Optimized Implementation of Warp-Cooperative GPU HCTR2-ARIA Wide-Block Encryption

Siwoo Eum[1⋆], Minho Song[1], Seung-Won Lee[1], Hagyeong Kim[1], and Hwajeong Seo[1]

Hansung University, Seoul, South Korea
{shuraatum, smino0906, dkajdfhd1, kimhaha4420, hwajeong84}@gmail.com

Abstract. HCTR2 is a *wide-block* encryption mode that encrypts one fixed-size message as a single unit, so that flipping a single plaintext bit re-randomizes the whole ciphertext. Its main use is disk encryption, where the message is a disk sector. We instantiate it with ARIA, the Korean national block-cipher standard, and implement it on an NVIDIA RTX 4080 GPU. With many independent messages, assigning one thread per message keeps the device occupied. At low queue depth, however, most of the GPU sits idle, and encrypting one 16 KiB message is several times *slower* than a single CPU core. The reason is that one HCTR2 message is internally sequential, so it cannot simply be split across threads. We present a *fine-grained* kernel in which one 32-thread warp cooperates on a single message. The kernel exploits the linearity of POLYVAL. It splits the long hash computation into contiguous per-thread pieces, has each thread compute a partial result, and combines the partials with warp shuffles to obtain exactly the same value as the sequential computation. This brings the latency of one message down to *parity* with a single CPU core (ARIA-256: 222 μs on the GPU vs. 209 μs on the CPU). Moreover, the number of threads per message and of messages per block acts as a knob that trades latency for throughput, spanning 222 μs/30 GB/s to 1774 μs/49 GB/s. With as few as four messages in flight the GPU already beats the CPU. We validate correctness for all three key sizes.

Keywords: HCTR2 · ARIA · Wide-block encryption · Tweakable cipher · GPU · CUDA · Bank conflict · POLYVAL · Storage encryption.

1 Introduction

Disk and file encryption must be *length-preserving*. A storage device keeps data in fixed-size sectors and has no room to make the ciphertext larger for a nonce or an authentication tag. Each such fixed-size unit, which we call a *message*, should also behave as one wide block, meaning that changing any single plaintext bit re-randomizes the whole ciphertext. The de-facto standard XTS-AES [3] does not do this. It is *narrow-block*, encrypting each 16-byte piece on its own, so an

⋆ Corresponding authors: Siwoo Eum (shuraatum@gmail.com) and Hwajeong Seo (hwajeong84@gmail.com).

attacker can reorder the pieces inside a message or confine a one-bit change to a single piece. *Wide-block* modes, also known as tweakable enciphering and recently as *accordion* [12] modes, instead realize a strong pseudorandom permutation over the whole message. HCTR2 [2] is a modern, efficient example. It has a Hash–Encrypt–Counter–Hash structure that uses one block-cipher call, a counter-mode keystream, and two passes of the POLYVAL [4] hash. It improves on the earlier HCTR [16], has a security proof, and ships in the Linux kernel.

We build HCTR2 on **ARIA** [6,7], the 128-bit block cipher that is a Korean national standard (KS X 1213, ISO/IEC 18033-3). ARIA is a substitution–permutation network with 12/14/16 rounds for 128/192/256-bit keys. This pairing gives a standards-compliant wide-block mode for settings that require a national cipher, yet no fast implementation of it has been reported.

Encrypting many messages is easy to parallelize because the messages are independent, which suits a GPU's high *throughput*. The hard case is low *latency* on only a few messages, such as a single random disk read or a queue with just a few messages in flight. The common GPU mapping, *one thread per message*, is ideal for the first case but the worst for the second. With only a few messages, a few threads out of thousands of GPU cores do all the work, and a single 16 KiB message ends up several times slower than one CPU core. Making it worse, one HCTR2 message is internally sequential. Two POLYVAL hash passes run in order, and the counter keystream between them depends on a value S produced by the first pass, so a message cannot simply be split across threads. Splitting it makes the threads communicate. This paper targets exactly this low-queue-depth *latency gap*.

At the implementation level, both halves of HCTR2-ARIA read *tables at positions that depend on the secret data*. ARIA does data-dependent S-box lookups, and a fast POLYVAL multiplies using a table indexed by message bytes. GPU memory handles this pattern badly. Constant memory is efficient for warp-uniform broadcasts but serializes divergent addresses. A single shared-memory copy of a table causes *bank conflicts* when threads in a warp read different entries that land in the same memory bank. Table placement is therefore a first-order design decision.

