IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–25. DOI:XXXXXXXX

Optimizing HQC with Enhanced Parallel Binary Polynomial Multiplication on GPUs

DongCheon Kim[1], JunHyeok Choi[1], Wai Kong Lee[2] and Seog Chung Seo[1*]

[1] Kookmin University, Seoul, Republic of Korea,
{kindongsy,c981126,scseo}@kookmin.ac.kr
[2] Universiti Tunku Abdul Rahman, Kampar, Malaysia, wklee@utar.edu.my

Abstract. Graphics processing units (GPUs) are widely employed as accelerators in modern servers to handle key-establishment requests in bulk. Such deployments require the GPU implementation to sustain high throughput while keeping each request within its latency bound. In this work, we present a high-performance, end-to-end GPU implementation of Hamming Quasi-Cyclic (HQC), a code-based key encapsulation mechanism (KEM). We propose two novel techniques to accelerate the dominant cost of HQC, which is binary polynomial multiplication over Galois Field GF(2). The first is a parallelization of the Frobenius additive fast Fourier transform (FAFFT), and the second is a mapping of the cyclic product, viewed as a Toeplitz matrix–vector product, onto one-bit integer (INT1) tensor cores. We further explore the design space of these multipliers, measuring them directly on the GPU to derive the best combination for the architecture. We further propose a GPU-resident computation strategy that keeps the entire KEM on the GPU, thereby reducing costly memory traffic. Beyond multiplication, we introduce warp-level parallelization strategies to accelerate the remaining stages, hashing, Reed–Muller/Reed–Solomon (RM/RS) coding, and fixed-weight sampling, which become the performance bottleneck once multiplication is optimized. By varying the mapping of KEM instances to GPU threads, we derive five variants spanning the trade-off between throughput and latency and organize them into a thread-to-KEM mapping taxonomy. On an RTX 4090, the Coarse-Grained (CG) variant maximizes batch throughput. Our multiplier reaches 3,828k polymul/s for standalone multiplication, about 2.9× the latest GPU HQC implementation HIGH, and for the full KEM the CG variant achieves 1.56×, 1.55×, and 2.03× higher end-to-end throughput than HIGH for keygen, encaps, and decaps. For single-shot latency the tensor-core-based BMMAw variant achieves a latency as low as 254μs. The proposed warp-parallel FGmw variant still attains a latency of 354μs, demonstrating that our approach remains effective on GPU architectures lacking INT1 tensor-core support. All five variants are further evaluated on an RTX 3090 and an embedded Jetson AGX Orin to confirm cross-platform portability.

Keywords: PQC · HQC · GPU · FAFFT · Tensor Cores

1 Introduction

HQC is a code-based KEM that NIST selected in March 2025 for standardization as an additional post-quantum KEM complementing ML-KEM. Since HQC is based on coding theory instead of the popular lattice hardness, it is viewed as a diversification effort toward crypto-agility that can function as a backup solution even if lattice-based schemes face vulnerabilities. Deploying post-quantum cryptography (PQC) on GPUs is viewed as a practical requirement, evident from several prior GPU KEM studies [ZZF+24,

*Corresponding author.

Licensed under Creative Commons License CC-BY 4.0.

DFQ+24, WDX+25]. In environments such as large-scale Transport Layer Security (TLS) handshake, where a server processes key agreements in bulk, throughput becomes the axis of performance, and at the same time each request must respect the latency bound set by the protocol or a service-level agreement (SLA). We expose a range of operating points spanning the throughput–latency spectrum. Each deployment scenario can then select the point that matches its needs, obtaining low latency at small batches or high throughput at large batches. On GPUs, which offer thousands of threads but carry a high per-instance cost, this throughput–latency trade-off becomes pronounced, and we organize it through the thread-to-KEM mapping taxonomy of Section 3.

Existing GPU HQC research mainly focused on improving the multiplication throughput. The closest state-of-the-art work, HIGH [DHW+26], implements the entire KEM on the GPU with unpadded-Karatsuba multiplication and kernel fusion. Similar to our work, they also employed dense×dense multiplication to avoid sparse-secret leakage, but its implementation is only limited to HQC-1, leaving three performance gaps unresolved. Firstly, the FAFFT, introduced by Li et al. [LCK+18], has been optimized for HQC polynomial multiplication on the Advanced Vector Extensions 2 (AVX2), Galois Field New Instructions (GFNI), and NEON CPUs [CCPY26] and ARM Cortex-M4 [CCC+26, JLK+25, LJKH25], and it has further been realized on field-programmable gate array (FPGA) hardware [RLC+25a, RLC+25b]. However, no GPU implementation of this FAFFT has been reported, and even the latest GPU HQC implementation, HIGH [DHW+26], used Karatsuba for the multiplication. This is because the FAFFT is an XOR-based operation over the binary field GF(2), whereas the number theoretic transform (NTT) that GPU PQC has optimized is integer-modular arithmetic, so the GPU kernels and optimization techniques matured for lattice NTT cannot be reused as-is and must be built from scratch. Besides bit-sliced binary-field multiplication [BSHST16], additive-FFT-related GPU work exists in other contexts, such as Classic McEliece root finding [WDX+25, XGZW25] and Fast Reed–Solomon Interactive Oracle Proof of Proximity (FRI)-based zkSNARK polynomial encoding [JG25], but cannot be used directly for HQC, which must further add the pointwise multiplication and the inverse transform to complete a multiplication. Secondly, existing tensor-core polynomial works target lattice and integer-modulus arithmetic [ZZF+24, LSZH22, ZZX+25], integer Toeplitz matrix multiplication [HLKH23], and zero-knowledge proof (ZKP) arithmetic [LZG+26]. These map naturally onto the tensor core's multiply–accumulate, but the carry-less XOR of cyclic GF(2) polynomial multiplication for a code-based KEM has never been considered as such a target. Thirdly, GPU code-based KEMs like BIKE [DFQ+24], McEliece [WDX+25], and HIGH [DHW+26] go no further than integrating SHAKE256 into their fused kernels or thread-parallelizing the fixed-weight vector write, and none of them parallelizes HQC's hashing, coding, and sampling at the warp level, component by component.

This work fills these three gaps. We present the first GPU FAFFT binary polynomial multiplication and the first tensor-core binary polynomial multiplication in this work. We also parallelize the HQC algorithm end to end on the GPU, including hashing, RM/RS coding, and sampling. In this work, the entire KEM is executed inside GPU kernels, so every stage and all intermediate data stays in device memory with no intermediate round-trips between GPU and CPU. We call this execution style GPU-resident execution. These proposed techniques are organized under a thread-to-KEM mapping taxonomy and the throughput–latency trade-off is characterized by measurement. We are also aware that timing and access-pattern leakage can break PQC software. A secret-dependent branch leaks through the run time [ABB+16], a lookup table indexed by a secret leaks it because its access time varies with the cache memory [GBHLY16], and a multiplication that exploits the sparsity of the secret leaks it directly [GAMA+25]. We counter these, respectively, with branch-free arithmetic masks, a constant-time masked-gather, and dense×dense multiplication.

1.1 Contributions

1. **The first parallelization of FAFFT binary polynomial multiplication on a GPU.** FAFFT-based binary multiplication has so far been implemented only on CPU [CCPY26], ARM [CCC+26, JLK+25, LJKH25], and FPGA hardware [RLC+25a, RLC+25b], and GPU additive FFT has been used only for evaluation and interpolation in other applications [WDX+25, XGZW25, JG25], never to optimize binary polynomial multiplication. We parallelize the FAFFT multiplication pipeline on the GPU with thread/warp/block level mappings, together with radix-4 butterfly fusion. The resulting multiplier reaches a peak of 3,828k polymul/s for HQC-1 standalone polynomial multiplication, about 2.9× the 1,318k ops/s of HIGH [DHW+26].

2. **The first INT1 tensor-core mapping of cyclic** GF(2) **polynomial multiplication for code-based PQC, BMMA.** Existing GPU implementations that exploit the tensor-core for polynomial multiplications were all targeting the lattice or integer arithmetic [ZZF+24, LSZH22, HLKH23]. In this work, we provide a new perspective, viewing the cyclic product as a Toeplitz matrix product and mapping it to the INT1 tensor-core instructions. This allows us to map AND operator as multiply, *popcount* as accumulation, and the accumulated value as a GF(2) coefficient, and finally obtain a single-shot multiplication latency ∼15 μs and a peak of 1,940k polymul/s. Together with the warp-parallel implementation of the non-multiplication part, this multiplier is the basis of our BMMAw variant, which achieves the lowest small-batch KEM latency.

3. **Parallelizing non-multiplier parts of the HQC scheme.** Our warp- and block-cooperative multiplication lowers the HQC multiplication share from ∼85% to about 22%, exposing the non-multiplication stages as the bottleneck. Going beyond multiplication, we parallelize the hash functions and RM/RS encoding/decoding at the warp level and optimize fixed-weight sampling component by component, making the entire flow GPU-resident. The warp-Keccak accelerates the permutation by 2.3×, and combined with warp RM/RS coding it accelerates Fine-Grained (FG) decapsulation by +49–75% over the baseline, and warp-parallel fixed-weight sampling cuts the single-shot encaps latency from 17,545 to 2,191 μs.

4. **Design space exploration of the optimal combination of multipliers.** Several ideas of splitting the multiplication into an FAFFT body and a tail, or into smaller multiplications, were proposed for AVX2/GFNI [CCPY26] and for ARM Cortex-M4 [CCC+26, JLK+25]. To find the best combination on a target GPU architecture, we port these methods to the GPU and evaluate them across Karatsuba, Chinese remainder theorem (CRT), schoolbook, and other splits and combinations. The findings differ markedly from prior CPU/ARM work, and the best throughput is achieved by the CG multiplier combining bitslice, schoolbook split, tower FAFFT, and a Karatsuba tail.

5. **Extensive evaluation across GPU architectures.** All experiments run on an RTX 4090 across the three security levels, covering standalone polynomial multiplication, KEM throughput at two measurement boundaries, single-shot and batch latency, and a comparison with the prior work HIGH [DHW+26]. The throughput champion CG achieves 1.56×, 1.55×, and 2.03× higher end-to-end throughput than state-of-the-art, HIGH [DHW+26] for keygen, encaps, and decaps. To show the design is not limited to one device, we further execute all the proposed variants on an RTX 3090 and a Jetson AGX Orin, confirming the taxonomy holds across GPU generations and embedded platforms.

Table 1: Notation and abbreviations.

