. Plane x^d thus uses only the eight words holding byte bits d and $d+4$; these sets are disjoint, so each plane rearranges its own words in place.

Within the byte lane at position c, this is an 8×8 bit transpose. With the eight words of plane x^d as rows and the bits of the byte lane as columns, rows are indexed by word (r, e) and columns by block j after the last round (Figure 5(a)). The bit must move to bit $n = 2q + e = 8c + 2r + e$ of the plane-x^d word of block j, so rows must become blocks j and columns coefficient positions $2r + e$ (Figure 5(b)).

We compute it with three stages of SWAPMOVE, which swaps mask-selected bits of two words at a fixed distance [AP21]; barrel-shiftrows uses the same circuit to return the state to bytes, and only the eight input words per plane change. At four SWAPMOVEs per stage, the four planes of eight blocks take 48. As the masks repeat per byte, each 32-bit operation acts on four byte positions at once. The in-place transpose needs no second copy of the state, and the planes of each block are just register selections.

When m is a multiple of 32, leaving no padding bits (only MAYO_2, Table 1), the transpose result goes straight to its destination. In the other three, 32-coefficient groups do not align with 16-byte AES blocks, so the result is staged in a contiguous keystream chunk in shared memory, from which, after a barrier, the threads of the block copy their destination groups. Neither path uses a separate keystream buffer in global memory.

The linear-term matrix Λ uses the same AES circuit and output paths; XORing the specification's initial value iv into the round-0 key lets the counter assembly and round circuit be reused unchanged. Since Λ changes per message, it resides in the per-instance workspace, and Sign expands it once, outside the retry loop. Verify and Sign start the first-column accumulators of PS and P_v, respectively, from the matching columns of Λ, so $S \cdot PS$ and $V_{\text{dec}} P_v$ also add Λs_i and $\Lambda_v v_i$ to block pair $(i, 0)$. The term $c = \Lambda_v O + \Lambda_o$ for



(a) chunk width $w = 1$ (b) chunk width $w = 3$

Figure 6: Accumulation of P_1O in MAYO_1 KeyGen (initial value P_2 omitted). Stacked layers mark m-vector entries; the prism shows one m-vector unfolded. (a) Chunk width 1: thread $t{+}1$ accumulates its own group per κ. (b) Chunk width 3: one thread applies one mask set to three groups.

the first block of M is computed once per message, XORed in at every retry, and, as it depends on the secret O, zeroized when Sign terminates.

4.3 Matrix Products: Implementation and Execution Structure

The KeyGen product $P_1O + P_2$, the Sign matrices $L = (P_1 + P_1^{\mathsf{T}})O + P_2$ (from ExpandSK), $P_v = P_1 V_{\text{dec}}^{\mathsf{T}}$, and $M = V_{\text{dec}} L$ (the last two per signing attempt), and the Verify product $PS = P \cdot S^{\mathsf{T}}$ all multiply a large matrix of m-vectors by a small matrix of $\mathbb{F}_{16}$ scalars. We call the summed index the K index, κ. Each output m-vector accumulates scalar–m-vector MACs along κ; outputs and their groups are independent. As P_1 and P_3 are upper triangular, the number of K indices varies by output row in $P_1O + P_2$, P_v, and PS.

Threads of a block take (output m-vector, group chunk) pairs. With w groups per chunk (the chunk width), an output splits into $n_c = \lceil m/32 \rceil / w$ chunks; chunk g of output e has pair index $p = e\, n_c + g$, and with block size T, thread t takes $p = t, t + T, t + 2T, \ldots$. In MAYO_1 KeyGen, for example, the 640 output m-vectors of $P_1O + P_2$, three width-1 chunks each, give 1,920 pairs, 15 per thread at $T = 128$. Figure 6(a) shows three neighboring threads accumulating the three groups of one P_1O entry along κ; each prism segment is a group, split by thin lines into four 32-bit bit-plane words. A thread keeps its accumulators in registers over all K indices and writes once; they start from the same-position entry of P_2 for $P_1O + P_2$ and L, from Λ for the first column of PS and P_v, and from 0 otherwise. The large matrix comes group by group from global memory, and the scalar matrices O, V_{dec}, and S from shared memory.

Verify builds PS in shared memory eight rows at a time; after a barrier, it computes their contribution to $SPS = S \cdot PS$, applies Upper, accumulates it, and moves on.

With one thread per group, threads sharing a scalar each build masks; a thread with several groups reuses one mask set for its chunk. The chunk width divides each parameter set's group count, so no narrower last chunk causes warp divergence. Wider chunks raise mask reuse but leave fewer threads per output m-vector, lowering parallelism. In Sign, one thread takes a whole output m-vector, while KeyGen and Verify use width 1, except MAYO_2 KeyGen (width 2). Figure 6(b) shows one mask set serving a width-3 chunk.

For L, P_v, and M of Sign and the P_3 rows of PS in Verify, a thread refills its register word every eight κ steps, gathering 32 bits of eight scalars from bit-plane words in shared memory. For O of KeyGen and S for the P_1 and P_2 rows of PS, the block builds once a register-word copy in shared memory that threads use directly, a word per eight κ steps.

Pair indices vary the chunk g fastest and run over outputs e row-major, so in the width-1 path neighboring threads of a warp take neighboring chunks of one output, reading neighboring groups of one input m-vector; in Sign, with one chunk, they take neighboring columns of a row. In P_1O+P_2, L, P_v, and PS, outputs in a row read the same large-matrix m-vectors in the same order, differing only in scalars, so same-chunk threads in different columns of a row read the same address at the same κ. In MAYO_1 KeyGen, a row has $o \times 3 = 24$ pairs, so a warp spans two or three rows and a warp load touches two or three 48-byte m-vectors. In $M = V_{\text{dec}}L$, neighboring threads read neighboring, storage-adjacent columns in a row of L. Each group is read with one 128-bit load (w loads per κ for width w) and written with one 128-bit store; these are the two 128b arrows in Figure 6.

Since products accumulate into one output, the width-1 path (Verify's PS and, except for MAYO_2, KeyGen's $P_1O + P_2$) defers reduction to four planes until just before the store, keeping partial products $D_0, \ldots, D_6$ in seven accumulators, three more than output planes. A product adds the AND of mask m_a and input bit-plane word i_b into D_s with $a + b = s$: 16 ANDs and 16 accumulating XORs, without the three shared intermediate values the standard circuit computes per product. After this unreduced accumulation (lazy reduction), the degree-reduction identities fold D_4, D_5, D_6 into the lower degrees once, yielding the output group's four bit-plane words. The acc in Figure 6(a) is $D_0, \ldots, D_6$; KeyGen combines the reduced value with the P_2 initial value before writing.