Related work. GPU symmetric cryptography has focused almost entirely on AES, with shared-memory T-table designs [10,5] and bitslicing [13], and later work covering more ciphers, including Korean ones [1,8]. It is a known principle that higher occupancy is not always faster [15]. Shared-memory bank conflicts have also been addressed for AES. Tezcan [14] kept T-tables in shared memory while removing conflicts, reaching 878.6 Gbps for AES-128 on an RTX 2070 Super, and showed that conflict removal is the key lever for GPU T-table kernels. We extend this bank-conflict-free technique to two different non-AES data-dependent tables (the ARIA S-box and the POLYVAL multiply table) as the foundation of our kernel. Our focus, however, is not throughput but closing the single-message *latency* gap by having a warp cooperate on one message. Ours is also, to our knowledge, the first GPU realization of HCTR2, and the first over a non-AES cipher.

Contributions.

- **A warp-cooperative fine-grained kernel.** A group of TPS threads cooperates on one message. Its core is a *parallel hash reduction* that uses a linearity property of POLYVAL. The length-b hash sequence is split into contiguous per-thread pieces whose partial sums d_t are recombined as $D = \bigoplus_t d_t \, (h^{\mathrm{CHUNK}})^{\mathsf{TPS}-1-t}$ with in-warp shuffles, which reproduces the sequential result exactly. The counter keystream is split the same way, so each thread hashes the block it just produced without re-reading it.
- **A latency–throughput design space with a fair CPU comparison.** Threads per message (TPS) and messages per block (SPB) trade latency for throughput (Section 5). Fine-graining lowers the latency of one 16 KiB message from several times slower than a well-tuned CPU core to *parity* with it (ARIA-256: 222 vs. 209 μs), and the GPU overtakes the CPU with just four messages in flight. The baseline is a single-core reference compiled with `-O3 -march=native`, chosen conservatively so it does not favor the GPU.
- **Bank-conflict-free shared tables (the foundation).** We replicate *both* the ARIA S-box table and the POLYVAL multiply table so that thread ℓ always reads bank ℓ, which drives shared-table conflicts to zero. We also use a half-size POLYVAL table built from two 4-bit sub-tables using the linearity of multiplication. Together these keep the kernel's data-dependent lookups cheap.

2 Background

ARIA. ARIA [6,7] is a 128-bit block cipher with 128/192/256-bit keys and $N_r = 12/14/16$ rounds. Each round XORs in a round key, applies a substitution layer, and then a diffusion (bit-mixing) layer, with no diffusion in the last round. Two substitution types alternate from round to round, built from two S-boxes S_1, S_2 and their inverses. The diffusion layer is a fixed 16×16 binary matrix A. The standard software trick packs the four S-box outputs for one input byte into a single 32-bit word, giving one combined table of 256 entries. Each thread then picks the correct S-box byte with a byte permutation, and the matrix A is applied with word XORs and byte shuffles inside the word rather than more table lookups. So the only data-dependent memory access in the whole cipher is a single 1 KB table.

HCTR2. A wide-block mode behaves like a tweakable random shuffle [9]. The ciphertext has the same length as the plaintext, yet is indistinguishable from a random permutation even when the attacker can choose ciphertexts. HCTR2 [2] achieves this with one block-cipher call E, a counter keystream, and two passes of a $\mathrm{GF}(2^{128})$ hash. From the key it derives a POLYVAL hash key h (from $\mathsf{E}(0)$) and a block $L = \mathsf{E}(\mathtt{0x01} \,\|\, 0^{120})$. It splits a message into its first block M and remainder N under a tweak T (a per-message extra input, such as the message's location). Writing $\mathcal{H}_h(\cdot)$ for the POLYVAL hash over a shared tweak/length

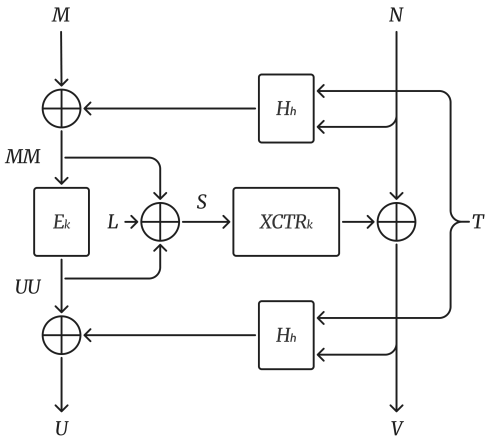


Fig. 1. HCTR2 encryption of one message.