Abbr.	Full form	Meaning
FAFFT	Frobenius additive fast Fourier transform	FFT methods for binary polynomials
SMEM	Shared memory	on-chip memory of a block
GMEM	Global memory	off-chip device memory
CG	Coarse-Grained	one thread per KEM operation
FG	Fine-Grained	one warp per KEM operation
FGmw	Fine-Grained multi-warp	one block per KEM operation
BMMA	Binary Matrix-Multiply–Accumulate	batched tensor-core multiplication
BMMAw	BMMA with warp-parallel stages	BMMA, other stages on one warp

Algorithm 1: HQC Public Key Encryption (PKE) with the KEM wrapper. The `mul` comments mark where the polynomial multiplication is used.

PKE.Keygen($seed_{\text{pke}}$):
- $(seed_{\text{dk}},\ seed_{\text{ek}}) \leftarrow \text{I}(seed_{\text{pke}})$
- $(y, x) \leftarrow \text{SampleFixedWeightVect}_s(seed_{\text{dk}})$ // rejection sampler
- $h \leftarrow \text{SampleVect}(seed_{\text{ek}})$
- $s \leftarrow x \oplus h \cdot y$ // `mul` 1
- **return** $ek = (seed_{\text{ek}}, s),\ dk = seed_{\text{dk}}$

PKE.Encrypt(ek, m, θ):
- $h \leftarrow \text{SampleVect}(seed_{\text{ek}})$
- $(r_2, e, r_1) \leftarrow \text{SampleFixedWeightVect}(\theta)$
- $u \leftarrow r_1 \oplus h \cdot r_2$ // `mul` 2
- $v \leftarrow \text{Encode}(m) \oplus \text{Truncate}(s \cdot r_2 \oplus e)$ // Encode = RS→RM; `mul` 3
- **return** $c = (u, v)$

PKE.Decrypt(dk, c):
- $y \leftarrow \text{SampleFixedWeightVect}_s(seed_{\text{dk}})$ // rejection sampler
- **return** $\text{Decode}\big(v \oplus \text{Truncate}(u \cdot y)\big)$ // Decode = RM→RS; `mul` 4

KEM = FO(PKE), implicit rejection:
- Encaps: $m \xleftarrow{\$} \mathcal{M};\ (K \,\|\, \theta) \leftarrow \text{G}(m);\ ct \leftarrow \text{Encrypt}(ek, m, \theta);$ output (ct, K)
- Decaps: $m' \leftarrow \text{Decrypt}(dk, ct)$
- $(K' \,\|\, \theta') \leftarrow \text{G}(m')$
- $ct' \leftarrow \text{Encrypt}(ek, m', \theta')$ // re-encryption; `mul` 5, 6
- output $[\, ct \stackrel{?}{=} ct' \,]\ ?\ K' : \bar{K}$ // FO comparison

Two of these results reach beyond HQC. Tensor cores built for integer arithmetic can also compute binary-field arithmetic, and the additive FFT and the RM/RS decoding that we parallelize at warp level come from coding theory, so the results extend to error-correction systems in communications and storage. A naive port that runs the reference implementation serially, one thread per instance, saturates at only 3.3k keygen/s at HQC-1 (Table 12), which makes these optimizations essential. Table 1 lists the abbreviations.

Our source code is available at https://github.com/kindongsy/TCHES-HQC_GPU.

2 Preliminaries

2.1 The HQC KEM

HQC is a code-based KEM whose security relies on the hardness of decoding quasi-cyclic codes. It achieves IND-CCA2 through a fixed-weight error model and the Fujisaki–

Table 2: HQC parameters per security level.

	HQC-1 (128)	HQC-3 (192)	HQC-5 (256)
Ring degree n	17,669	35,851	57,637
Body FFT transform size (bits)	32,768	65,536	131,072
Fixed weight ω	66	100	131
Error weight $\omega_e = \omega_r$	75	114	149
RS correction capability δ	15	16	29
RM$(1, 7)$ length	384	640	640

Table 3: Per-phase profiling of HQC-3.

		% of runtime			
Implementation	Op.	Mult.	Code	Hash & Sample	Other
Reference C	keygen	98.4	–	1.5	0.1
	encaps	98.4	0.2	1.2	0.2
	decaps	97.3	1.1	1.1	0.4
AVX2/PCLMULQDQ	keygen	37.9	–	60.2	1.8
	encaps	40.0	1.2	45.7	13.1
	decaps	29.2	24.4	34.5	12.0
Naive GPU	keygen	87.5	–	11.0	1.5
	encaps	88.0	1.0	9.5	1.5
	decaps	84.6	4.4	7.8	3.2
GPU, after FG mult.	decaps	21.9	12.1	56.5	9.5

Okamoto (FO) transform. One KEM operation comprises four primitive operations. They are polynomial multiplication in the ring $R = \mathbb{F}_2[X]/(X^n - 1)$, fixed-weight sampling that draws vectors with exactly ω ones, SHA-3/SHAKE-based hashing and the extendable-output function (XOF), and the encoding and decoding of the RM+RS concatenated code. The public key is a uniformly random h together with $s = x \oplus y \cdot h$. The secret key is the fixed-weight pair (x, y). Polynomial multiplication is the heaviest operation, invoked once in keygen, twice in encaps, and three times in decaps. Algorithm 1 summarizes this data flow.

The RM+RS concatenated code. Encoding flows from the outside in, with the outer RS code correcting byte-level symbols and the inner RM(1,7) code correcting the bits within them. This concatenation achieves the extremely low decoding-failure probability that security requires, efficiently using a combination of two short codes instead of a single large one.

Decoding runs in reverse. RM decoding recovers each block, then RS decoding locates the errors as the roots of the error-locator $\sigma(X)$, which the reference computes with the Gao–Mateer additive FFT.

Fixed-weight sampling. Fixed-weight sampling draws a vector with exactly ω ones, and the HQC specification [GAMA+25] prescribes two samplers, differing in the length of XOF randomness they consume. The rejection sampler, whose consumption is data-dependent, draws keygen's (x, y) and decaps's y, whereas the fixed-amount sampler consumes a fixed amount of randomness per weight and draws the (r_1, r_2, e) of encaps and decaps re-encryption. Table 2 summarizes the parameters of the three security levels.

Algorithm 2: FAFFT multiplication of $c = a \cdot b$ over $\mathbb{F}_2$.

Input: $a, b \in \mathbb{F}_2[X]$ (power-of-two-size polynomials)
Output: $c = a \cdot b$

foreach $f \in \{a, b\}$ **do** // evaluate both inputs
  $g \leftarrow \text{BasisConversion}(f)$
  $g \leftarrow \text{Encode}(g)$ // bitslice, reflects Frobenius
  $\hat{f} \leftarrow \text{Butterfly}(g)$ // subspace divide-and-conquer
end
foreach *evaluation point* p **do**
  $\hat{c}[p] \leftarrow \hat{a}[p] \cdot \hat{b}[p]$ // pointwise multiply
end
$\hat{c} \leftarrow \text{InverseButterfly}(\hat{c})$
$\hat{c} \leftarrow \text{Decode}(\hat{c})$ // inverse Encode
return $\text{InverseBasisConversion}(\hat{c})$ // $\Rightarrow$ product $a \cdot b$

2.2 HQC Bottleneck Analysis across Platforms

We profile the representative level HQC-3 and let measurement decide the optimization order. Firstly, the baseline version shows that polynomial multiplication dominates. It accounts for about 97–98% in the reference C implementation and still about 85–88% in the naive GPU port (Table 3), which makes it a target to optimize first (Section 4). Secondly, once the cost of multiplication is lowered, the bottleneck moves. When AVX2 accelerates multiplication with PCLMULQDQ, hash+sample overtakes multiplication at 46–60% of keygen and encaps, and coding rivals multiplication at 24% in decaps. The same happens on the GPU. After the FG warp-cooperative multiplier of Section 4.3, the HQC-3 decapsulation (see the last row of Table 3) shifts the shares to 21.9% multiplication, 12.1% coding, 56.5% hash+sample, and 9.5% other. This bottleneck-migration check motivates the warp-level non-multiplication optimizations presented in Section 5.

2.3 Polynomial Multiplication and the FAFFT

The core arithmetic on which every HQC KEM operation depends is multiplying two binary polynomials in the ring $R = \mathbb{F}_2[X]/(X^n - 1)$ to obtain $c(X) = a(X) \cdot b(X) \bmod (X^n - 1)$. Since the degree n ranges from 17,669 to 57,637 (Table 2), the choice of multiplication method dominates performance. The simplest schoolbook method is $O(n^2)$ and thus expensive at this scale. Karatsuba and Toom–Cook lower this to subquadratic via divide and conquer but still fall short of transform-based multiplication. What removes this obstruction is the additive FFT, which takes $\mathbb{F}_2$-linear subspaces closed under addition as evaluation points instead of roots of unity.

The additive FFT was refined through a series of three works. Gao–Mateer [GM10] first established the additive FFT via evaluation over power-of-two-sized subspaces. Lin–Chung–Han [LCH14] presented a polynomial basis that lowers the evaluation to $O(n \log n)$. The FAFFT by Li et al. [LCK+18] specialized it to binary inputs, reducing the number of evaluation points further. The FAFFT is a data-parallel computation of butterfly sweeps across several levels whose combine step is a simple XOR, which fits the GPU's wide single-instruction multiple-thread (SIMT) execution and high memory bandwidth well. In this paper, we showcase the first GPU parallelization of the FFT body under thread, warp, and block mappings, covering all three security levels (Section 4.2).

Algorithm 2 lists the stages of the FAFFT multiplication. The only places where multiplications actually occur are the per-level twiddles of the butterfly and the pointwise multiplication. Basis conversion, encode, and decode are linear operations consisting only

of XORs and repositioning. The HQC degree n is not a power of two, so the transform size must grow to hold the product. Because each FAFFT call carries a fixed per-call overhead, it is beneficial to split the multiplication while minimizing the number of costly FAFFT calls. Chen proposes a schoolbook split on AVX2/GFNI [CCPY26] and a CRT split on the Cortex-M4 [CCC+26], while Jang proposes a hybrid of FAFFT-CRT and FAFFT-Karatsuba splits [JLK+25]. Which split wins depends on the architecture, so we measure directly on the GPU and adopt the schoolbook split, selected in Section 4.1.

Even after this split, the body FFT and the Karatsuba tail each take a large share of multiplication time. Referring to Table 4, the bottleneck differs by level. The tail dominates HQC-1/3 and the butterfly the tail-less HQC-5, so both are acceleration targets.