Traversal follows matrix storage: the matrix P_1 is in packed storage (upper-triangular entries consecutive in row order), so P_1O of KeyGen and P_v of Sign scan a contiguous packed range per output row. Symmetrization makes L read both row r and column r, so its traversal splits into the K indices below r, down column r with a stride shrinking by one per step, and those above r, along row r with stride 1; the packed index formula is evaluated once per range, and the address then advances by the stride. The diagonal entry (r, r) appears twice in $P_1 + P_1^{\mathsf{T}}$ and cancels, so it is not read.

4.4 Keeping Bit Planes through the Remaining Operations

The subsequent Upper operation, E-accumulation, assembly of A, and solver read the matrix-product bit planes directly rather than converted coefficient arrays, so the representation never changes between steps. In E-accumulation, multiplying by z shifts each plane left by 1 bit, carrying bits across group boundaries into the next word; the coefficient overflowing into degree m is multiplied by the public coefficients of $f(z)$ and added back into the lower plane bits. The coefficient matrix A is assembled likewise by gathering, on the bit planes, the columns of M multiplied by $\mathbf{Z}[i, j]$; the assembled array holds one bit-plane m-vector per column, plus a last column for the right-hand side.

The final E-accumulation result also stays in bit planes: Sign stores the right-hand side y (that result plus t) for the right-hand-side column above, and Verify compares the result with t directly. The solver keeps the column bitsliced layout from the assembly of A through the rank check and back-substitution. Keeping the elimination semantics of the reference implementation (a search window over a public range, mask-based pivot selection, and XOR-accumulated row elimination), we reimplemented pivot search, elimination, and back-substitution with warp votes and bit-plane word operations.

The representation returns to nibble-sliced form only in the final steps writing specification-defined byte strings: the Upper result of P_3 into the compact public key in KeyGen and the completed S into the signature in Sign, matching the official KAT vectors. On input, only O, V_{dec}, and t from the SHAKE256 output, the solution's random initial value, the signature S in Verify, and P_3 of the compact public key are converted to bit planes.

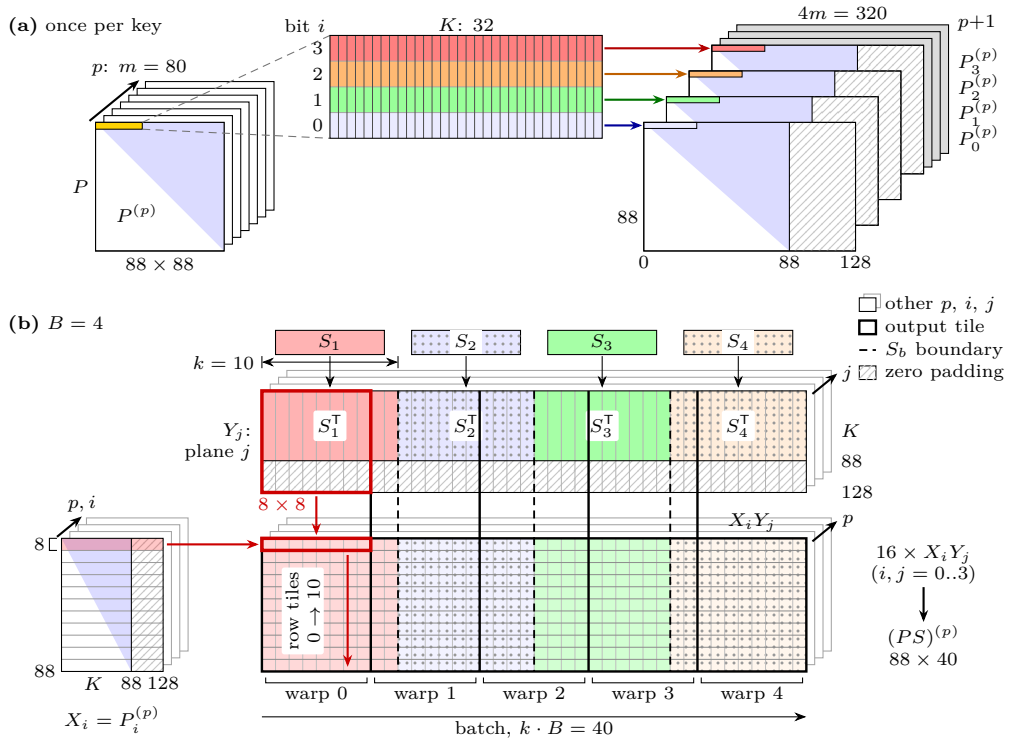


Figure 7: Verify $PS = P \cdot S^\mathsf{T}$ on the same-key tensor path (MAYO_1, $B = 4$). (a) Preparing key-side binary matrices $P_i^{(p)}$. (b) Multiplying one of them by the signature matrices concatenated along the batch axis.

5 Tensor-Core Path for Same-Key Batches

This section presents the tensor path, which computes the matrix products of Sign and Verify on same-key batches with Tensor-Core BMMA instructions.

5.1 Tensor-Core Binary Products and the Same-Key Condition

Tensor-Core integer matrix products cannot directly compute $\mathbb{F}_{16}$ matrix products: $x \cdot x^3$ (integer values 2 and 8) is 0 modulo 16 but $x^4 = x + 1$ (integer value 3) in $\mathbb{F}_{16}$. The bitsliced representation avoids this problem: each plane is already a matrix over $\mathbb{F}_2$, so BMMA directly computes the binary matrix products between planes into which an $\mathbb{F}_{16}$ matrix product splits. In MAYO, the key-side matrix entries are m-vectors, so the tensor path splits each of the m per-polynomial $\mathbb{F}_{16}$ coefficient matrices into four planes, handling $4m$ binary matrices. We write $P^{(p)}$ for the coefficient matrix of polynomial p, $P_i^{(p)}$ for its plane-i binary matrix, and X_i and Y_j for the planes of the key-side operand and the other operand of a product. Bits are packed along the axis opposite to that of the bitsliced implementation. Its word, called a SIMT word, holds the same-degree bits of 32 polynomials of one m-vector, whereas BMMA sums out bits along the K dimension with AND and population count, so a tensor-path word, called a BMMA word, holds the same-degree bits of one polynomial from 32 entries adjacent along the K dimension. In Figure 7(a), bit i of 32 adjacent entries in a row of $P^{(p)}$ forms one BMMA word of $P_i^{(p)}$.