prefix together with its argument, it computes

$$MM = M \oplus \mathcal{H}_h(N,T), \quad UU = \mathsf{E}(MM), \quad S = MM \oplus UU \oplus L, \tag{1}$$

$$V = N \oplus \mathrm{XCTR}_S, \quad U = UU \oplus \mathcal{H}_h(V,T), \quad C = U \,\|\, V, \tag{2}$$

where XCTR_S is the counter keystream whose block i equals $\mathsf{E}(S \oplus i)$. Two properties drive how we map this onto the GPU. First, the tweak and a length block form a *common prefix* of both hashes, so we can hash it once and then split the hash state. Second, for a message of b blocks the work is one ECB call (one block-cipher call), $b-1$ counter-mode calls, and $2(b-1)+2$ field multiplications. For our 16 KiB message this is 1 ECB, 1023 counter-mode calls, and about 2048 multiplications (Fig. 1).

POLYVAL and $\mathrm{GF}(2^{128})$. POLYVAL [4] is the hash used in AES-GCM-SIV, a close relative of GHASH [11]. With key h, the hash of blocks $X_1, \ldots, X_m$ is $\sum_j X_j\, h^{m-j+1}$, computed one block at a time as $\mathrm{acc} \leftarrow (\mathrm{acc} \oplus X_j) \cdot h$. Because h is fixed, the multiply by h can be turned into a per-key table. An 8-bit table $M_8[v] = v \cdot h$ replaces each multiply with about 16 lookups and XORs instead of about 128 bit-serial steps. How to place and replicate that table without bank conflicts is the subject of Section 4.

GPU memory and bank conflicts. A GPU runs threads in groups of 32, called *warps*, that execute together. *Shared memory* (fast on-chip memory) is split into 32 *banks*. The 32-bit word at byte address a lives in bank $\lfloor a/4 \rfloor \bmod 32$, and a warp can serve one word per bank at a time. If two threads want different words

in the same bank, the access is serialized. This is a *bank conflict*, and it is exactly what a data-dependent table lookup tends to cause. *Constant memory* serves a warp-uniform broadcast in a single access, but serializes k different addresses into k separate ones. Global-memory accesses *coalesce* when consecutive threads touch consecutive addresses, combining into one transaction. Finally, high *occupancy* (many warps resident at once) hides waiting time but is not always faster. A kernel limited by computation or by shared-memory speed can reach its peak at only modest occupancy [15].

3 GPU Design

Warp-cooperative fine-grained mapping. HCTR2 exposes abundant parallelism *across* messages. Each message is a self-contained unit with its own tweak, so with many messages *one thread per message* fills the GPU and gives high throughput. At low queue depth, however, only a few threads run, most cores sit idle, and the latency of one message is high. Reducing that latency requires parallelism *within* a message, but an HCTR2 message has two *sequential* POLYVAL hash chains and a counter keystream tied to a value S between them, so it cannot simply be split.

We therefore have a *group* of TPS threads (a full warp, or a power-of-two fraction of one so that TPS divides 32) cooperate on one message, in three phases. *(A) Parallel hash 1.* Because POLYVAL is linear, the length-b hash sequence (the tweak block followed by the N blocks) is split into contiguous pieces of $\mathsf{CHUNK} = b/\mathsf{TPS}$ blocks, one per thread. Thread t hashes its piece into a partial sum d_t, where only the group's first thread starts from the shared prefix state. A *warp shuffle* then recombines the partials via (3), giving *exactly* the sequential result. *(B) Single ECB and S broadcast.* The group leader forms MM, UU, S and shuffles S to the other threads. *(C) Parallel XCTR + fused hash 2.* The counter blocks $\mathsf{E}(S \oplus i)$ are independent, so with the same split each thread makes and writes its own V_i and folds V_i into a second partial sum while it is still in registers, with no re-read. The second partials are recombined as in (A) to produce U. The only communication between threads is these cheap shuffles, and nothing is staged in shared memory.

This mapping has two knobs. A larger TPS splits the counter and hash work more finely, which lowers the latency of one message. A smaller TPS gives each group more work and raises throughput (Section 5). We also pack SPB messages (groups) into one thread block so they share the block's shared tables, which are staged once. Round keys, L, h, and the hashed prefix sit in constant memory, where these accesses are warp-uniform. The three key sizes differ only in N_r, which we make a compile-time parameter so the round loop unrolls fully.

ARIA device core. The device core follows Section 2. A 256-entry combined table packs the four S-box outputs per byte. One round indexes this table and uses the hardware byte-permute instruction (`__byte_perm`) to select and place the substitution byte for the current round type, then applies the diffusion A

entirely in registers. The final round substitutes and XORs the last round key to produce ciphertext. The only data-dependent memory access is the 1 KB table lookup, which is exactly what makes aggressive replication affordable.

