IRON Engine GPU Kernel Optimization: Alignment Fix and Hybrid Parallelism for Token-Processing Throughput

Isaac Oravec
Independent Researcher
*

January 31, 2026

Abstract

I document GPU kernel optimizations for the IRON engine, a token-processing system with $N = 4096$ nodes. Initial deployment suffered from ∼1–5 MB/s throughput and alignment-related crashes. Two root causes: (1) XOR accumulators were `uint16_t` but used with 32-bit `atomicXor` (misalignment and data corruption); (2) a 2D kernel did one thread per (token, node) pair, yielding ∼23M atomics to global memory. I fix (1) by switching to `uint32_t` throughout, and (2) by replacing the 2D kernel with a *Hybrid* kernel: one block per node, 256 threads, strided token processing, warp/shared-memory reduction, one write per node. Throughput rises to ∼280 MB/s (Hybrid) versus ∼108 MB/s (node-centric) and ∼5 MB/s (broken 2D). This note is confined to GPU kernel optimization; algorithm and quantization are in a companion paper. **Keywords:** CUDA, GPU kernel optimization, memory alignment, atomic contention, parallel reduction, token processing, IRON engine.

1 Introduction

The IRON engine processes tokenized input on the GPU: each of $N = 4096$ nodes maintains an XOR accumulator and a potential value, updated from a stream of tokens. The computation is implemented in CUDA; a C++ server exposes HTTP endpoints and invokes the kernels on demand. Achieving high GPU throughput is critical so that downstream stages are not bottlenecked by the engine.

This note is confined to *GPU kernel optimization*. I describe the kernel-level changes that restored and then increased throughput from ∼1–5 MB/s to ∼280 MB/s: an alignment/type fix for XOR accumulators, and a switch from a 2D atomic-heavy kernel to a Hybrid (block-per-node, reduction-based) kernel. I do not change the high-level algorithm or any quantization scheme; I only change how the work is mapped to GPU threads and memory. Resonant-lattice and algorithm aspects are treated in a separate paper.

2 Problem: Alignment and Contention

2.1 XOR accumulator type and alignment

The XOR accumulator array was declared as `uint16_t*` (2 bytes per node). In the 2D kernel, updates were performed with:

```
atomicXor((unsigned int*)&state->
    xor_accumulators[node_idx], val);
```

Casting a `uint16_t*` to `unsigned int*` and calling `atomicXor` causes:

- **Misalignment:** `uint16_t` arrays can be allocated at 2-byte boundaries; 32-bit atomics require 4-byte alignment. Misaligned access can crash or trap.
- **Data corruption:** A 32-bit write overwrites two consecutive 16-bit elements, so `xor_accumulators[i]` and `xor_accumulators[i+1]` are modified together.
- **Performance:** The GPU may serialize or emulate non-native transaction sizes, reducing effective bandwidth.

*iron-292gbs repository.

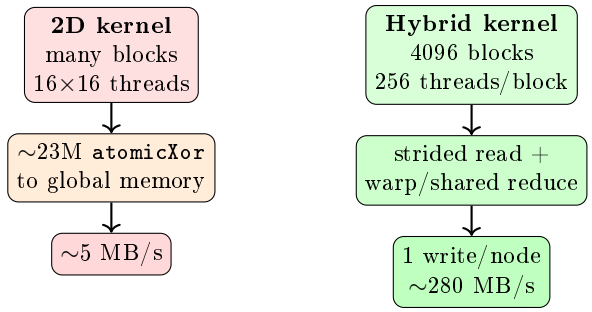

Figure 1: 2D kernel: many atomics to global memory (bottleneck). Hybrid: block-per-node, reduce in registers/shared, one write per node.

2.2 2D kernel and atomic contention

The original 2D kernel launched a grid of (token_blocks × node_blocks) with 16×16 threads per block. Each thread handled one (token, node) pair and updated that node's XOR with `atomicXor`. For ~22.5K tokens and 1024 nodes, this yields ~23M atomic operations to global memory. Even with correct 32-bit alignment, the memory controller cannot sustain that rate; throughput collapsed to ~5 MB/s. Figure 1 contrasts the 2D and Hybrid designs.

3 Method

3.1 Alignment fix: `uint32_t` everywhere

I changed the accumulator type and all related code to 32-bit:

- **State struct:** `uint16_t* xor_accumulators` → `uint32_t* xor_accumulators`.
- **Allocation:** `cudaMalloc` and `cudaMemset` use `sizeof(uint32_t)` per node.
- **2D kernel:** `atomicXor` on `xor_accumulators[node_idx]` with no cast; read-back as `uint32_t`.
- **Host readback:** In the Iron→Flesh bridge, `h_xor_acc` is `uint32_t*` with matching `cudaMemcpy` size.
- **Other kernels:** Node-centric, Hybrid, and tiled kernels use `uint32_t` for local XOR and write `(uint32_t)` values.

This removes undefined behavior and allows native 32-bit atomics where atomics are still used.