The partial products $D_0, \ldots, D_6$ collect the 16 plane products X_iY_j by degree, so one coefficient-matrix product takes 16 binary matrix products (Figure 7(b), right). The AND variant of BMMA yields an integer whose least significant bit is the inner product over

$\mathbb{F}_2$; parity being linear, all products of one D_t share one integer accumulator, and this bit is extracted once at the end. With at most four inner products of length at most n, an accumulator stays at most $4n \leq 616$, within its 32 bits. Degree-reduction identities then add the extracted D_4, D_5, D_6 into $D_0, \ldots, D_3$, giving the four output planes.

We use the PTX instruction `mma.sync.aligned.m8n8k128.row.col.s32.b1.b1.s32.and.popc` [NVI26b], which multiplies eight rows of X_i by eight columns of Y_j over 128 bits of the K dimension, accumulating into an 8×8 output tile (highlighted in Figure 7(b)). Rows are thus zero-padded to a multiple of 8, and the K dimension to a multiple of 128 bits. Only MAYO_5 has two K tiles; the others, with K-dimension length (n or v) at most 128, have one. Per polynomial, each output tile takes 16 binary matrix products per K tile. As BMMA skips only whole 128-bit tiles, the upper-triangular P and P_1 save only the first K tile of MAYO_5 row tiles from row index 128 on.

In a same-key batch, all instance products merge into one large product of a single key-determined matrix and the vectors of the whole batch. In $PS = P \cdot S^\mathsf{T}$ of Verify, the key-determined P is shared by the B instances and only S differs per signature. Concatenating $S_1^\mathsf{T}, \ldots, S_B^\mathsf{T}$ ($n \times k$ each) column-wise into an $n \times kB$ matrix and multiplying each X_i by its planes Y_j as above yields PS for all B instances at once; we call this column direction the batch axis (Figure 7(b)). Sign likewise merges the two products involving P_1 and L, which the secret key determines: $P_v = P_1 V_{\text{dec}}^\mathsf{T}$ concatenates the per-instance $V_{\text{dec}}^\mathsf{T}$, as does $M = V_{\text{dec}} L$ rewritten as $M^\mathsf{T} = L^\mathsf{T} V_{\text{dec}}^\mathsf{T}$. The kB batch-axis columns form eight-column output tiles distributed among warps that all read the same key-side binary matrices. In Figure 7, MAYO_1 ($n = 88$, $m = 80$, $k = 10$) with $B = 4$ signatures has 40 output columns in five output column tiles; as $k = 10$ is not a multiple of 8, the second through fourth tiles each hold columns of two signatures. An output column depends only on the key-side matrix and its own input, so under the same key, sharing a tile leaves each signature's result identical to a separate computation. Multiplied separately, each signature would use $\lceil 10/8 \rceil = 2$ column tiles, six columns of the second left empty, so four would need eight tiles; concatenation fills all 40 columns of five.

The tensor path first lays out the key-side matrices for BMMA input, unfolding P, or P_1 and L, into per-polynomial, per-plane binary matrices ($n \times n$, $v \times v$, and $o \times v$ for P, P_1, and L^T) and zero-filling lower triangles of upper-triangular ones, a step the bitsliced implementation lacks. In a same-key batch, this runs once for all batch calls with that key.

Both paths perform the same key expansion from the seed. For a batch with U distinct keys, $U = 1$ gives a same-key batch and $U = B$ a distinct-key batch, whose B instances use B distinct keys. KeyGen, with a new key per instance, always has $U = B$.

When keys differ across instances, the tensor-path overhead grows in two places. First, the key-side binary matrices must be prepared per instance, and in workloads in which each request uses a different key, the next call need not reuse the key, so this preparation cannot be amortized over calls either. Second, no batch axis forms: columns under different keys cannot share an output tile, so each instance's product has only k columns (one output column tile for MAYO_2, two for the others) yet reads the whole key-side binary matrix. Our tensor path therefore covers Sign and Verify on same-key batches ($U = 1$); the bitsliced implementation handles distinct-key batches.

5.2 From AES and Hash Outputs to BMMA Operands

These key-side operands originate from the AES output, which key preparation writes into $P_1 \| P_2$ in bit planes without passing through bytes, using the bitwise AES circuit and transpose of the bitsliced implementation; Verify also unpacks P_3 of the compact public key into bit planes, and Sign computes $L = (P_1 + P_1^\mathsf{T})O + P_2$ on them. For a fixed plane, 32 adjacent entries in a key-side matrix row and 32 polynomials form a 32×32 bit block, whose transpose gives the BMMA inputs, each output word gathering one bit from each of 32 source words. We do not write the AES output directly in the BMMA input layout:

a BMMA word takes bits from 32 entries far apart in the keystream, so it is complete only after all $16m$ bytes of keystream are produced, not per AES block, and this transpose already belongs to key preparation, so little would be gained.

Per-instance operands are built for every batch. One block per instance computes the target vector t and the vinegar vectors as in the bitsliced implementation. The signature blocks s_i of Verify and the matrix V_{dec} of Sign are unpacked from the signature or SHAKE256 output into one-coefficient-per-byte arrays, then converted by packing degree-d bits into words along the K dimension and concatenating instance columns along the batch axis. As Sign's P_v and M share K dimension v, one V_{dec} conversion serves both.

5.3 Consuming BMMA Outputs in Bit Planes

BMMA results feed the Upper operation, E-accumulation, assembly of A, and solver. Degree reduction of a BMMA output fragment gives one polynomial's $\mathbb{F}_{16}$ value for one output entry; taking the 32 polynomials in turn, the tensor path places each value's degree-d bit at that polynomial's position in the plane-d word. As in the bitsliced implementation, each output entry's m-vector emerges as four bit-plane words per 32-coefficient group, which Sign writes directly to the per-instance P_v and M buffers with no intermediate byte array or layout conversion. Later steps run on bit planes, one block per instance.

Verify does not even write PS to global memory, avoiding a round trip for its nk m-vectors per instance before the next product $S \cdot PS$. Instead, one warp computes all row tiles of an output column tile in turn (Figure 7(b)), multiplies each PS value in bit-plane words by the signature coefficient $s_i[r]$ of its row, and accumulates $H_{ij} = \sum_r s_i[r] \cdot PS[r][j]$ in registers. This bitsliced MAC uses no Tensor Cores; each lane obtains the PS entries of the eight rows in its column through warp shuffles and multiplies them itself, so no partial sums are regathered across lanes. Only H (k^2 m-vectors per instance, k/n the size of PS) is written out; a final step completes the Upper operation of SPS via $H_{ij} \oplus H_{ji}$ $(i < j)$ and H_{ii}, then performs E-accumulation and compares with t.