2.4 GPU Architecture and the CUDA Execution Model

GPU execution is based on SIMT [NVI25a]. 32 threads are grouped into a warp and execute the same instruction in lockstep, and each thread in a warp is called a lane. GPUs since the Volta generation also provide *tensor cores*. These are warp-level matrix-multiply–accumulate (MMA) units that, in a single instruction, multiply two tiles and accumulate the product ($D = A \cdot B + C$), at much higher matrix-multiply throughput than the Compute Unified Device Architecture (CUDA) cores' scalar operations. Introduced for general matrix multiply (GEMM) acceleration, they offer low-precision modes from FP16 through INT8 and INT4 down to 1-bit integers [NVI25a]. We exploit this 1-bit mode for GF(2) polynomial multiplication (see Section 4.4). We take the RTX 4090 as the reference device because the state-of-the-art GPU HQC work we compare with, HIGH [DHW+26], and most recent GPU PQC studies [DFQ+24, WDX+25] report on it. Section 6.4 then confirms that the same results carry over to the older RTX 3090 and the embedded Jetson Orin.

The second axis that differentiates performance is the memory hierarchy. It runs from per-thread registers, through the on-chip shared memory (SMEM) that a block shares, to off-chip global memory (GMEM), with latency rising down the tiers. When a thread's local variables overflow the registers, they spill to the local stack that is backed by GMEM. Thus, a growing per-thread working set silently turns into GMEM traffic. Hence, data placement in the hierarchy is a critical factor that decides performance.

Unless stated otherwise, timings are kernel times on the RTX 4090 of Table 9. Throughput is the number of instances N in a batch divided by the kernel time of that batch, at the peak of a batch sweep, and the N=1 latency is the kernel time of a single instance, so the two are not reciprocals of each other.

The GPU hides memory latency by switching immediately to another resident warp while one warp waits on memory operations. Warps are resident on a Streaming Multiprocessor (SM), and the *occupancy* is how close the number of resident warps is to the hardware maximum. A kernel that demands many registers or much SMEM keeps fewer warps resident, lowering the occupancy and exposing memory latency. The two goals of GPU optimization are therefore to use few resources and to keep data in the fast tiers. Each variant is then bound by a different one of these resources. CG is bound by GMEM bandwidth and the local stack, FG and FGmw by shared-memory capacity and the occupancy it caps, and BMMA by the throughput of the tensor-core multiply instruction. These limits drive the performance divergence quantified in Section 6.3.

3 Mapping HQC Instances onto GPU Parallelism

The key design choice is how much GPU parallelism to assign to a single KEM instance, and this assignment makes each variant trade off throughput and latency differently. Concentrating more threads on one instance lowers the latency, which pays off at small

	FAFFT family			tensor-core family	
	CG	**FG**	**FGmw**	**BMMA**	**BMMAw**
	1 TH	1 W = 32 TH	4 W, 2 at HQC-5	P0 MMA P1	wP0 MMA wP1
mapping	1 thread/KEM	1 warp/KEM	1 block/KEM	batch-wide phases	batch-wide phases
working set	GMEM	SMEM	SMEM	registers + tiles	registers + tiles
non-mult.	serial in-thread	warp-parallel	warp-parallel	serial in-thread	warp-parallel

Figure 1: Thread-to-KEM mapping taxonomy of the five variants.

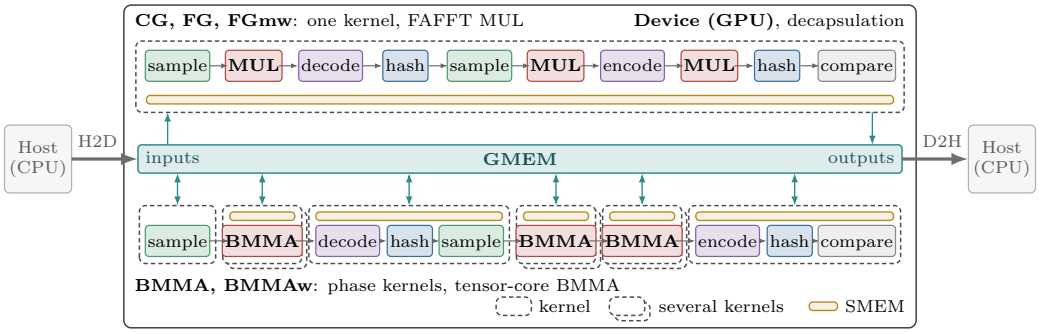


Figure 2: End-to-end overview of our GPU-resident HQC KEM, shown for decapsulation.

batches. Conversely, spending less parallelism per instance and filling the GPU with more instances raises throughput at large batches. Which operating point is preferable depends on the deployment's batch size and latency bound, so we provide the five variants as flexible choices, which are illustrated in Figure 1. The five variants fall into two families. The first three, CG, FG, and FGmw, use the FAFFT of Section 2.3 and differ only in the cooperating unit, a thread, a warp, or a block, and this cooperation granularity trades off against occupancy. The "mw" in FGmw denotes multi-warp. The last two, BMMA and BMMAw, move the multiplication onto the INT1 tensor cores and batch it across instances, and BMMAw adds the warp techniques applied in FG.

Figure 2 shows the end-to-end flow. The host copies the inputs to the device, launches the kernels, and copies the outputs back, and every intermediate value stays in device memory. CG, FG, and FGmw run an operation as one kernel, and BMMA and BMMAw split it into phase kernels at the multiplication boundaries.

- **CG (1 thread/KEM).** A single thread runs an entire instance serially, so the per-instance latency is the highest. However, thousands of independent threads fill the GPU at maximal occupancy, and CG leads large-batch throughput at every level. Its per-thread working set is too large for the block-shared SMEM, so CG keeps it in GMEM.

- **FG (1 warp/KEM).** One warp cooperates on one instance, dividing the FAFFT, Keccak, and coding among themselves with the operands in SMEM, so per-instance latency is lower than CG's. SMEM usage for one instance limits the resident warps, so large-batch throughput falls short of CG. Relying only on warp shuffles, FG needs no special configuration and is also portable.

- **FGmw (1 block/KEM).** The warps of one block, four at HQC-1 and HQC-3 and two at HQC-5, cooperate on a single instance, using the widest per-instance parallelism and thereby giving the lowest small-batch latency among the FAFFT

variants. But each instance's larger SMEM share reduces the number of instances co-resident on an SM, so its large-batch throughput falls short of CG.

- **BMMA (batch-wide phases).** The INT1 tensor-core instruction is efficient only when its tiles are filled across thousands of instances, so the KEM is cut into phases at the multiplication boundaries. A phase P0 prepares the operands, one batched kernel multiplies all N instances, and a phase P1 resumes the non-multiplication work. One KEM operation thus runs as several kernels, yet the data stays on the GPU, whereby the host only launches them in a fixed order. Tensor-core multiplication makes both batched throughput and single-shot latency appealing.

- **BMMAw (batch-wide phases + warp non-multiplication).** BMMA with its P0 and P1 non-multiplication phases parallelized across a warp instead of a single thread. Multiplication being already batched, the non-multiplication work is the remaining bottleneck, and the warp techniques applied in FG remove it, so BMMAw ranks first in small-batch latency at every level.

4 Binary Polynomial Multiplication on the GPU

4.1 Multiplier Selection and the Design Space

Which combination of multiplication methods excels is hard to know without running them directly on the GPU, because the memory hierarchy, hardware-resource occupancy, and kernel configuration interact differently than on CPUs and ARM. The GPU lacks dedicated field instructions such as GFNI or PCLMULQDQ, so the same computation unravels into shift-XOR chains, and the thread/warp/block mapping governs occupancy and latency hiding. Hence, we ported four prior FAFFT implementations, Chen's single instruction, multiple data (SIMD) implementation [CCPY26] and three ARM Cortex-M4 implementations [CCC+26, JLK+25, LJKH25], to the GPU, measured all of them on HQC-1 CG, and analyzed each to determine the fastest combination on the GPU.

Referring to the results shown in Figure 3, the biggest factor is the coefficient representation. Chen's SIMD implementation [CCPY26] chose its fields to match CPU vector instructions, which a GPU lacks, making them unsuitable for a GPU. The non-bitslice methods achieve only 65–682k polymul/s. In contrast, the bitsliced representation was first combined with the FAFFT implementation [CCK21], carried to the HQC tower FAFFT following [LJKH25], and extended with the split and tail technique proposed by Jang [JLK+25] and Chen-M4 [CCC+26]. This representation stores 32 GF(2^{32}) elements bit-sliced across 32 words, each word holding one bit position of all 32 elements, so a single word-wise XOR or AND advances all 32 at once. The prior bitsliced methods reach 1,650–2,406k polymul/s, while our combination ($\star$) reaches 2,990k, the highest. Reverting only the representation to non-bitslice slows the schoolbook split (SB) from 2,990k to 937k, a 3.2× reduction.

Within the bitsliced methods, the splitting method is the next factor (Figure 3(b)). Schoolbook applies a 32768-point body FAFFT but fills only its 16384-bit half, and handles the remaining cross terms as lightweight radix-16 tail products. It achieves the fastest throughput at 2,990k polymul/s. Karatsuba 2-way calls the same FAFFT twice, so the body working buffer measured in a controlled sweep grows from schoolbook's 6–19KB to 25–27KB, increasing register spill, and the throughput saturates at 2,109k polymul/s. CRT calls the FAFFT only once and reconstructs the result through the CRT. The pre- and post-processing and the reconstruction cost outweigh schoolbook's light tail, giving 2,406k with Chen and 2,061k with Jang. This throughput ordering is based on the peak reached by scaling up the GPU batch, and the differences from register spill and occupancy surface as the batch grows.

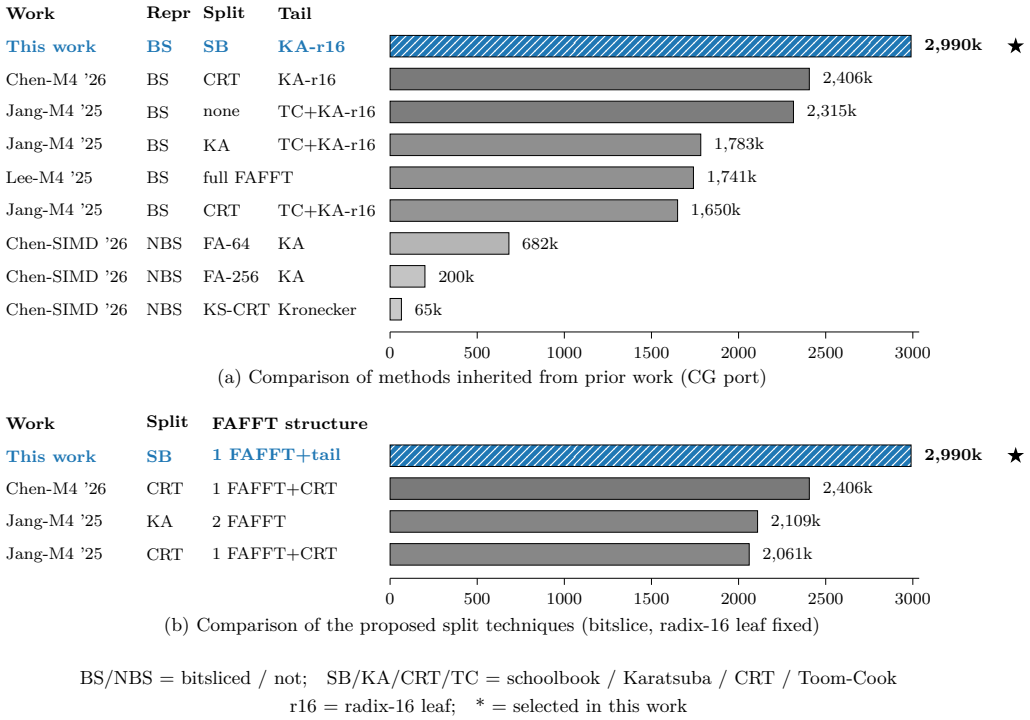


Figure 3: Polynomial Multiplication design space on HQC-1 CG (RTX 4090).