POLYVAL backends. We implement three interchangeable multiply backends behind one block-update routine, each selected at compile time so it can be measured on its own in Section 5. They are BITSERIAL, the table-free $\mathrm{GF}(2^{128})$ multiply (about 128 steps), TABLE, an 8-bit table in constant memory, and TABLE_SH, the replicated conflict-free shared table of Section 4, which is the optimized default. The POLYVAL byte-reversal is folded into the load and store steps.

Parallel reduction and the algorithm. The recombination in phase (A) is the identity

$$D \;=\; \bigoplus_{t=0}^{\mathsf{TPS}-1} d_t \cdot \left(h^{\mathsf{CHUNK}}\right)^{\mathsf{TPS}-1-t}, \qquad \mathsf{CHUNK} = b/\mathsf{TPS}, \tag{3}$$

which equals the *sequential* POLYVAL accumulator over the whole sequence. Here d_t is thread t's partial sum over its contiguous piece (only thread 0 starts from the shared prefix), and h^{CHUNK} shifts the degree by one piece. The leader does the recombination by multiplying by h^{CHUNK} and XOR-ing in the d_t it reads by shuffle ($\mathsf{TPS}-1$ steps). The identity holds whenever TPS divides b. Our 16 KiB message has $b = 1024$, which is divisible by every $\mathsf{TPS} \in \{2, 4, 8, 16, 32\}$. Algorithm 1 states the procedure from one thread's view, and each data block is read once and written once.

4 Bank-Conflict-Free Tables and the Design Space

For the fine-grained kernel to be cheap, its data-dependent table lookups must be fast. We first summarize the foundation, bank-conflict-free shared tables, which is a technique also used by throughput-oriented mappings. We then treat this kernel's own design space: threads per message and messages per block.

The problem. In both tables, each thread of a group indexes with a different, data-dependent byte (an ARIA state byte, or a POLYVAL message byte). Two natural placements fail. *Constant memory* broadcasts one address per access, so different indices are serialized through the constant cache. A *single shared copy* of a 32-bit-per-entry table puts entry i in bank i mod 32, so any threads whose indices are equal modulo 32 conflict and must replay. With random bytes this happens on nearly every access, which serializes the lookup.

Bank replication. The fix is to give every thread its own private copy of each table column, so that any access lands in *that thread's* bank. We declare the ARIA table `__shared__ u32 tS[256][RF]` with the *replication factor* RF (how

Algorithm 1 HCTR2-ARIA encryption of one message (thread t of a TPS-thread group)

Require: message blocks $M \,\|\, N_1 \,\|\, \cdots \,\|\, N_{b-1}$, tweak T, shared tables, constants h, L, prefix state acc_0, piece size $\mathsf{CHUNK} = b/\mathsf{TPS}$, sequence $X_0{=}T$ and $X_j{=}N_j$ for $j{\geq}1$

Phase A: parallel hash 1

$a \leftarrow \mathrm{acc}_0$ if $t{=}0$, else 0
for $j = t \cdot \mathsf{CHUNK}$ **to** $(t{+}1) \cdot \mathsf{CHUNK} - 1$ **do**
  $a \leftarrow \textsc{PolyUpdate}(a, X_j)$
end for
$dg_1 \leftarrow \textsc{Emit}(\textsc{Recombine}(a))$ ▷ by (3), via shuffle

Phase B: one block-cipher call (leader only)

if $t = 0$ **then**
  $MM \leftarrow M \oplus dg_1$
  $UU \leftarrow \textsc{Aria}_{N_r}(MM)$
  $S \leftarrow MM \oplus UU \oplus L$
end if
broadcast S from the leader by shuffle

Phase C: parallel counter mode and hash 2

$a \leftarrow \mathrm{acc}_0$ if $t{=}0$, else 0
for $j = t \cdot \mathsf{CHUNK}$ **to** $(t{+}1) \cdot \mathsf{CHUNK} - 1$ **do**
  if $j = 0$ **then**
    $X'_j \leftarrow T$ ▷ tweak, only hashed
  else
    $ks \leftarrow \textsc{Aria}_{N_r}(S \oplus j)$
    $V_j \leftarrow N_j \oplus ks$, then **store** V_j and set $X'_j \leftarrow V_j$
  end if
  $a \leftarrow \textsc{PolyUpdate}(a, X'_j)$
end for
$dg_2 \leftarrow \textsc{Emit}(\textsc{Recombine}(a))$
if $t = 0$ **then**
  store $U \leftarrow UU \oplus dg_2$
end if