In the third round, the linear term differs per message, so the tensor path adds it in per-instance steps after, not inside, the batch-shared products. Verify, which does not write out PS, precomputes Λs_i in bit planes per instance and adds it to entry $(0, i)$ of the Upper result. Sign expands Λ per instance, adding its columns to the first column of the P_v written by BMMA and $\Lambda_v O + \Lambda_o$ to the first block of M. Retries also differ per instance, so the tensor path covers only the first signing attempt: an instance whose A is then rank-deficient only sets a retry flag, and the host re-signs such instances with the bitsliced implementation from the second attempt onward. A retry samples fresh vinegar vectors, so first-attempt product results are not reused.

6 Security and Correctness Scope

For the bitsliced implementation and the tensor path, we state whether their outputs match the specification and how far they protect secret values.

Correctness. For all four parameter sets, the KeyGen and Sign outputs of the bitsliced implementation match the official KAT vectors bit for bit, and Verify accepts the official signatures. As these vectors alone exercise the retry path insufficiently, we added a bit-exact cross-check against the CPU reference implementation on the same inputs, including message-length boundary values and searched retry inputs fixed so that the reference implementation fails its first attempt; on every input, Verify accepts our signature and rejects it with one byte changed. For the same-key tensor path, official KAT vectors grouped into same-key batches yield matching signatures that Verify accepts, and intermediate and final values were compared with the bitsliced implementation on random-input batches.

Constant-time design. By *constant-time* we mean that no branch or memory address depends on secret values, with the number of retries public as in the reference implementation. cuMAYO notes that its lookups into the $\mathbb{F}_{16}$ multiplication table, AES T-tables, and S-box

Table 2: Measurement devices, toolchain, and conditions. CC denotes the CUDA compute capability; L4T denotes NVIDIA Linux for Tegra.

Setting	RTX 4090	Jetson AGX Orin	Jetson AGX Xavier
GPU	Ada, CC 8.9, 128 SMs, 24 GB, 101,376 B shared memory/block	Ampere, CC 8.7, 14 SMs, 32 GB shared with CPU	Volta, CC 7.2, 8 SMs, 32 GB shared with CPU
Host	Intel Core i7-13700K	L4T R35.4.1, 8-core Arm	L4T R35.4.1, 8-core Arm
Driver; CUDA	595.91.07; 13.3	L4T built-in; 11.4	L4T built-in; 11.4
Our build flags	`-arch=sm_89 -O2`	`-arch=sm_87 -O2`	`-arch=sm_72 -O2`
GPU clock	2,520 MHz (boost)	930 MHz (max)	1,377 MHz (max)
Throughput B	$\{2^{10}, 2^{12}, 2^{14}, 2^{16}\}$		
Latency B	1 (all GPUs); batch completion latency $\{1, 8, 64, 256, 2^{10}, 2^{12}, 2^{14}\}$ (RTX 4090)		

make value-dependent memory accesses, possibly observable under stronger GPU attack models via cache behavior or shared memory bank conflicts, and leaves countermeasures to future work [CHJ+26]. We multiply in $\mathbb{F}_{16}$ without a table, expanding scalar bits into masks applied with AND and XOR, so coefficients of the secret matrix O and the vinegar vectors never become memory addresses. The AES round function is likewise bitwise logic without table lookups; only the AES key schedule keeps an S-box lookup, which can reveal only public values because MAYO's AES keys, inputs, and outputs are all public [BCC+26]. The solver searches a window over a public range and selects pivots with masks, so pivot positions never serve as branch conditions or addresses.

Machine-code audit for constant-time verification. To check that this design survives compilation, we statically audited the 76 kernels of both paths in machine code (SASS), built from the same source and compiler flags as the RTX 4090 evaluation binaries; Jetson builds were not audited. The audit tool treats every value loaded from memory as a secret candidate and tracks per instruction whether it reaches a branch condition or memory address. Warnings arose only from block-uniform retry branches, AES key-schedule S-box addresses indexed by public values, and packing kernels reading only public inputs; all 113 bugs deliberately injected into the linear-term code were detected. We assume, without verifying, that the latency of warp votes, population counts, warp shuffles, shared-memory atomics, BMMA, and similar instructions is independent of secret operands.

Zeroization of secret temporaries. On termination, the KeyGen and Sign kernels of the bitsliced implementation zeroize the shared-memory workspace and designated secret temporary buffers in global memory, as machine-code inspection of our implementation and an instrumented build reading the buffers afterward confirmed. The resident keys L and O, reused across batch calls, stay in global memory until freed; the guarantee excludes zeroization in the same-key tensor Sign kernel and values left in registers.

7 Evaluation

7.1 Experimental Setup

We evaluated all operations of the bitsliced implementation on distinct-key batches ($U = B$) and Sign and Verify of the tensor path on same-key batches ($U = 1$), including key preparation. The CPU baseline is the AVX2 path of the third-round reference implementation. cuMAYO [CHJ+26, Sys26] follows the second-round specification, which

Table 3: Throughput (10^3 ops/s) on the three GPUs at the largest batch measured for each implementation. Parenthesized values are throughput ratios to cuMAYO on the same GPU.

		RTX 4090			Jetson AGX Orin			Jetson AGX Xavier		CPU AVX2
Param.	Op.	cuMAYO	Bitsliced	Tensor	cuMAYO	Bitsliced	Tensor	cuMAYO	Bitsliced	(1 core)
MAYO_1	KeyGen	679.1	2,161.8 (3.18×)	—	24.3	81.2 (3.34×)	—	23.1	64.2 (2.79×)	49.5
	Sign	212.8	1,003.2 (4.71×)	2,927.0 (13.75×)	7.6	32.6 (4.32×)	126.1 (16.69×)	7.1	15.8 (2.22×)	17.7
	Verify	433.7	1,806.4 (4.16×)	7,555.0 (17.42×)	17.1	70.1 (4.09×)	296.7 (17.30×)	13.1	48.7 (3.72×)	34.4
MAYO_2	KeyGen	756.6	2,629.4 (3.48×)	—	26.5	100.5 (3.78×)	—	13.2	73.1 (5.53×)	60.4
	Sign	415.9	1,549.1 (3.72×)	6,746.0 (16.22×)	14.3	56.0 (3.91×)	264.0 (18.42×)	12.9	24.3 (1.88×)	36.8
	Verify	1,139.1	3,132.7 (2.75×)	17,917.0 (15.73×)	40.6	118.1 (2.91×)	658.9 (16.23×)	45.6	77.0 (1.69×)	102.4
MAYO_3	KeyGen	283.8	843.6 (2.97×)	—	12.1	31.4 (2.59×)	—	9.5	26.4 (2.80×)	19.5
	Sign	85.3	399.5 (4.69×)	1,544.5 (18.12×)	3.7	14.7 (4.01×)	62.1 (17.00×)	2.9	8.6 (2.92×)	8.6
	Verify	181.7	749.0 (4.12×)	3,723.4 (20.49×)	7.0	29.0 (4.12×)	153.7 (21.84×)	5.5	21.9 (3.97×)	16.0
MAYO_5	KeyGen	93.4	362.4 (3.88×)	—	3.7	14.4 (3.88×)	—	3.1	11.8 (3.84×)	7.1
	Sign	25.5	171.4 (6.73×)	780.4 (30.62×)	1.1	7.2 (6.30×)	28.9 (25.41×)	0.8	3.7 (4.33×)	3.7
	Verify	57.4	349.2 (6.09×)	1,351.4 (23.55×)	2.8	13.5 (4.82×)	61.2 (21.78×)	2.0	10.6 (5.36×)	6.6

differs from the third round in the MAYO_1 and MAYO_2 parameter sets and, for all four, in the linear term and the whipping schedule. We built it unmodified from its public code and measured it with the same harness and conditions.