Removing the tail with one more butterfly layer is the design of the Lee row in Figure 3(a). On HQC-1 it shortens the single-shot latency by 1.4× but, with a per-thread working set of 33 instead of 22 KB, lowers the throughput by 2.4× in a fixed-configuration sweep, so CG keeps the split and BMMA covers the latency.

A method that multiplies everything by recursive Toom-Cook alone, without any FAFFT (Figure 3(a)), is also competitive in bitsliced form achieving 2,315k polymul/s, yet does not surpass the schoolbook method, which computes the body with the FAFFT. The best combination in this controlled sweep is bitsliced, schoolbook split, tower FAFFT, Karatsuba tail, and radix-16 leaf, reaching 2,990k polymul/s, 1.24× ahead of the best prior method, the Chen-M4 CRT at 2,406k, unlike on the ARM Cortex-M4 where CRT leads. This characterization is for HQC-1 CG. Table 8 validates the selected combination at all levels. No prior work adopts this combination. The schoolbook split and the Karatsuba tail are elements used by Chen [CCPY26], and the bitsliced tower FAFFT and the radix-16 leaf by Jang [JLK+25] and Chen-M4 [CCC+26], each previously evaluated on a different platform. This section's contribution is to identify the GPU-governing factors, namely representation bit-parallelism, the number of FAFFTs with the resulting register spill, and tail/reconstruction cost, and to derive a new and optimal combination that fits the GPU architecture.

4.2 Optimizing the CG Multiplier

The schoolbook split makes CG's multiplication a three-part computation. Writing each input as a power-of-two body plus a high-degree tail, $a = a_b + X^m a_t$ and likewise b with m half the body transform size of Table 2, the product is the body $c_b = \mathrm{FAFFT}(a_b, b_b)$ from the FAFFT of Algorithm 2, the cross and tail terms $c_t = (a_b b_t \oplus a_t b_b)X^m \oplus (a_t b_t)X^{2m}$ from radix-16 Karatsuba, and a final reduction $\mathrm{Reduce}(c_b \oplus c_t) \bmod (X^n - 1)$. For HQC-5

Table 4: Per-phase profiling of one CG multiplication (RTX 4090).

	% of one multiplication		
Phase	HQC-1	HQC-3	HQC-5
Karatsuba tail	**55.9**	**76.4**	–
Butterfly (forward/inverse)	28.4	13.1	**59.7**
Basis conversion (forward/inverse)	10.4	7.5	30.5
Pointwise multiplication	1.8	1.0	3.6
Truncated enc/dec	1.5	0.9	2.9
Reduce	2.0	1.2	3.2
Batch size N	512	256	128

Table 5: Radix-4 butterfly fusion over the radix-2 baseline, CG (RTX 4090).

	Throughput (polymul/s)				N=1 latency (μs)		
Level	radix-2	radix-4	at N	Gain	radix-2	radix-4	Gain
HQC-1	3,292k	**3,828k**	12,288	1.16×	2,268	**1,659**	1.37×
HQC-3	889k	**961k**	8,192	1.08×	7,353	**5,525**	1.33×
HQC-5	453k	**825k**	5,120	1.82×	7,752	**3,460**	2.24×

the tail is empty, leaving only the body FAFFT and the reduction. Both inputs enter as full n-bit vectors, also when one of them is a sparse secret, so no loop bound or address depends on its weight.

Table 4 shows the per-phase profiling of one multiplication, and the bottleneck differs per level. The Karatsuba tail dominates HQC-1/3 (56% and 76%), and the butterflies dominate HQC-5 (60%). We optimized the CG implementation along two axes matched to these bottlenecks, compiler code generation and the butterflies. Firstly, FAFFT selects one of several twiddle-product routines at each butterfly layer. Function-pointer dispatch is an indirect call the compiler cannot inline across, so register reuse and constant propagation break between layers. Replacing it with the direct calls of an inline-switch dispatch lets nvcc inline each routine into the butterfly and reuse registers across layers, raising the multiplication by +7.8%, from 2,990k to 3,224k. In CG the coefficient array lives in per-thread local memory, the stack, and reaches 22, 40, and 74 KB per multiplication for HQC-1/3/5. As in Section 2.3, each butterfly layer reads and writes it once. Nsight Compute shows that the HQC-1 CG multiplication is memory-latency-bound. With the arithmetic logic unit (ALU) at 27% and DRAM at 15%, neither compute nor bandwidth is the bottleneck, yet the top stall reason is waiting on local-memory loads at 52.2%. The multiplication cost therefore grows with the number of passes over the array, i.e., the number of layers.

Radix-4 butterfly fusion merges two adjacent layers into a single sweep. Layers k and k+1 pair the same radix-4 group of four coefficients into two pairs, each layer forming a different pairing, so reading those four coefficients into registers once, applying the two layers back to back, and writing once eliminates the store–reload between the layers. We fuse only up to radix-4, that is, two layers. Fusing r layers holds 2^r coefficients in registers at once, and past radix-4 this per-thread working set spills back to local memory, cancelling the traffic the fusion is meant to save. We show HQC-5, the representative case where the butterflies dominate. Of its twelve layers, only the upper seven access data word-wise and are fused, as above, into three adjacent pairs, leaving four passes. The lower five rearrange bits within a word and cannot be fused, staying five passes. Together this cuts the total from twelve passes to nine. Because each layer is an $\mathbb{F}_2$-linear map, applying two of them in registers computes the same composed function, so the result is bit-identical. The gain scales with the butterfly share, from 1.82× in throughput and 2.24× in single-shot latency for HQC-5, where butterflies dominate at 60%, down to 1.08–1.16× for HQC-1/3, where

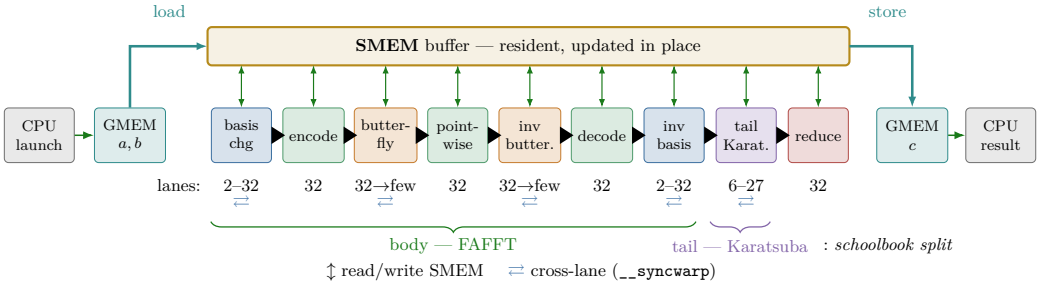


Figure 4: The FG multiplication as one GPU-resident kernel.

the heavy tail dilutes it. Table 5 lists the per-level throughput and latency. The gain exceeds the 12/9 pass ratio because each fused pair also removes the dependent store–load round-trip between the two layers.

4.3 FG/FGmw: Warp- and Block-Level Parallelization of the FAFFT

As shown in Section 4.2, CG is bound by the local-memory spill of its 22–74KB per-thread state. FG and FGmw, a 32-lane warp and a multi-warp block respectively, cooperate on the *same* multiplication and address that bottleneck from two directions. They keep the working arrays resident in SMEM, eliminating the spill, and they divide each stage of the multiplication across lanes, reducing single-shot latency (Figure 1). This is the first cooperative warp/block parallelization of FAFFT multiplication on a GPU, and this thread→warp→block mapping creates the throughput–latency operating points of Section 6. How to split a stage of the FAFFT across the 32 lanes depends on that stage's data dependencies. A split that ignores them forces synchronization that erases the parallel gain.

FG has the 32 lanes of one warp cooperate on every stage of Algorithm 2. The FAFFT stages split by how their data crosses lanes. Encode, pointwise multiplication, decode, and reduce have mutually independent output values, so 32 lanes taking values at stride 32 parallelize them with no communication. In contrast, basis conversion, the butterflies, and the tail Karatsuba have inter-lane dependencies and their parallelization requires careful design. Figure 4 maps the per-stage lane occupancy of the whole pipeline.

Basis conversion maps the input coefficients into the basis the FAFFT butterflies use, in three passes whose first two fold words of the buffer into one another while the last folds within 256-bit blocks. Although it looks element-wise, each pass folds words into one another, so it does not split directly across lanes. In the first pass layer 0 uses all 32 lanes, but later layers fold progressively fewer word pairs. The second pass is an XOR-fold in which each layer reads the upper half of the buffer and writes it back in place. If all 32 lanes ran at once, a lane could read a word not yet updated, so each layer is split into two steps, fold one half, synchronizing with `__syncwarp`, then folding the rest. Only in the third pass are the 256-bit blocks fully independent of one another, and with one block per lane all 32 lanes are filled. The inverse basis conversion differs. The product spectrum is twice as large, the reverting fold has no read–write conflict, so it does not need two-step split, and the layers run in reverse order.

The butterfly is the core stage of the FAFFT. Each layer sweeps the whole buffer, pairing and combining points at a fixed stride. In the upper, long-stride layers the stride is at least 32 words, so the two points of a butterfly pair lie in different lanes' chunks. Each lane multiplies its chunk by the layer twiddles, and after a `__syncwarp` the 32 lanes complete the XOR combination across chunks. In the lower, short-stride layers the stride

drops below 32 words and the paired points fall within one lane's chunk at sub-word bit positions, where a plain word XOR cannot reach them. Each lane therefore applies SWAPMOVE, a masked shift-and-XOR, to bit-transpose into word alignment, and the twiddle product and XOR then proceed as above.