many copies we keep) as the innermost dimension, and thread ℓ reads column $c = \ell \bmod \mathsf{RF}$. Element `tS[i][c]` sits at word $i \cdot \mathsf{RF} + c$, hence in bank

$$\big(i \cdot \mathsf{RF} + c\big) \bmod 32. \tag{4}$$

Setting $\mathsf{RF} = 32$ makes (4) equal $(32i + c) \bmod 32 = c = \ell$, so *every thread ℓ reads bank ℓ for every index i*. A warp then touches 32 different banks in one conflict-free access (Fig. 2). The price is 32× storage, so the 1 KB table becomes 32 KB. The POLYVAL table, declared `[nibble][word][RF]` with RF innermost, becomes conflict-free in the same way, since element $[n][w][c]$ is in bank $((4n + w) \cdot \mathsf{RF} + c) \bmod 32 = \ell$. Staging the tables (copying each row into all RF columns) is a one-time cost paid before the encrypt loop.

A half-size POLYVAL table from linearity. Replicating the 8-bit table M_8 (256× 16 bytes) by 32 would need 128 KB. We avoid this using the fact that multiplying

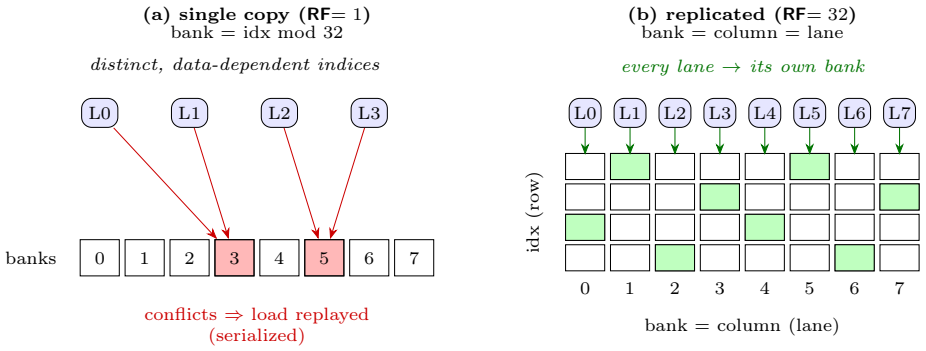


Fig. 2. Bank replication eliminates shared-memory conflicts (8 banks shown; the design uses 32).

by the fixed key h is linear (multiplying a sum equals summing the products). Since a byte splits as $v = (\mathrm{hi} \ll 4) \oplus \mathrm{lo}$, we can write

$$v \cdot h = M_4^a[\mathrm{hi}] \oplus M_4^b[\mathrm{lo}], \quad M_4^a[k] = (k \ll 4) \cdot h, \; M_4^b[k] = k \cdot h, \tag{5}$$

using two 16-entry tables, one for each 4-bit half of the byte. Replicated by $\mathsf{RF} = 32$ they take $2 \times 16 \times 4 \times 32 \times 4 = 16\,\mathrm{KB}$, which is half of a replicated 8-bit table and a quarter of the naive 128 KB. Together with the 32 KB ARIA table this comes to **48 KB of shared memory per block**, which is what sets occupancy.

Design space: threads per message TPS and messages per block SPB. Once the data-dependent lookups are cheap, performance is set by the two knobs of Section 3. First, *threads per message* TPS trades latency for throughput. A larger TPS splits a message's counter and hash work more finely and cuts the latency of one message, but each thread's piece gets shorter and the relative cost of the recombination (3) grows, which lowers per-group efficiency. A smaller TPS gives each group more work and raises throughput at the cost of latency. So TPS sweeps a frontier between latency and throughput.

Second, *messages per block* SPB sets the staging cost of the 48 KB shared tables and the occupancy. The 48 KB is a fixed per-block cost paid once, so packing SPB messages (groups) into a block spreads it over SPB messages and raises the number of resident warps. On the Ada SM two 48 KB blocks (96 KB) fit within the 100 KB capacity (with the shared-memory carveout, the split between L1 cache and shared memory, set to its maximum), so 2 blocks live on each SM, and raising SPB increases the warps those 2 blocks carry. At $\mathsf{SPB} = 1$ (one thread block per message) the 48 KB is re-staged for every message, which is wasteful. Grouping a few messages per block removes that waste.

Every global load and store is one 128-bit `uint4` access through the read-only path (`__ldg`). Section 5 sweeps this design space (TPS, SPB) in both latency and throughput and compares against a well-tuned single CPU core.

Table 1. Latency of one 16 KiB message (μs, lower is better).