We timed the GPU implementations over kernel execution with inputs on the device (the device-only region, which includes key expansion in the bitsliced implementation and cuMAYO), except Sign of the tensor path, timed over the whole application programming interface (API) call excluding transfers, and the CPU baseline in wall-clock time. Per batch size, throughput was measured ten times and latency 1,000 times (200 for $B > 1{,}024$). We report the median of per-run medians over three runs (one latency run on the Jetson devices) for the GPU implementations and the median of one run for the CPU baseline. Values in the introduction and conclusion without a named device are from the RTX 4090. The Jetson AGX Orin and Xavier, embedded GPUs that share memory with the CPU and have 14 and eight SMs (versus 128 on the RTX 4090), show whether the gain over cuMAYO persists with fewer resources. As the three GPUs (Table 2) differ in clock, compiler, and memory architecture, we compare implementations only within each GPU.

7.2 Distinct-Key Batches

Table 3 lists the throughput of the four implementations on the three GPUs. On the RTX 4090, the bitsliced implementation outperformed the CPU baseline by 30.58× (MAYO_2 Verify) to 56.78× (MAYO_1 Sign) and cuMAYO by 2.75× (MAYO_2 Verify) to 6.73× (MAYO_5 Sign) for every operation and parameter set. It also outperformed cuMAYO in every combination on the Jetson AGX Orin, by 2.59× (MAYO_3 KeyGen) to 6.30× (MAYO_5 Sign), and on the Jetson AGX Xavier, by 1.69× (MAYO_2 Verify) to 5.53× (MAYO_2 KeyGen). Since this throughput includes key expansion, performed per instance in the same kernel as the operation, the GPU can achieve it even for workloads in which each request uses a different key, such as remote signing services.

For single-shot latency ($B = 1$, Table 4), the bitsliced implementation was faster than

Table 4: Single-shot latency ($B = 1$, μs) on the three GPUs. Parenthesized values are latency ratios t_{cuMAYO}/t on the same GPU.

		RTX 4090			Jetson AGX Orin			Jetson AGX Xavier		CPU AVX2
Param.	Op.	cuMAYO	Bitsliced	Tensor	cuMAYO	Bitsliced	Tensor	cuMAYO	Bitsliced	(1 core)
MAYO_1	KeyGen	1,330.2	93.2 (14.27×)	—	3,958.7	256.5 (15.44×)	—	2,538.7	184.4 (13.77×)	20.3
	Sign	4,548.6	303.1 (15.01×)	354.6 (12.83×)	11,027.4	836.0 (13.19×)	991.4 (11.12×)	8,348.7	570.5 (14.63×)	53.6
	Verify	1,689.6	129.0 (13.10×)	254.0 (6.65×)	4,170.2	329.7 (12.65×)	690.3 (6.04×)	3,032.0	240.4 (12.61×)	29.1
MAYO_2	KeyGen	815.1	87.0 (9.36×)	—	2,323.9	219.1 (10.60×)	—	2,206.2	164.0 (13.45×)	16.4
	Sign	3,191.8	180.2 (17.71×)	234.0 (13.64×)	8,617.9	495.2 (17.40×)	655.3 (13.15×)	7,530.3	346.6 (21.73×)	27.2
	Verify	641.0	91.7 (6.99×)	143.4 (4.47×)	1,626.2	229.3 (7.09×)	419.8 (3.87×)	1,002.3	171.9 (5.83×)	9.6
MAYO_3	KeyGen	2,746.4	195.6 (14.04×)	—	7,619.0	539.3 (14.13×)	—	5,275.2	391.0 (13.49×)	51.5
	Sign	9,272.3	435.2 (21.31×)	502.6 (18.45×)	24,611.4	1,238.7 (19.87×)	1,475.1 (16.68×)	16,998.1	951.2 (17.87×)	117.8
	Verify	3,540.0	270.9 (13.07×)	464.9 (7.61×)	8,796.1	729.7 (12.05×)	1,319.9 (6.66×)	5,911.8	623.4 (9.48×)	61.8
MAYO_5	KeyGen	7,303.2	465.9 (15.67×)	—	20,152.5	1,332.9 (15.12×)	—	24,276.0	1,131.1 (21.46×)	141.7
	Sign	26,943.5	784.4 (34.35×)	916.3 (29.40×)	70,917.7	2,338.1 (30.33×)	3,232.8 (21.94×)	72,637.0	1,811.9 (40.09×)	275.4
	Verify	9,152.5	515.9 (17.74×)	999.4 (9.16×)	23,063.7	1,476.1 (15.62×)	3,882.0 (5.94×)	26,483.2	1,306.7 (20.27×)	150.4

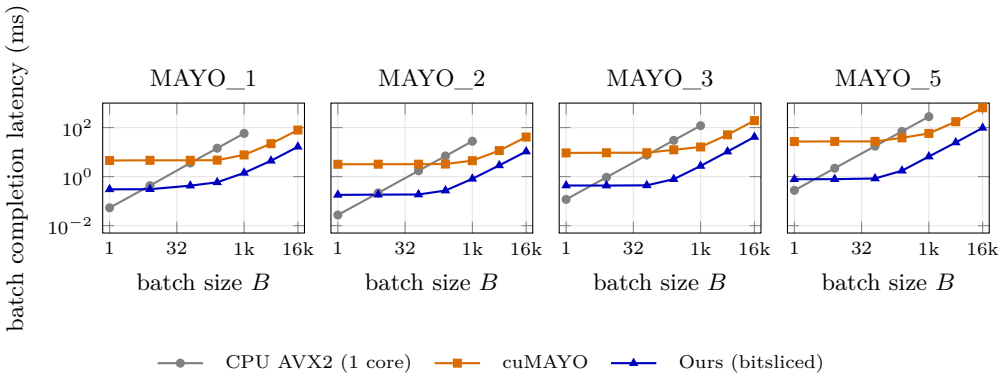


Figure 8: Batch completion latency of Sign versus B on the RTX 4090, from the start of a request until all B instances finish (log–log scale). The CPU baseline was measured only for $B \leq 1{,}024$.

cuMAYO for every operation and parameter set, by 6.99× to 34.35× on the RTX 4090, 7.09× to 30.33× on the Orin, and 5.83× to 40.09× on the Xavier, with the smallest ratio for MAYO_2 Verify and the largest for MAYO_5 Sign on each GPU.