The tail Karatsuba is divided into several independent Karatsuba subtrees, whose number falls short of 32, so some lanes are left idle. Processing one tail with a warp runs in three stages. First, all 32 lanes build the subtree inputs together. Next, one lane per subtree evaluates it, so only as many lanes as there are subtrees are active in this stage. Finally, all 32 lanes merge the results into the output. Because the middle stage where lanes idle is the bottleneck, packing several tails into one warp to raise the subtree count fills the idle lanes, for a cumulative in-KEM gain of +19–25%. FG keeps its working arrays compact by reusing buffers whose work is done, 17.5KB of SMEM per warp at HQC-1, but its parallelization across lanes is uneven. The later basis-conversion and lower butterfly layers use only 2–16 of the 32 lanes and the tail sub-products only 6–27, so many lanes idle. This idle-lane ceiling on single-warp cooperation is what FGmw widens by moving to a block.

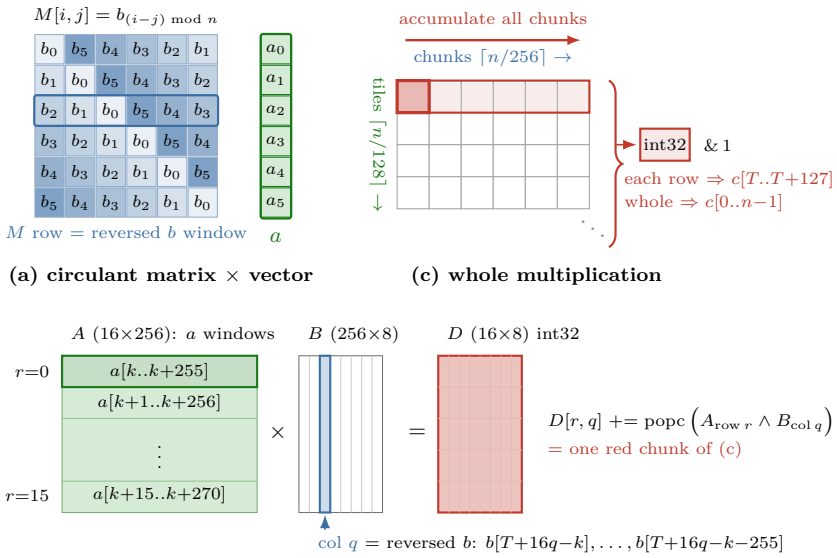
FGmw uses the warps of one block, but does not widen the butterfly to the block. The butterfly is already split by position within a warp, and widening it to the block would replace the per-layer `__syncwarp` with a block-wide `__syncthreads`, whose synchronization cost exceeds the parallel gain. Instead it adds two other kinds of parallelism across the warps. The first is task parallelism. The forward FAFFTs of a and b are mutually independent, so they run concurrently on two warps. The second is iteration parallelism. The tail cross-term loop is divided roughly evenly over the four warps. Because the iterations overlap and several warps touch the same output word, each warp first writes its result to a separate edge buffer, and after `__syncthreads` one warp XOR-merges them. Besides these two, the pointwise multiplication, the inverse FAFFT, and reduce run on one warp as in FG. For small batches with few instances ($N \leq 256$), a 4-warp kernel that maximizes the warps per instance to reduce latency is better, while for large batches where instances fill the GPU ($N \geq 512$), a 2-warp kernel in which one warp runs most of the work and only the cross terms are split across two warps is better, since there the warp split's synchronization cost exceeds the gain and also lowers occupancy. The host picks between the two by batch size. For HQC-3, whose decaps SMEM is tight, even the forward FAFFTs run serially on one warp.

The N=1 single-shot multiplication takes about 147 μs with FGmw and about 244 μs with FG, making FGmw 40% shorter, and for multiplication alone FGmw also leads FG in throughput. In the full KEM, however, FGmw uses a whole block while FG uses only one warp, so under the heavy register pressure FG's lower resource use wins back the HQC-1 throughput (Section 6.2). Furthermore, warp cooperation lowers the in-KEM multiplication share from ∼85% to about 22% for encaps and decaps, so the non-multiplication load of Section 5 emerges as the dominant bottleneck.

4.4 INT1 Tensor-Core Binary Toeplitz Multiplication

Another path entirely different from the FAFFT is to view the multiplication as a tensor-core matrix product. Prior work TMVP-Saber [HLKH23] also implemented a Toeplitz matrix–vector product on tensor cores, but for the integer modular arithmetic of Saber. Ours is the first mapping of binary GF(2) polynomial multiplication onto INT1 tensor cores.

Construction. The circular product $c[i] = \bigoplus_m a[m] \wedge b[(i-m) \bmod n]$ can be viewed as the product $c = M \cdot a$ of the circulant Toeplitz matrix $M[i,j] = b[(i-j) \bmod n]$ with the vector a (Figure 5(a)). Row i of M is b reversed and rotated by i positions, so one output coefficient is the GF(2) inner product of a with that row. This inner product matches



(a) circulant matrix × vector **(c) whole multiplication**

(b) one instruction `mma.sync.m16n8k256`

Figure 5: INT1 tensor-core Toeplitz multiplication.

exactly the 1-bit tensor-core instruction `mma.sync.m16n8k256.b1.b1.s32.and.popc`. The AND handles the coefficient products and the *popcount* counts the ones. The XOR sum we want is exactly the low bit of the integer sum, so accumulating in a 32-bit integer and keeping only the low bit gives the GF(2) value. One instruction computes 128 such 256-bit inner products at once and accumulates them into a 16×8 accumulator tile, which is one chunk of 128 consecutive coefficients of the output c (Figure 5(b)). The `m16`, `n8`, `k256` of the instruction name are exactly this 16×8 output tile and the 256-long inner product. It is the largest 1-bit MMA the Ampere and Ada GPUs provide, which minimizes the number of instructions per multiplication. This instruction requires sm_80, the Ampere generation, or later [NVI25b].

Tiling. Since one instruction contributes only one 256-bit chunk to one 128-coefficient tile, a whole multiplication is a lattice of such instructions (Figure 5(c)). The n output coefficients of c are cut into $\lceil n/128 \rceil$ tiles of 128 coefficients, and the n-bit inner product for each tile is cut into $\lceil n/256 \rceil$ 256-bit chunks with one instruction per chunk, so one multiplication becomes 139×70, 281×141, and 451×226 tensor instructions at the three levels. This lattice is the same for every operand, since a and b are taken as full n-bit vectors whatever their weight. This chunk count is also the number of tensor instructions accumulated into one accumulator entry. Since one instruction's *popcount* is at most 256 and the chunk count is at most 226, the accumulator peaks at about 57,856, well below 2^{31}, so the GF(2) reduction that keeps only the low bit is exact with no overflow. The reduction mod $X^n - 1$ is not a separate step. The b window index wraps mod n, so the reduction is already built into the matrix structure. The matrix M is never actually built either. Only the two bit vectors a and bit-reversed b are kept, and a single funnel-shift instruction extracts the 256-bit window at the desired position on the fly to fill a fragment. This fragment is the input form the tensor instruction takes, a 256-bit window of a or b held as a 1-bit-integer tile.

GPU mapping. Tensor instructions are warp-cooperative. A warp jointly holds the fragment words. One block takes TB output tiles of one polynomial, the S warps in the

Algorithm 3: INT1 tensor-core batched Toeplitz multiplication (BMMA).

Input: batch of polynomial pairs $\{(a_p, b_p)\}$, $p{=}1 \dots N$ // one per block
Output: $\{c_p = a_p \cdot b_p \bmod (X^n{-}1)\}$ // c spans $\lceil n/128 \rceil$ tiles of 128 coefficients
Data: each block covers TB output tiles with S warps

foreach *block* **do** // output tiles $T_0 \dots T_0{+}TB{-}1$
  $acc[tb] \leftarrow 0$, $tb = 0 \dots TB{-}1$ // register accumulators, $16 \times 8 = 128$ per output tile
  foreach *256-bit chunk* k **do** // shared by the S warps
    $A_k \leftarrow$ funnel-shift(a_p, k) // 16×256 fragment of a
    for $tb \leftarrow 0$ **to** $TB - 1$ **do**
      $B_{k,tb} \leftarrow$ funnel-shift(b_p, k, tb) // 256×8 fragment of b
      $acc[tb]$ += `mma.b1.b1.s32.and.popc`$(A_k, B_{k,tb})$ // AND = product, popcount = accumulation
    end
  end
  for $tb \leftarrow 0$ **to** $TB - 1$ **do**
    $acc[tb] \leftarrow$ XOR-merge of the S warps' partial accumulators in SMEM
    for $j \leftarrow 0$ **to** 127 **do**
      $c_p[128(T_0{+}tb) + j] \leftarrow acc[tb][j]$ & 1 // GF(2) reduction
    end
  end
end
return $\{c_p\}$

Table 6: The two BMMA configurations used in the evaluation.

Purpose	S	TB	Coeffs/blk*	Used for
Latency	14	1	128	N=1 multiplication and N=1 KEM latency
Throughput	2	8	1024	peak multiplication and KEM throughput

*Coeffs/blk (output coefficients per block) $= 128 \times TB$

block divide the chunks, accumulate partial sums, and XOR-merge them in SMEM, and the blocks of the whole kernel cover all output tiles $\times$ the batch N. The key remaining cost is the funnel-shift packing, which we reduce with three reuses. First, the A fragment is independent of the output tile, so it is extracted once per chunk and shared by all output tiles of the block. Second, within the same chunk the B fragments of neighboring output tiles overlap by half, which is handed over as is. Third, in the S=2, TB=8 configuration the B fragment built in one chunk is used again in the next, held briefly in a register and reused. Together the three reuses lift the peak throughput by 3.1$\times$ by removing the packing bottleneck. As the batch grows, packing's share of the multiplication time falls from 38.6% at N=1 to 2.0% at N=4096, because tensor-core multiplication grows more efficient as its tiles fill with more instances, so multiplication increasingly dominates the runtime and the packing cost is buried beside it. The full procedure is Algorithm 3.

The choice of the warps per block S and the output tiles per block TB trades latency against throughput. Table 6 fixes the two configurations used throughout the evaluation. Peak-throughput rows use the throughput setting, and N=1 latency rows the latency setting. The single-shot HQC-1 multiplication reaches $\sim$15 μs of latency, while the throughput setting reaches 1,940k polymul/s at N=4096 (Table 8).

5 Warp Parallelization of the Non-Multiplication Work

5.1 Warp-Keccak (SHA-3/SHAKE)

After multiplication is accelerated, hashing and sampling become the dominant bottleneck at ∼56% of the FG profile, as shown in Section 2.2. Every hash and XOF in HQC is a sponge that absorbs and squeezes over a single permutation, `Keccak-f1600` [D+15]. We unfold that permutation across a warp. The Keccak state is a 5×5 array of 25 64-bit lanes, i.e., 1600 bits. We map these 25 words one-to-one onto 25 lanes of a warp, leave the remaining lanes unused, and implement the round functions θ, ρ, π, χ, ι with data-independent-offset `__shfl_sync`. The warp-shuffle Keccak mapping itself is a known approach [CHS11, LWGP17], since adopted by lattice-based GPU work such as Kyber [OBS21] and Dilithium [SYD+24].