Approach	ARIA-128	ARIA-192	ARIA-256
One thread per message (naive)	—	—	∼2310
CPU one core (`-O3 -march=native`)	185	203	209
GPU fine-grained (TPS=32)	**207**	**217**	**222**
Ratio (CPU/GPU time)	0.90×	0.94×	0.94×

5 Evaluation

Setup. We measure on an NVIDIA RTX 4080 (AD103, Ada, `sm_89`, 76 SMs, ≈716.8 GB/s DRAM peak), CUDA 12.0, driver 535.309.01, gcc 13.3, Ubuntu 24.04, with GPU flags `-arch=sm_89 -O3`. Shared tables use RF = 32 (conflict-free) with the maximum shared-memory carveout. We report two metrics. *Latency* is the time to encrypt a single 16 KiB message, in μs per message. For the GPU we run the kernel many times in a row and divide the device time by the number of runs (this is within 1% of the wall-clock time including launch overhead), and we take the minimum over 300 repetitions. *Throughput* is plaintext bytes over device time for 65536 messages (1 GiB), averaged over 20 iterations. The *CPU baseline*, chosen to be *fair*, is a single core running the reference HCTR2-ARIA compiled with `-O3 -march=native -funroll-loops` (T-table ARIA and table-assisted POLYVAL), timed by wall clock. It is not AES-NI-class, because ARIA has no x86 hardware instruction, and a hand-vectorized CPU (CLMUL POLYVAL, AVX2 ARIA) would be faster. In other words this baseline is conservative and does not favor the GPU.

Correctness. All checks pass. We run device-core ARIA known-answer tests for all three key sizes, check the GF(2^{128}) multiply (bit-serial and table) over 2^{20} inputs, and confirm that the HCTR2 output is byte-identical to an independent reference. The fine-grained kernel is checked for *every* TPS $\in \{2, 4, 8, 16, 32\}$, several values of SPB, and all three key sizes, including a message count that is not a multiple of SPB. The CPU baseline output also matches the reference.

Single-message latency: parity with a CPU core. Table 1 is the headline. The common *one thread per message* mapping takes about 2310 μs for a single 16 KiB message at ARIA-256, which is 11× slower than a CPU core (209 μs), because the GPU sits nearly idle. The warp-cooperative fine-grained kernel (TPS=32) cuts this to 222 μs, reaching *parity* with one CPU core (it uses 0.90–0.94× the CPU time across the three key sizes). It thus erases the single-message latency gap, from an order of magnitude down to parity.

Design space: TPS trades latency for throughput. Table 2 and Fig. 3 sweep TPS at a fixed 256-thread block (SPB = 256/TPS) for ARIA-256. Latency at queue depth 1 is measured on one message, and throughput on 65536 messages. Increasing TPS steadily drops the latency of one message from 1774 to 222 μs.

Table 2. The TPS design space (ARIA-256).

TPS	SPB	latency (μs)	throughput (GB/s)
2	128	1774	**49.0**
4	64	879	50.5
8	32	469	50.6
16	16	283	43.3
32	8	**222**	29.7

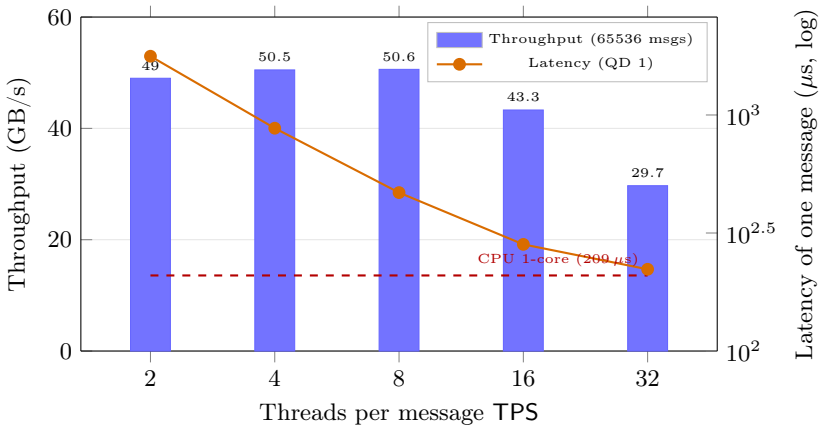


Fig. 3. The latency–throughput tradeoff as threads per message (TPS) varies (ARIA-256).

Throughput is highest for TPS≤8 (about 50 GB/s, and at TPS=2 it reaches 49 GB/s, close to a throughput-oriented mapping) and falls as TPS grows, because of lower per-group efficiency and recombination cost. So TPS is a knob along a frontier between latency and throughput. TPS=32 is best for latency (222 μs), TPS≤8 is best for throughput (about 50 GB/s), and TPS=16 is the balanced middle (283 μs, 43 GB/s).