Figure 8 plots the batch completion latency of Sign on the RTX 4090 against B. For all four parameter sets, the CPU baseline was faster than the bitsliced implementation at $B = 1$ but slower from $B = 8$ onward. From $B = 1$ to 256, the bitsliced implementation's latency grew by only 1.50× to 2.22×, staying below cuMAYO's at every measured B.

7.3 Same-Key Batches

On same-key batches, the tensor path with key preparation achieved 13.75× to 30.62× higher throughput (Table 3) than cuMAYO on distinct-key batches on the RTX 4090, and

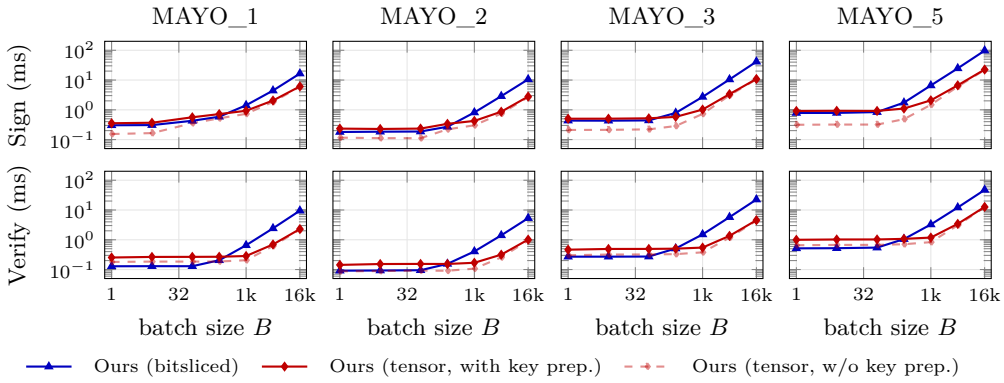


Figure 9: Batch completion latency of the same-key tensor path versus the batch size B on the RTX 4090 (log–log scale). The bitsliced implementation was measured on distinct-key batches.

16.23× (MAYO_2 Verify) to 25.41× (MAYO_5 Sign) on the Jetson AGX Orin. At the largest batch on the RTX 4090, per-key preparation lowered its throughput by at most 1.7% in Sign and Verify of the four parameter sets. For Verify on distinct-key batches, by contrast, the tensor path pays key preparation per instance, and at the largest batch it reached, the bitsliced implementation achieved 15.33× to 27.20× higher throughput than it. Both results support using the tensor path only for same-key batches and the bitsliced implementation for distinct-key batches.

On the RTX 4090, the tensor path's same-key latency with key preparation (Table 4) was 4.47× to 29.40× lower than cuMAYO's but higher than the bitsliced implementation's in every combination; on the Orin, it was 3.87× to 21.94× lower than cuMAYO's.

Figure 9 shows the batch completion latency of the same-key tensor path on the RTX 4090 with and without key preparation, alongside the bitsliced implementation. At $B = 1$, key preparation took 29.1% to 65.6% of the tensor path's latency. Among the measured B, the tensor path with key preparation first finished before the bitsliced implementation at $B = 1{,}024$ for MAYO_1 Sign and Verify, MAYO_2 Sign, and MAYO_5 Verify and at $B = 256$ for the other four combinations, staying ahead at every larger B.

Under the conditions of Figure 1 (MAYO_3, $B = 4{,}096$, RTX 4090 under WSL2), we re-measured the per-step share of execution time for both optimized paths, excluding key preparation on same-key batches. On distinct-key batches, the share of $\mathbb{F}_{16}$ arithmetic fell from 87.8% (Sign) and 80.2% (Verify) in the GPU port to 43.7% and 30.8% in the bitsliced implementation. In the same-key tensor path, it fell from 94.0% to 52.3% in Sign, raising Others, which contains solving the linear system with one block per instance, from 6.0% to 47.7%, but remained the largest step in Verify at 86.7%. Both paths thus cut their target step's time, the largest in the GPU port, proportionally more than the other steps. In same-key Sign, Others takes a share comparable to $\mathbb{F}_{16}$ arithmetic's, so once on Tensor Cores, the matrix products alone no longer determine Sign's execution time.

7.4 Comparison with Prior Work

Table 5 compares MAYO_1 with prior signature implementations at NIST security levels 1 and 2, alongside our implementations and cuMAYO on the three GPUs. For other works, we list only the metrics each paper reported; the MAYO row for the field-programmable gate array (FPGA) implements the first-round specification. As devices, specification versions, and measured regions differ, we give no ratios between their values and ours. For single-shot requests, MAYO_1 on one AVX2 core had lower latency than our implementations on the

Table 5: Comparison of MAYO_1 with prior signature implementations; for other works, only the metrics each paper reported are listed (—: not reported).

		Throughput (kops/s)			Latency (μs)
Implementation	Device	KeyGen	Sign	Verify	KeyGen/Sign/Verify
AVX2 ref. [BCC+26] (MAYO_1)	i7-13700K	49.5	17.7	34.4	20.3/53.6/29.1
FPGA [HSK+24] (MAYO1, round 1)	Artix-7	—	—	—	80.0/430.0/50.0
ML-Cube [ZZX+25] (ML-DSA-44)	RTX 4090	—	1,153.9	—	—
cuFalcon [LWS+26] (Falcon-512)		—	172.1	3,879.2	—
GOLF [DDQ+25] (Falcon-512)		—	420.3	10,311.0	—
HERO-Sign [ZW26] (SPHINCS+-128f)		—	119.5	—	—
GRASP [NDL+24] (SPHINCS+-128f)		—	146.4	1,499.6	—
cuMAYO [CHJ+26] (MAYO_1)	RTX 4090	679.1	212.8	433.7	1,330.2/4,548.6/1,689.6
Ours, bitsliced (MAYO_1)		2,161.8	1,003.2	1,806.4	93.2/303.1/129.0
Ours, tensor (MAYO_1)		—	2,927.0	7,555.0	—/354.6/254.0
cuMAYO [CHJ+26] (MAYO_1)	Jetson AGX Orin	24.3	7.6	17.1	3,958.7/11,027.4/4,170.2
Ours, bitsliced (MAYO_1)		81.2	32.6	70.1	256.5/836.0/329.7
Ours, tensor (MAYO_1)		—	126.1	296.7	—/991.4/690.3
cuMAYO [CHJ+26] (MAYO_1)	Jetson AGX Xavier	23.1	7.1	13.1	2,538.7/8,348.7/3,032.0
Ours, bitsliced (MAYO_1)		64.2	15.8	48.7	184.4/570.5/240.4

RTX 4090 in all three operations, so the GPU advantage lies in throughput.