The shuffle sources are fixed lane indices, so no lane is selected by a value of the state, and the permutation becomes about 2.3× faster than the serial reference.

5.2 Warp-Parallel Secret-Independent Fixed-Weight Sampling

The HQC specification defines two samplers with different parallelization potential. What distinguishes the two is whether the length of the XOF randomness consumed is fixed.

The sampler used by encaps and decaps re-encryption is the specification's SampleFixedWeightVect. It has no rejection and consumes exactly one word of randomness for each 1 in the vector. Since the consumption length is fixed, all required randomness can be expanded in advance with a single warp-Keccak squeeze of Section 5.1, and the three subsequent stages are placed on the warp. First, the transform that maps each random word to the position of one 1 is independent across elements, so these positions are distributed over the lanes at stride 32. Second, deduplication has a serial outer index loop because of its in-place updates, but the heavy part of each step, checking whether a position collides with the ones that follow, is swept by the 32 lanes and reduced within the warp, after which one lane performs only a branch-free swap. Third, the settled positions are written into the bit vector, a write shared by both samplers. Writing a secret position directly to its address would leak it through the data-dependent address, so we use a masked gather instead. It visits every output word in a fixed order and, for each secret position, decides with a branch-free equality mask whether that word holds the position, accumulating the bit. The address accessed is then independent of the secret, so the device arithmetic is data-independent. We divide this output-word loop across the warp at stride 32. This lowers the single-shot encaps latency of FGmw at HQC-5 from 17,545 to 2,191 μs. The sampler that draws keygen's (x, y) and decaps's y, in contrast, is a rejection sampler. It draws one 24-bit candidate and, if the candidate is out of range or duplicates an already-chosen position, discards it and draws again. Because it cannot be known in advance when a valid candidate appears, the length of randomness consumed varies, and only a serial XOF that squeezes as much as needed on demand can be used, so the candidate-generation loop stays on a single lane. This rejection loop is data-dependent by the specification itself, and we keep it as in the reference. Even so, the heaviest part of keygen, the write across the whole bit vector, is distributed across the warp, which lowers the single-shot keygen latency of BMMAw at HQC-5 from 4,607 to 1,010 μs.

5.3 Warp-Parallel RM/RS Coding

We distribute the output bytes of RM/RS encoding and decoding over the 32 lanes, so that adjacent lanes read consecutive bytes and the memory accesses coalesce. Figure 6 shows the decoding pipeline and where each stage is warp-parallelized. RM(1,7) decoding is independent per 128-bit block and therefore maps naturally onto a warp. Its three

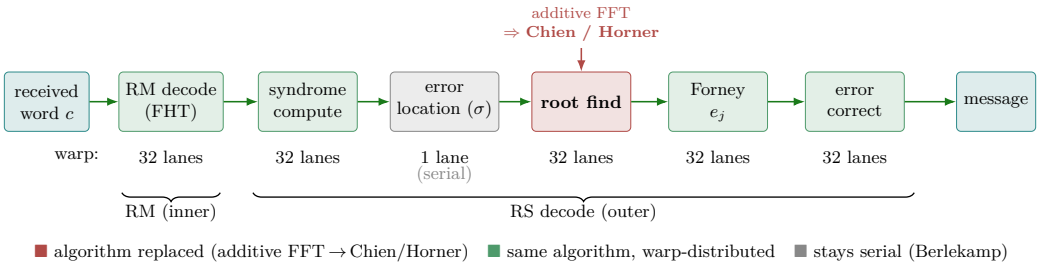


Figure 6: HQC decoding pipeline and its per-stage warp parallelization.

Table 7: Warp non-multiplication ablation on the FG path (RTX 4090).

			peak ops/s
Level / op.	Baseline	+RM/RS warp	+warp-Keccak
HQC-1 encaps	285,276	284,919 (−)	**371,626** (+30.4%)
HQC-1 decaps	104,182	137,993 (+32.5%)	**182,199** (+32.0%)
HQC-3 encaps	72,422	73,877 (−)	**94,716** (+28.2%)
HQC-3 decaps	28,772	34,080 (+18.5%)	**45,446** (+33.4%)
HQC-5 encaps	37,258	37,364 (−)	**47,629** (+27.5%)
HQC-5 decaps	14,517	17,096 (+17.8%)	**21,573** (+26.2%)

stages, expand-and-sum, the fast Hadamard transform, and the peak search, are divided among the 32 lanes byte by byte, and the peak selection keeps the branch-free form of the reference implementation, an arithmetic mask for the absolute value and a select for the running maximum. Once multiplication and hashing are also warp-parallelized, RM/RS decoding rises to about 20% of decaps and becomes one of the remaining large pieces, within which the root finding of the RS decoder is the heaviest part. Since the roots of the error-locator polynomial $\sigma(X)$ are exactly the error positions, root finding reduces to evaluating σ at every candidate point and keeping the zeros. The reference implementation [GAMA+25] finds the roots with an additive FFT, evaluating a σ of at most 32 coefficients at the 256 points of GF(2^8). Its butterflies are essentially the same as the FAFFT of the FG multiplier (Section 4.3) and could therefore be warp-parallelized by lane cooperation. Most prior GPU work, however, does not process this transform with warp cooperation [DHW+26, WDX+25, JG25], and where it is cooperative the target is a much larger McEliece transform [XGZW25]. Because the additive FFT used in the RS decoder is small, its per-layer synchronization overhead is not spread over enough work, unlike in FG's large transform. We compared this warp-parallelized additive FFT head-to-head with a warp Chien search, finding the warp Chien at least as fast at all three levels in the full decaps, and up to about 17% faster at HQC-3 decapsulation. We therefore choose the warp Chien search. It evaluates σ at every candidate point and keeps the points where it is zero. Each lane takes one distinct candidate point and computes its σ value independently with Horner's rule [SER11]. Since the candidate points do not depend on one another, the warp stays full with no extra synchronization, produces the same evaluations as the additive FFT, and reuses the subsequent root test and error-position mapping unchanged. The computation of σ by Berlekamp's algorithm and the error-value computation are strongly dependent and stay serial on a single lane, but the Forney error-value loop is again distributed across the warp. This warp RM/RS coding accelerates FG decaps by +18–33% over the baseline (Table 7), and FGmw uses the same technique.

Finally, the serial copies at the kernel boundaries, packing the key and ciphertext bytes into the output buffer at the end and reading the input ciphertext at the beginning, are likewise distributed over the 32 lanes at byte stride and handled as coalesced transfers.

Table 8: Standalone polynomial multiplication throughput and N=1 latency (RTX 4090, kernel-only).

Level (n)	CG	BMMA	FG	FGmw
Peak throughput (polymul/s), throughput-optimal configuration				
HQC-1 (17,669)	**3,828k**	1,940k	1,653k	2,022k
HQC-3 (35,851)	**961k**	498k	426k	508k
HQC-5 (57,637)	**825k**	196k	176k	312k
Batch size N of the peak				
HQC-1	12,288	4,096	32,768	32,768
HQC-3	8,192	1,024	6,144	8,192
HQC-5	5,120	256	3,072	10,240
N=1 latency (μs), latency-optimal configuration				
HQC-1	1,659	**15**	244	147
HQC-3	5,525	**15**	715	323
HQC-5	3,460	**25**	731	416

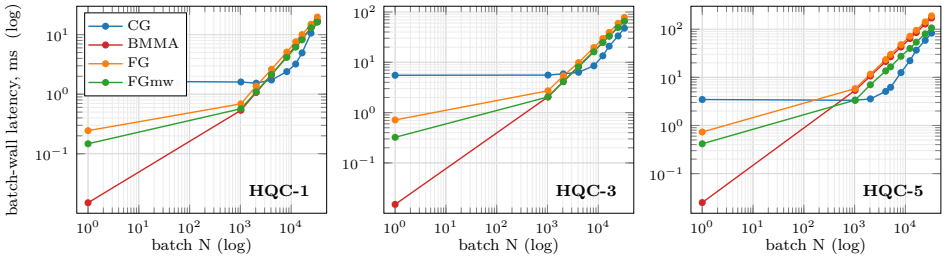


Figure 7: Standalone polynomial multiplication latency versus batch size (RTX 4090).

Since this costs no shuffles and no extra registers or SMEM, it parallelizes readily at the warp level, and keygen speeds up by +11–14% and encaps/decaps by +2–6%.

Table 7 takes as its baseline the KEM with only the FG multiplier and the warp fixed-weight sampling of Section 5.2 applied, and adds warp RM/RS coding (Section 5.3) and then warp-Keccak (Section 5.1) in turn, showing the effect of each step. RM/RS warp accelerates the decode side, so its effect on encaps is within noise (marked −). Warp-Keccak targets fixed-size hashing. The cost of warp RM/RS does grow with the security level, but the throughput gain from multiplication grows even more, so the RM/RS share is diluted and its relative gain shrinks as Table 7 shows.

6 Evaluation

6.1 Setup

Our implementation is written to avoid secret-dependent memory addresses and branches by the measures of Sections 4 and 5. Correctness is verified by regenerating the known-answer tests (KATs) provided by the HQC team. Performance is measured with CUDA events for kernel-time throughput and latency. We report both measurement boundaries across Sections 6.2–6.3, where kernel-only is our headline GPU-resident boundary that excludes host↔device transfers and launches, and end-to-end adds them and is timed as wall-clock. The standalone polynomial multiplication of Table 8, whether throughput or N=1 latency, is only a component operation of the KEM and is therefore reported kernel-only.

Each data point is the median of repeated runs after warm-up. Exact grid/block

Table 9: Performance-evaluation environment.

	RTX 4090 (Ada)	RTX 3090 (Ampere)	Jetson AGX Orin (Ampere, embedded)
Compute capability	sm_89	sm_86	sm_87
SMs	128	82	14
GPU clock (base/boost)	2235/2520 MHz	1395/1695 MHz	930 MHz (max)
SMEM per SM	100 KB	100 KB	164 KB
Global memory	24 GB	24 GB	32 GB
TDP	450 W	350 W	15–40 W
Host CPU	i7-13700K	i7-13700K	ARM Cortex-A78AE
CUDA	13.1	13.1	11.4 (JetPack 5)
Compiler	`gcc -O3 -march=native` (CPU), `nvcc -O3` (GPU)		
OS	WSL2	WSL2	L4T (Ubuntu)

Table 10: KEM phase profile of the five variants, share of kernel time in %.