Messages per block: sharing the table-loading cost. Table 3 increases SPB at TPS=32. With SPB=1 (one thread block per message) the 48 KB shared tables are re-loaded for every message, which caps throughput at 10.8 GB/s with only 2 resident warps. Grouping messages spreads the loading cost and raises occupancy, reaching 30.0 GB/s at SPB=8 (a 2.8× gain, 2 to 16 warps per SM). Beyond that, at SPB=16, the gain saturates.

Latency vs concurrency: the crossover. So far we fixed a single message (queue depth 1). Fig. 4 varies the number of messages processed at once and measures the time per message (TPS=32, SPB=8). The single CPU core's per-message time is flat at 209 μs, because it processes messages one after another. The GPU's

Table 3. Throughput vs messages per block SPB (ARIA-256, TPS=32).

SPB	block threads	warps/SM	GB/s
1	32	2	10.8
2	64	4	21.2
4	128	8	28.2
8	256	16	**30.0**
16	512	16	28.7

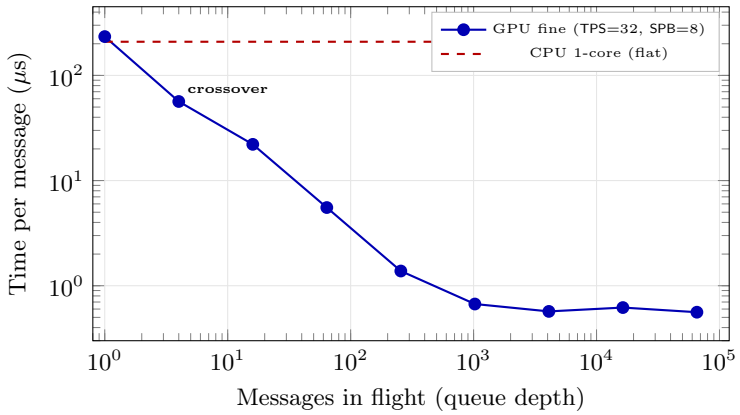


Fig. 4. Time per message as more messages run at once (ARIA-256, log–log). The GPU overtakes the CPU between 1 and 4 messages in flight.

per-message time falls steeply as the device fills up. At queue depth 1 the GPU is 234 μs, close to the CPU, but with just *four* messages in flight it is 56 μs, which is 3.7× faster than the CPU. It reaches 5.5 μs at 64 messages (38×) and saturates near 0.56 μs per message (about 30 GB/s). The crossover lies between queue depths 1 and 4. The value of fine-graining is exactly in this low-queue-depth regime. It matches or beats the CPU on a single message, and slides straight into GPU-class throughput as soon as a little concurrency appears.

Discussion. Fine-graining brings the latency of one message from 11× slower than a naive mapping to parity with one CPU core (222 μs), and it overtakes the CPU with as few as four messages in flight. TPS and SPB trade latency for throughput, so the same kernel spans 222 μs at 30 GB/s (latency-oriented) to about 880 μs at 50 GB/s (throughput-oriented). At queue depth 1 the GPU is at parity rather than a clear *win* over the CPU, and the main gain appears once concurrency is available. A faster CPU version (CLMUL POLYVAL, AVX2 ARIA) would put the queue-depth-1 CPU further ahead, which only reinforces the crossover-focused conclusion. We evaluated encryption on a single GPU. The design supports all three key sizes, and decryption is symmetric.

6 Conclusion

We presented a GPU implementation of HCTR2 wide-block encryption over the Korean standard ARIA, to our knowledge the first, focused on the *latency of one message at low queue depth*. The common one-thread-per-message mapping is ideal for bulk throughput, but with only a few messages it leaves the GPU idle and runs several times slower than a single CPU core. We close that gap with a fine-grained kernel in which a group of threads *cooperate* on one message. Its core is a parallel hash reduction (3) that uses the linearity of POLYVAL. Each thread hashes a piece into a partial sum, and a warp shuffle combines the partials to reproduce the sequential result exactly, with the counter keystream split the same way so each thread hashes the block it just produced. To make these data-dependent lookups cheap, we replicate *both* the ARIA S-box and POLYVAL multiply tables so that every thread reads its own bank ((4) guarantees exactly zero conflicts), and we shrink the replicated POLYVAL table with a 4-bit split.