8 Conclusion

Servers receiving many signing and verification requests, such as TLS servers, DNS servers signing DNSSEC responses, and Certificate Transparency logs, can raise throughput by batching independent requests on a GPU. MAYO's nibble-sliced representation, however, suits CPU shuffle instructions or multiplication tables, so on GPUs every $\mathbb{F}_{16}$ multiplication costs a memory access or several instructions. We presented a GPU implementation of the third-round MAYO specification that is bitsliced throughout on the GPU yet keeps the input and output byte strings exactly as specified.

A bitsliced MAC over 32 coefficients needs only word-wide AND and XOR, without table lookups, cutting the 60 primitive operations of the nibble-sliced reference code path to 35, excluding per-scalar setup. KeyGen, Sign, and Verify compute their matrix products with this circuit, each in a single kernel from key expansion to the output. AES outputs land directly in bit planes, where data stay up to solving the linear system. Bit planes also suit Tensor Cores: splitting an $\mathbb{F}_{16}$ matrix product into 16 binary products between planes and concatenating same-key instances along the batch axis yields large binary matrix products of the whole batch with the key's single matrix. The tensor path runs these Sign and Verify products on Tensor Cores, preparing the key-side matrix only once per key.

On an RTX 4090 with distinct-key batches, the bitsliced implementation achieved 2.75× to 6.73× higher throughput and 6.99× to 34.35× lower single-shot latency ($B = 1$) than cuMAYO for all three operations of the four parameter sets. As this throughput includes per-instance key expansion, it holds even for workloads in which each request uses a different key. Even including key preparation, the tensor path in same-key batches achieved 13.75× to 30.62× higher throughput and 4.47× to 29.40× lower single-shot latency than cuMAYO on distinct-key batches. Per-step times remeasured after optimization show a substantially smaller share of $\mathbb{F}_{16}$ arithmetic, the largest step in the GPU port, confirming that both paths shortened their targeted steps proportionally more than other steps. The bitsliced implementation serves workloads that expand the key per request, and the tensor path bulk workloads that gather same-key requests to share the key-side matrix and its preparation. With 256 to 1,024 or more same-key requests, the tensor path finished before the bitsliced implementation on distinct-key batches.

Use of AI Tools

The authors used Claude for parts of grammar checking, figure drawing, and code writing. All outputs were reviewed and verified by the authors.

References

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status report on the third round of the NIST post-quantum cryptography standardization process. NIST Internal Report 8413-upd1, National Institute of Standards and Technology, 2022.

[AAL+05] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. DNS security introduction and requirements. RFC 4033, IETF, March 2005.

[ABC+26] Gorjan Alagic, Maxime Bros, Pierre Ciadoux, Quynh Dang, Thinh Hung Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Hamilton Silberg, Daniel Smith-Tone, and Noah Waller. Status report on the second round of the additional digital signature schemes for the NIST post-quantum cryptography standardization process. NIST Internal Report 8610, National Institute of Standards and Technology, May 2026.

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):402–425, 2021. ePrint 2020/1123.

[BCC+24] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. Nibbling MAYO: Optimized implementations for AVX2 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):252–275, 2024. ePrint 2023/1683.

[BCC+25] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO: Algorithm specifications and supporting documentation. NIST Post-Quantum Cryptography Standardization, Additional Digital Signature Schemes, Round 2 Submission, 2025.

[BCC+26] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO: Algorithm specifications and supporting documentation. NIST Post-Quantum Cryptography Standardization, Additional Digital Signature Schemes, Round 3 Submission, 2026. Round 3 version, August 31, 2026. https://pqmayo.org/assets/specs/mayo-round3.pdf.

[Beu22] Ward Beullens. MAYO: Practical post-quantum signatures from oil-and-vinegar maps. In *Selected Areas in Cryptography (SAC 2021)*, volume 13203 of *LNCS*, pages 355–376. Springer, 2022.

[CHJ+26] Haoyang Chen, Junhao Huang, Shutong Jin, Ray C. C. Cheung, Donglong Chen, and Wangchen Dai. High-throughput GPU design and implementation of MAYO with matrix computation reordering. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(4):496–521, 2026.

[DDQ+25] Ruihao Dai, Jiankuo Dong, Mingrui Qiu, Zhenjiang Dong, Fu Xiao, and Jingqiang Lin. GOLF: Unleashing GPU-driven acceleration for FALCON post-quantum cryptography. Cryptology ePrint Archive, Paper 2025/749, 2025.

[DSvW21] Léo Ducas, Marc Stevens, and Wessel van Woerden. Advanced lattice sieving on GPUs, with tensor cores. In *Advances in Cryptology – EUROCRYPT 2021*, LNCS, pages 249–279. Springer, 2021. ePrint 2021/141.

[FWX+23] Shengyu Fan, Zhiwei Wang, Weizhi Xu, Rui Hou, Dan Meng, and Mingzhe Zhang. TensorFHE: Achieving practical computation on encrypted data using GPGPU. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA 2023)*, pages 922–934, 2023.

[FZZ+25] Guang Fan, Mingzhe Zhang, Fangyu Zheng, Shengyu Fan, Tian Zhou, Xianglong Deng, Wenxu Tang, Liang Kong, Yixuan Song, and Shoumeng Yan. WarpDrive: GPU-based fully homomorphic encryption acceleration leveraging tensor and CUDA cores. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA 2025)*, pages 1187–1200, 2025.

[Gra16] Dani Grant. Economical with the truth: Making DNSSEC answers cheap. Cloudflare blog, June 2016.

[Guð15] Ólafur Guðmundsson. DNSSEC done right. Cloudflare blog, January 2015.

[HSK+24] Florian Hirner, Michael Streibl, Florian Krieger, Ahmet Can Mert, and Sujoy Sinha Roy. Whipping the multivariate-based MAYO signature scheme using hardware platforms. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3421–3435. ACM, 2024. ePrint 2023/1267.