Variant	Op.	HQC-1				HQC-3				HQC-5			
		Mult.	Hash & Sample	Code	Other	Mult.	Hash & Sample	Code	Other	Mult.	Hash & Sample	Code	Other
CG	keygen	68	9.3	0	23	78	8.8	0	13	59	24	0	18
	encaps	75	14	0.4	10	82	11	0.9	6.1	43	51	1.6	4.8
	decaps	73	14	5.8	7.6	81	11	2.6	5.2	58	29	6.5	7.2
FG	keygen	56	44	0	0.7	53	46	0	0.5	59	40	0	1.1
	encaps	73	22	4.4	1.0	75	22	2.6	0.9	75	19	3.4	2.1
	decaps	54	26	12	7.2	55	31	7.1	7.1	52	31	10	7.0
FGmw	keygen	44	56	0	0.5	48	51	0	0.3	51	48	0	0.7
	encaps	59	33	6.4	1.4	70	25	4.0	1.0	69	25	4.2	1.8
	decaps	40	31	21	8.4	49	32	11	7.7	43	34	16	8.0
BMMA	keygen	71	6.2	0	23	81	5.6	0	14	85	5.0	0	9.6
	encaps	74	7.7	0.3	18	83	6.1	0.1	11	87	6.3	0.1	6.7
	decaps	76	9.3	4.4	10	86	6.9	1.9	5.8	90	4.9	1.4	4.0
BMMAw	keygen	51	17	0	32	63	12	0	24	75	8.8	0	16
	encaps	82	12	2.1	3.6	90	7.1	0.8	2.3	92	5.4	0.6	1.7
	decaps	56	18	23	3.4	69	20	8.5	2.7	75	14	8.7	2.0

configurations, peak batch sizes, and stack limits are fixed in the artifact build scripts.

6.2 Multiplication Performance and KEM Throughput

Table 8 reports the standalone polynomial multiplication peaks and $N{=}1$ latencies. CG owns the throughput end, reaching 3,828k, 961k, and 825k polymul/s at HQC-1, HQC-3, and HQC-5, BMMA owns the latency end at 15, 15, and 25 μs, and FG/FGmw form intermediate operating points. Figure 7 shows how these latencies scale with batch size.

Folding these multipliers into the full KEM, we place the two measurement boundaries of Section 6.1 side by side in Table 11. The kernel time measured inside each end-to-end run agrees with the separate kernel-only measurement to within ±2%.

CG leads in every cell at both boundaries, and the reason lies in the mapping itself (Section 3). With one thread per KEM, CG has zero cooperation overhead. There is no synchronization and no shared-memory partitioning. As the batch grows it simply fills the SMs with independent KEMs. FG/FGmw, in contrast, pay in throughput for the synchronization of lane cooperation and the idle-lane cost of non-uniform stages (Section 4.3), and BMMA pays for packing and the phase-kernel boundary cost (Section 4.4).

Even end-to-end we exceed HIGH [DHW+26]. At HQC-1, CG reaches 1,429k, 909k, and 758k ops/s for keygen, encaps, and decaps, which are 1.56×, 1.55×, and 2.03× over HIGH's 917k, 587k, and 374k.

Table 11: KEM peak throughput and single-shot N=1 latency (RTX 4090).

		throughput kops/s, latency ms; each cell kernel-only/end-to-end				
Level	Op.	CG	BMMA	BMMAw	FG	FGmw
Peak throughput (kops/s)						
HQC-1	keygen	**2,170/1,429**	1,391/1,031	1,028/862	778/617	658/538
HQC-1	encaps	**1,243/909**	747/546	804/629	371/334	258/224
HQC-1	decaps	**972/758**	508/407	364/317	181/172	109/105
HQC-3	keygen	**640/500**	400/328	337/298	144/136	156/146
HQC-3	encaps	**411/301**	209/173	228/202	95/89	103/97
HQC-3	decaps	**278/238**	143/125	116/108	46/44	48/46
HQC-5	keygen	**379/306**	171/142	155/135	76/69	91/83
HQC-5	encaps	**199/145**	87/76	94/83	47/44	58/54
HQC-5	decaps	**140/102**	59/53	51/46	21/20	23/23
Batch size N of the kernel-only peak						
HQC-1		16,384	32,768	32,768	65,536	65,536
HQC-3		16,384	16,384	16,384	65,536	65,536
HQC-5		8,192	16,384	16,384	16,384	16,384
Single-shot N=1 latency (ms)						
HQC-1	keygen	2.44/2.55	0.66/0.69	**0.26/0.30**	0.45/0.50	0.37/0.40
HQC-1	encaps	4.71/4.53	1.04/1.09	**0.25/0.33**	0.70/0.77	0.52/0.57
HQC-1	decaps	7.58/7.12	2.76/2.59	**0.92/0.89**	1.40/1.35	1.20/1.21
HQC-3	keygen	7.98/7.31	2.35/2.22	**0.57/0.58**	0.91/0.90	0.84/0.82
HQC-3	encaps	15.12/14.11	4.05/3.87	**0.42/0.46**	1.37/1.40	1.26/1.28
HQC-3	decaps	23.52/21.51	7.04/6.49	**2.15/2.01**	2.83/2.66	2.70/2.54
HQC-5	keygen	6.57/6.10	4.64/4.31	**0.91/0.88**	1.71/1.61	1.42/1.40
HQC-5	encaps	14.56/13.47	6.56/6.05	**0.61/0.65**	2.69/2.55	2.19/2.07
HQC-5	decaps	23.86/21.92	12.04/10.97	**3.11/2.86**	5.92/5.49	5.31/4.88

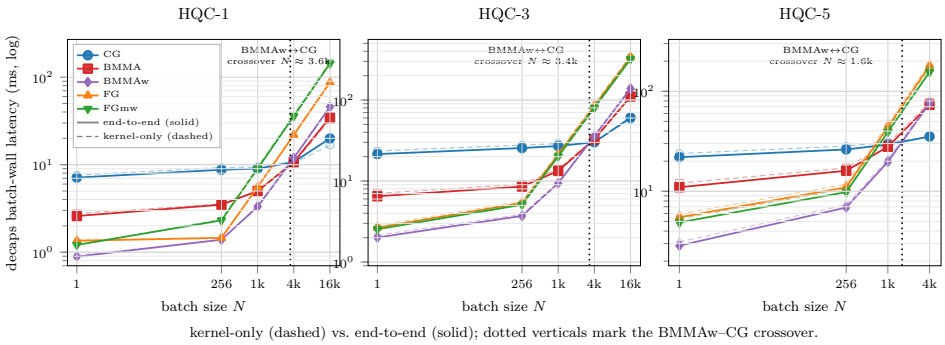


Figure 8: Decapsulation latency versus batch size N (RTX 4090).

Table 10 breaks the kernel time of each KEM operation, measured at saturation, into multiplication, hashing with sampling, RM/RS coding, and the rest. Multiplication takes 68–82% of CG at HQC-1 and HQC-3 and 71–90% of BMMA, while hashing and sampling take 40–56% of keygen in FG and FGmw. Against the intermediate stage of Table 3, the warp parallelization of Section 5 lowers hashing with sampling from 56.5% to 31% and RM/RS coding from 12.1% to 7.1% of FG decapsulation at HQC-3, so multiplication is again the largest share at 55%.

6.3 Single-Shot and Batch Latency

Single-shot N=1 is a diagnostic, not a target. A GPU earns its keep on batches, so the metric that matters is the small-N batch-wall latency of Figure 8. The key point is that

the throughput leader and the latency leader are *different* variants, so the right choice depends on the batch size. At small batches BMMAw has the lowest latency at every level, because its tensor-core multiplication and warp-parallel coding remove the serial bottlenecks that dominate a lone instance. As the batch grows the order inverts and CG takes over, filling the GPU with independent instances once the batch is large enough, with the crossover at a few thousand instances, near 1.6k at HQC-5 and 3.6k at HQC-1. The heavier level crosses over at a smaller batch because its larger per-instance work saturates the GPU with fewer instances, so CG takes the throughput lead sooner.

The variant trade-off is clearest between FG and FGmw. At HQC-1 decapsulation FG has higher large-batch throughput than FGmw. FGmw uses four times as many threads per instance but cuts single-shot latency by only about a tenth, so once the batch fills the GPU, FG uses fewer threads per instance and runs more instances at once, leading in throughput. At HQC-3 and HQC-5 the multiplication is heavier and FGmw's multi-warp cooperation earns its keep, so it edges back ahead. This trade-off reads directly off Figure 8. A single isolated call is faster on a CPU core, tens of microseconds with AVX2 as Table 12 shows, than on any GPU variant. The GPU's value is throughput under a latency bound, not single-shot response. A latency-sensitive service that batches sporadic requests chooses the FG and BMMA family, while bulk work such as mass key issuance chooses CG above the crossover. This choice is driven by the target batch size and latency, and can be made per operation and level.

6.4 Comparison with Prior Work

The most direct comparison target is HIGH [DHW+26], measured on the same RTX 4090. It reports only HQC-1 performance, where the latency appears only on the secondary axis of the chart and there is no numeric latency table. Table 12 discloses all five of our variants with throughput at both boundaries and single-shot latency, and its Mult. column gives the standalone polynomial multiplication peak. The table also lists the reference C and AVX2 software on one core of the host CPU. The throughput champion CG outruns the AVX2 single core by about 17× on polynomial multiplication and by 30–61× across the three KEM operations. On the throughput axis, our champion CG leads HIGH's best numbers by 1.55–2.03× end-to-end. On the latency axis no cell-level comparison is possible, so we present our numbers alone. BMMA/BMMAw and FG/FGmw target latency and small batches rather than throughput. With INT1 tensor cores BMMAw leads small-batch latency, but FG/FGmw, which do not rely on tensor cores, open the same low-latency end on a commodity GPU with CUDA cores alone. The purpose of our taxonomy is to span not just latency and throughput but this hardware availability as well.

Table 12 also lists two lightweight Artix-7 designs [ABPS24, PDB+26] and two GPU ML-KEM implementations [ZZF+24, JDD+24]. The FPGA designs generate an HQC-1 key in 30 and 20 μs, below BMMAw's 254 μs, yet one instance yields at most the reciprocal of its latency, 33k and 50k keygen/s against BMMAw's 862k. The difference from ML-KEM is algorithmic, since on one i7-13700K core HQC-1 decapsulation takes 218k cycles against 14k for the AVX2 Kyber-512.