The result is that fine-graining brings the latency of one 16 KiB message from 11× slower than a naive mapping to *parity* with a well-tuned CPU core (ARIA-256: 222 vs. 209 μs). Threads per message (TPS) and messages per block (SPB) then trade latency for throughput, so one kernel spans a range from latency-oriented (222 μs) to throughput-oriented (about 50 GB/s), and the GPU overtakes the CPU with as few as four messages in flight. At queue depth 1 the result is parity rather than a clear win against the CPU baseline; the larger gain comes from the crossover once concurrency appears. This approach should carry over to other wide-block modes and to national or lightweight ciphers built around a similar POLYVAL-style sequential hash. A faster CPU baseline (CLMUL/AVX), an overlap pipeline, and multi-GPU scaling are natural next steps.

References

1. An, S., Seo, S.C.: Highly efficient implementation of block ciphers on graphic processing units for massively large data. Applied Sciences **10**(11), 3711 (2020). https://doi.org/10.3390/app10113711
2. Crowley, P., Huckleberry, N., Biggers, E.: Length-preserving encryption with HCTR2. Cryptology ePrint Archive, Paper 2021/1441 (2021), https://eprint.iacr.org/2021/1441
3. Dworkin, M.: Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices. NIST Special Publication 800-38E (2010). https://doi.org/10.6028/NIST.SP.800-38E
4. Gueron, S., Langley, A., Lindell, Y.: AES-GCM-SIV: Nonce misuse-resistant authenticated encryption. RFC 8452, Internet Engineering Task Force (IETF) (2019). https://doi.org/10.17487/RFC8452
5. Iwai, K., Nishikawa, N., Kurokawa, T.: Acceleration of AES encryption on CUDA GPU. International Journal of Networking and Computing **2**(1), 131–145 (2012). https://doi.org/10.15803/ijnc.2.1_131
6. Kwon, D., Kim, J., Park, S., Sung, S.H., Sohn, Y., Song, J.H., Yeom, Y., Yoon, E.J., Lee, S., Lee, J., Chee, S., Han, D., Hong, J.: New block cipher: ARIA.

In: Information Security and Cryptology – ICISC 2003. Lecture Notes in Computer Science, vol. 2971, pp. 432–445. Springer (2004). https://doi.org/10.1007/978-3-540-24691-6_32
7. Lee, J.H., Lee, J., Kim, J., Kwon, D., Kim, C.: A description of the ARIA encryption algorithm. RFC 5794, Internet Engineering Task Force (IETF) (2010). https://doi.org/10.17487/RFC5794
8. Lee, W.K., Goi, B.M., Phan, R.C.W.: Terabit encryption in a second: Performance evaluation of block ciphers in GPU with Kepler, Maxwell, and Pascal architectures. Concurrency and Computation: Practice and Experience **31**(11), e5048 (2019). https://doi.org/10.1002/cpe.5048
9. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. In: Advances in Cryptology – CRYPTO 2002. Lecture Notes in Computer Science, vol. 2442, pp. 31–46. Springer (2002). https://doi.org/10.1007/3-540-45708-9_3
10. Manavski, S.A.: CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In: IEEE International Conference on Signal Processing and Communications (ICSPC). pp. 65–68. IEEE (2007). https://doi.org/10.1109/ICSPC.2007.4728256
11. McGrew, D.A., Viega, J.: The security and performance of the Galois/Counter Mode (GCM) of operation. In: Progress in Cryptology – INDOCRYPT 2004. Lecture Notes in Computer Science, vol. 3348, pp. 343–355. Springer (2004). https://doi.org/10.1007/978-3-540-30556-9_27
12. National Institute of Standards and Technology: Proposal of requirements for an accordion mode: Discussion draft. NIST, Workshop on Block Cipher Modes of Operation (2024), https://csrc.nist.gov/projects/block-cipher-techniques
13. Nishikawa, N., Amano, H., Iwai, K.: Implementation of bitsliced AES encryption on CUDA-enabled GPU. In: Network and System Security (NSS 2017). Lecture Notes in Computer Science, vol. 10394, pp. 273–287. Springer (2017). https://doi.org/10.1007/978-3-319-64701-2_20
14. Tezcan, C.: Optimization of Advanced Encryption Standard on graphics processing units. IEEE Access **9**, 67315–67326 (2021). https://doi.org/10.1109/ACCESS.2021.3077551
15. Volkov, V.: Better performance at lower occupancy. In: GPU Technology Conference (GTC) (2010)
16. Wang, P., Feng, D., Wu, W.: HCTR: A variable-input-length enciphering mode. In: Information Security and Cryptology (CISC 2005). Lecture Notes in Computer Science, vol. 3822, pp. 175–188. Springer (2005). https://doi.org/10.1007/11599548_15