[JDW+25] Dian Jiao, Xianglong Deng, Zhiwei Wang, Shengyu Fan, Yi Chen, Dan Meng, Rui Hou, and Mingzhe Zhang. Neo: Towards efficient fully homomorphic encryption acceleration using tensor core. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA 2025)*, pages 107–121. ACM, 2025.

[LMS21] Ben Laurie, Eran Messeri, and Rob Stradling. Certificate transparency version 2.0. RFC 9162, IETF, December 2021.

[LWS+26] Wenqian Li, Hanyu Wei, Shiyu Shen, Hao Yang, Wangchen Dai, and Yunlei Zhao. cuFalcon: An adaptive parallel GPU implementation for high-performance Falcon acceleration. *IEEE Transactions on Parallel and Distributed Systems*, 37(5):1153–1167, 2026. ePrint 2025/249.

[LZS+24] Wai-Kong Lee, Raymond K. Zhao, Ron Steinfeld, Amin Sakzad, and Seong Oun Hwang. High throughput lattice-based signatures on GPUs: Comparing Falcon and Mitaka. *IEEE Transactions on Parallel and Distributed Systems*, 35(4):675–692, 2024. ePrint 2023/399.

[MTBM21] Brendan Moran, Hannes Tschofenig, David Brown, and Milosch Meriac. A firmware update architecture for internet of things. RFC 9019, IETF, April 2021.

[Nat22] National Institute of Standards and Technology. Call for additional digital signature schemes for the post-quantum cryptography standardization process. NIST Post-Quantum Cryptography Standardization, 2022. Published September 2022, updated October 2022.

[Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Federal Information Processing Standards Publication 204, National Institute of Standards and Technology, August 2024.

[Nat24b] National Institute of Standards and Technology. Stateless hash-based digital signature standard. Federal Information Processing Standards Publication 205, National Institute of Standards and Technology, August 2024.

[Nat26] National Institute of Standards and Technology. Round 3 additional signatures. NIST Post-Quantum Cryptography: Digital Signature Schemes, 2026. Accessed 2026-10-12.

[NDL+24] Yijing Ning, Jiankuo Dong, Jingqiang Lin, Fangyu Zheng, Yu Fu, Zhenjiang Dong, and Fu Xiao. GRASP: Accelerating hash-based PQC performance on GPU parallel architecture. Cryptology ePrint Archive, Paper 2024/1030, 2024.

[NMTA22] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. Sigstore: Software signing for everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, pages 2353–2367. ACM, 2022.

[NVI17] NVIDIA Corporation. NVIDIA Tesla V100 GPU architecture. Technical Report WP-08608-001_v1.1, NVIDIA, 2017. Whitepaper.

[NVI20] NVIDIA Corporation. NVIDIA A100 tensor core GPU architecture. Technical report, NVIDIA, 2020. Whitepaper, Table 4.

[NVI26a] NVIDIA Corporation. *CUDA Programming Guide*, 2026. Release 13.4.2, September 10, 2026.

[NVI26b] NVIDIA Corporation. *Parallel Thread Execution ISA*, 2026. Version 9.4, last updated October 12, 2026. Section "Multiply-and-Accumulate Instruction: mma", Target ISA notes.

[Res26] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. RFC 9846, IETF, July 2026.

[SBR24] Samyuktha M, Pallavi Borkar, and Chester Rebeiro. Bitsliced Jasmin implementation of the Mayo signature scheme. Cryptology ePrint Archive, Paper 2024/1893, 2024.

[SFY+26] Yintai Sun, Shengyu Fan, Zhenhua Yin, Xinkai Song, Xing Hu, Zidong Du, Qi Guo, Weizhi Xu, Rui Hou, Dan Meng, Song Bian, and Mingzhe Zhang. TensorFHE+: Fully homomorphic encryption acceleration based on linear algebra. *IEEE Transactions on Computers*, 75(2):612–627, 2026.

[SMA+24] Oussama Sayari, Soundes Marzougui, Thomas Aulbach, Juliane Krämer, and Jean-Pierre Seifert. HaMAYO: A fault-tolerant reconfigurable hardware implementation of the MAYO signature scheme. In *Constructive Side-Channel Analysis and Secure Design (COSADE 2024)*, LNCS, pages 240–259. Springer, 2024. ePrint 2023/1135.

[Swi26] Swisscom IT Services Finance S.E. Terms and conditions of use Swisscom ITSF trust service (qualified and advanced electronic signatures). Terms of use, July 2026.

[SYD+24] Shiyu Shen, Hao Yang, Wangchen Dai, Hong Zhang, Zhe Liu, and Yunlei Zhao. High-throughput GPU implementation of Dilithium post-quantum digital signature. *IEEE Transactions on Parallel and Distributed Systems*, 35(11):1964–1976, 2024. arXiv:2211.12265.

[SYLZ25] Shiyu Shen, Hao Yang, Wenqian Li, and Yunlei Zhao. cuML-DSA: Optimized signing procedure and server-oriented GPU design for ML-DSA. *IEEE Transactions on Dependable and Secure Computing*, 22(3):2295–2307, 2025. ePrint 2023/1522.

[Sys26] SysuCrypto. cuMAYO: GPU implementation of MAYO. GitHub repository, 2026. Commit `a2b68a0`, 2026-07-09.

[WDC+25] Ziheng Wang, Xiaoshe Dong, Heng Chen, Yan Kang, and Qiang Wang. CUSPX: Efficient GPU implementations of post-quantum signature SPHINCS+. *IEEE Transactions on Computers*, 74(1):15–28, 2025.

[WZF+22] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. A novel high-performance implementation of CRYSTALS-Kyber with AI accelerator. In *European Symposium on Research in Computer Security (ESORICS 2022), Part III*, volume 13556 of *LNCS*, pages 514–534. Springer, 2022. ePrint 2022/881.

[ZW26] Yaoyun Zhou and Qian Wang. HERO-Sign: Hierarchical tuning and efficient compiler-time GPU optimizations for SPHINCS+ signature generation. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–13, 2026.

[ZZF+24] Tian Zhou, Fangyu Zheng, Guang Fan, Lipeng Wan, Wenxu Tang, Yixuan Song, Yi Bian, and Jingqiang Lin. ConvKyber: Unleashing the power of AI accelerators for faster Kyber with novel iteration-based approaches. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):25–63, 2024. ePrint 2024/095.

[ZZX+25] Tian Zhou, Fangyu Zheng, Zhuoyu Xie, Wenxu Tang, Guang Fan, Yijing Ning, Yi Bian, Jingqiang Lin, and Jiwu Jing. ML-Cube: Accelerating module-lattice-based cryptography using machine learning accelerators with a memory-less design. In *Proceedings of the 32nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2025)*, pages 1829–1843. ACM, 2025.