To show the design is not tied to a single GPU, we ran the same five variants without modification on the RTX 3090 and Jetson AGX Orin of Table 9. The lower blocks of Table 12 give their HQC-1 results. On the embedded Jetson AGX Orin, whose SM count and memory bandwidth are far lower, absolute throughput drops, yet the relative ordering of the five variants is preserved. CG leads throughput and BMMAw leads small-batch latency on all three GPUs. This platform also makes clear why we accelerate the whole KEM rather than the multiplier alone. Its standalone multiplication is close to that of a single AVX2 core, since one large GF(2) product is exactly what the CPU's PCLMULQDQ instruction handles well and a small GPU cannot pull far ahead. Even so, CG's full KEM outruns that AVX2 core by 2.7–4.3× across the three operations, because scaled up to the

Table 12: HQC-1 comparison with prior work, and our variants across three GPUs.

Impl.	Device	Mult. (k/s)	KEM throughput (kops/s)			N=1 lat. (μs) keyg./enc./dec.
			keygen	encaps	decaps	
Ref C [GAMA+25]	i7-13700K	2	1.6	0.8	0.5	625/1250/2000
AVX2 [GAMA+25]	(1 core)	220	72	38	16	14/26/63
HIGH [DHW+26]	RTX 4090	1,318	917	587	374	—
Naive GPU port	RTX 4090	—	3.3	1.6	1.1	74555/149038/223556
HW [ABPS24]	Artix-7 FPGA	—	33.3	12.7	8.4	30/79/119
HW [PDB+26]	Artix-7 FPGA	—	50.0	25.0	12.3	20/40/81
ConvKyber [ZZF+24] Kyber-1024	RTX 3080	485,600	2,074	1,587	8,458	—
HI-Kyber [JDD+24] Kyber-1024	RTX 3080	68,418	1,916	2,205	6,092	—
CG	RTX 4090	**3,828**	**2,170/1,429**	**1,243/909**	**972/758**	2738/4989/8090
BMMA		1,940	1,391/1,031	747/546	508/407	707/1181/2994
BMMAw		1,940	1,028/862	804/629	364/317	**254/272/1012**
FG		1,653	778/617	371/334	181/172	435/693/1399
FGmw		2,022	658/538	258/224	109/105	354/500/1186
CG	RTX 3090	1,330	1,051/806	522/422	364/325	3605/6558/10816
BMMA		915	732/633	372/339	249/233	963/1581/4120
BMMAw		915	582/535	376/346	166/160	356/333/1365
FG		783	336/311	165/157	79/78	598/935/1916
FGmw		917	295/271	115/109	47/46	504/682/1623
CG	Jetson Orin	257	194/185	104/101	69/67	7433/13245/22539
BMMA		77	65/64	34/33	23/22	2151/3582/9340
BMMAw		77	53/52	33/33	14/14	746/688/2930
FG		75	46/45	22/21	8.5/8.4	1249/1932/4100
FGmw		82	24/23	16/16	6.8/6.8	1041/1456/3575

whole KEM the GPU's advantage shows in the throughput across all operations.

7 Conclusion

We presented the complete GPU-resident HQC KEM for all three security levels, implemented as five variants with different thread-to-KEM mappings. For multiplication, we explored the design space one controlled variable at a time and selected the bitslice, schoolbook and tower-FAFFT combination, then realized it along three paths, namely CG's radix-4 butterfly fusion, FG/FGmw's whole-pipeline warp parallelization, and the INT1 tensor-core Toeplitz. The hashing, sampling, and coding bottlenecks exposed once multiplication was solved were also parallelized at the warp granularity. As a result, CG leads throughput at HQC-1, 1.55–2.03× end-to-end over the prior GPU HQC, and on GPUs with INT1 tensor cores BMMAw leads small-batch latency. For versatility we also provide FG/FGmw, which do not rely on tensor cores and leave a low-latency path on commodity GPUs with CUDA cores alone. Since the real problem of GPU deployment is securing throughput within a latency bound, we provide the variants so that a deployment can choose the one matching its batch size, operation, and level.

Acknowledgments

We thank the editor, the shepherd, and the anonymous reviewers for their helpful comments during the review process. We also thank Ming-Shing Chen for his help on the FAFFT. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. RS-2025-25411243).

References

[ABB+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying {Constant-Time} implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, 2016.

[ABPS24] Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi, and Ruggero Susella. A high efficiency hardware design for the post-quantum KEM HQC. In *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 431–441, 2024.

[BSHST16] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. Fast multiplication in binary fields on GPUs via register cache. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*, pages 1–12. ACM, 2016.

[CCC+26] Ming-Shing Chen, Tun-You Chien, Chun-Ming Chiu, Han-Hsuan Lin, Chun-Tao Peng, and Bo-Yin Yang. Additive FFTs for HQC on ARM Cortex-M4, revisited. *Cryptology ePrint Archive*, 2026.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. *Cryptology ePrint Archive*, 2021.

[CCPY26] Ming-Shing Chen, Chun-Ming Chiu, Chun-Tao Peng, and Bo-Yin Yang. Accelerating HQC with additive FFT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):520–544, 2026.

[CHS11] Pierre-Louis Cayrel, Gerhard Hoffmann, and Michael Schneider. GPU implementation of the Keccak hash function family. In *International Conference on Information Security and Assurance*, pages 33–42. Springer, 2011.

[D+15] Morris J Dworkin et al. SHA-3 standard: Permutation-based hash and extendable-output functions. 2015.

[DFQ+24] Jiankuo Dong, Yusheng Fu, Xusheng Qin, Zhenjiang Dong, Fu Xiao, and Jingqiang Lin. Eco-BIKE: Bridging the gap between PQC BIKE and GPU acceleration. *IEEE Transactions on Information Forensics and Security*, 19:8952–8965, 2024.

[DHW+26] Jiankuo Dong, Yuze Hou, Shiqin Wang, Letian Sha, Fu Xiao, Zhenjiang Dong, and Jingqiang Lin. HIGH: Harnessing GPU parallelism for optimized HQC performance. *IEEE Transactions on Computers*, 2026.

[GAMA+25] Philippe Gaborit, Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, et al. Hamming quasi-cyclic (HQC). *NIST Post-Quantum Standardization, 4th Round; National Institute of Standards and Technology: Gaithersburg, MD, USA*, 2025.

[GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload–a cache attack on the BLISS lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 323–345. Springer, 2016.

[GM10] Shuhong Gao and Todd Mateer. Additive fast Fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, 2010.

[HLKH23] Muhammad Asfand Hafeez, Wai-Kong Lee, Angshuman Karmakar, and Seong Oun Hwang. TMVP-based polynomial convolution for Saber and Sable on GPU using CUDA-cores and tensor-cores. *Cryptology ePrint Archive*, 2023.

[JDD+24] Xinyi Ji, Jiankuo Dong, Tonggui Deng, Pinchang Zhang, Jiafeng Hua, and Fu Xiao. HI-Kyber: A novel high-performance implementation scheme of Kyber based on GPU. *IEEE Transactions on Parallel and Distributed Systems*, 35(6):877–891, 2024.

[JG25] Tanmayi Jandhyala and Guang Gong. Air-FRI: Acceleration of the FRI protocol on the GPU for ZkSNARK applications. In *International Conference on Selected Areas in Cryptography*, pages 315–338. Springer, 2025.

[JLK+25] Jihoon Jang, Myeonghoon Lee, Donggeun Kwon, Seokhie Hong, Suhri Kim, and Sangjin Lee. Efficient polynomial multiplication for HQC on ARM Cortex-M4. *Cryptology ePrint Archive*, 2025.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S Han. Novel polynomial basis and its application to Reed-Solomon erasure codes. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 316–325. IEEE, 2014.

[LCK+18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast Fourier transform. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, pages 263–270, 2018.

[LJKH25] Myeonghoon Lee, Jihoon Jang, Suhri Kim, and Seokhie Hong. Improved Frobenius FFT for code-based cryptography on Cortex-M4. *IEEE Internet of Things Journal*, 2025.

[LSZH22] Wai-Kong Lee, Hwajeong Seo, Zhenfei Zhang, and Seong Oun Hwang. TensorCrypto: High throughput acceleration of lattice-based cryptography using tensor core on GPU. *IEEE Access*, 10:20616–20632, 2022.

[LWGP17] Wai-Kong Lee, Xian-Fu Wong, Bok-Min Goi, and Raphael C-W Phan. CUDA-SSL: SSL/TLS accelerated by GPU. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–6. IEEE, 2017.

[LZG+26] Tao Lu, Jipeng Zhang, Yanpei Guo, Xuanming Liu, Wenjie Qu, Zonghui Wang, Wenzhi Chen, and Jiaheng Zhang. TensorZKP: Repurposing GPU tensor cores for high-performance zero-knowledge proofs. *Cryptology ePrint Archive*, 2026.

[NVI25a] NVIDIA Corporation. CUDA C++ Programming Guide, release 13.1. NVIDIA Documentation, 2025.

[NVI25b] NVIDIA Corporation. Parallel Thread Execution ISA, Version 9.1. NVIDIA Documentation, 2025.

[OBS21] Tatsuki Ono, Song Bian, and Takashi Sato. Automatic parallelism tuning for module learning with errors based post-quantum key exchanges on GPUs. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[PDB+26] Nishant Pandey, Sanjay Deshpande, Dixit Dutt Bohra, Debapriya Basu Roy, Dip Sankar Banerjee, and Jakub Szefer. Performance analysis of parameterizable HQC hardware architecture. Cryptology ePrint Archive, Paper 2026/592, 2026.

[RLC+25a] Antonio Ras, Antoine Loiseau, Mikael Carmona, Simon Pontié, Guénaël Renault, Benjamin Smith, and Emanuele Valea. Optimizing HQC using Frobenius additive FFT on a RISC-V-based system-on-chip. In *2025 28th Euromicro Conference on Digital System Design (DSD)*, pages 608–615. IEEE, 2025.

[RLC+25b] Antonio Ras, Antoine Loiseau, Mikaël Carmona, Simon Pontié, Guénaël Renault, Benjamin Smith, and Emanuele Valea. PHOENIX: Crypto-agile hardware sharing for ML-KEM and HQC. *Cryptology ePrint Archive*, 2025.

[SER11] Davide Schipani, Michele Elia, and Joachim Rosenthal. On the decoding complexity of cyclic codes up to the BCH bound. In *2011 IEEE International Symposium on Information Theory Proceedings*, pages 835–839. IEEE, 2011.

[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.

[WDX+25] Wen Wu, Jiankuo Dong, Zhen Xu, Zhenjiang Dong, Dung Duong, Fu Xiao, and Jingqiang Lin. Symphony of speeds: Harmonizing classic McEliece cryptography with GPU innovation. *IEEE Transactions on Information Forensics and Security*, 2025.

[XGZW25] Dingyan Xu, Yiwen Gao, Yongbin Zhou, and Jian Weng. Hybrid-grained GPU implementations for the classic McEliece. In *2025 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 1194–1200. IEEE, 2025.

[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.

[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 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 1829–1843, 2025.