3.2 Hybrid kernel: block-per-node and reduction

I replaced the 2D kernel with a *Hybrid* kernel that avoids per-token atomics:

1. **Grid:** One block per node: `blocks = num_nodes` (4096), `threads = 256`.
2. **Per-thread work:** Each thread in a block processes a *strided* subset of tokens: `for (token_idx = threadIdx.x; token_idx < buffer_size; token_idx += 256)`. So each of 256 threads does ~buffer_size/256 iterations.
3. **Local accumulation:** Each thread holds `local_xor` and `local_potential` in registers.
4. **Reduction:** After the loop, threads reduce via warp shuffle (`__shfl_xor_sync`, `__shfl_down_sync`) and then shared memory to combine 256 partial results into one XOR and one potential per node.
5. **Write-back:** One write per node to `xor_accumulators[node_idx]` and `node_potentials[node_idx]` (no atomics).

Thus the token buffer is read in a coalesced, strided pattern; the only global writes are one per node per batch. This eliminates the 23M-atomic traffic and restores throughput.

3.3 Node-centric baseline

For comparison, the *node-centric* kernel uses 16 blocks × 256 threads (4096 threads total, one per node). Each thread loops over *all* tokens sequentially. No atomics, but only 4096-way parallelism, so the GPU is underutilized; measured throughput is ~108 MB/s. I keep this mode as an optional fallback.

4 Results

Throughput is measured as (data size in bytes) / (kernel time in seconds), with kernel time from CUDA events. Server runs on Windows with WDDM; buffer size 768K tokens (~3 MB). Figure 2 and Table 1 summarize results.

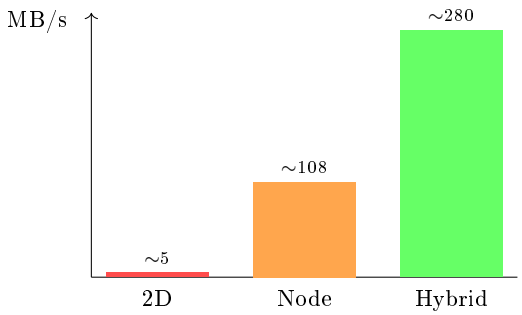

Figure 2: Throughput by kernel mode (MB/s). Hybrid ∼2.6× node-centric.

Table 1: Kernel throughput (MB/s) by mode.

Mode	Configuration	MB/s
2D (broken)	256×256 bl., 16×16, atomics	∼5
Node-centric	16 bl., 256 th., 1 th./node	∼108
Hybrid	4096 bl., 256 th., reduce/node	∼280

The alignment fix alone (2D with `uint32_t`) would remove crashes but not the atomic bottleneck. Switching to Hybrid removes contention and increases parallelism (256 threads per node), yielding $\sim$2.6$\times$ over node-centric.

4.1 Tradeoffs

- **Hybrid vs node-centric:** Hybrid uses 4096 blocks and reads the token buffer once per block; node-centric uses 16 blocks and one read per thread. Hybrid achieves higher throughput but more total memory traffic. For buffers larger than L2, Hybrid can approach cache thrashing; node-centric is lighter on bandwidth.
- **Tiled variants (Mode 4/5):** I tested shared-memory tiling (load token tiles, broadcast to threads). With 16 blocks (Mode 4) or 128 blocks $\times$ 32 threads (Mode 5), occupancy or tile overhead limited gains; Hybrid remained best for the tested workloads.
- **On-demand:** Kernels run only when the server receives feed/quantize requests; there is no 24/7 GPU load from these optimizations.
- **Beyond 280 MB/s:** The Hybrid kernel has 4096$\times$ read amplification (each block reads the full token buffer); $\sim$280 MB/s is achieved because the $\sim$3 MB buffer fits in L2 and is reused. To go higher: (1) *Tiled kernel* (Mode 4) reads the buffer only 16$\times$ (16 blocks), so it has 256$\times$ less global read than Hybrid; currently $\sim$120 MB/s, likely limited by low occupancy (16 blocks) and the serial inner loop over 256 tokens per tile—increasing effective blocks (e.g. 64 blocks with warp-level reduction per node) and vectorized tile loads (`uint4`) could raise throughput. (2) *Vectorized loads* in Hybrid: load 4 tokens per iteration with `uint4` to reduce instruction count. (3) *Buffer size*: keep batch in the 2–4 MB range so it stays L2-resident.

5 Conclusion

I restored and improved IRON engine GPU throughput by (1) fixing XOR accumulator type and alignment to `uint32_t`, and (2) replacing the 2D atomic-heavy kernel with a Hybrid block-per-node kernel using strided token processing and warp/shared-memory reduction. Throughput increases from $\sim$5 MB/s (broken 2D) and $\sim$108 MB/s (node-centric) to $\sim$280 MB/s (Hybrid). This note addresses only GPU kernel optimization; algorithm and quantization are covered elsewhere. Maximizing kernel speed provides headroom so that the engine is not the bottleneck.