IRON Engine Part II: Reduce-Then-Apply From 280 MB/s to 326 GB/s Token Throughput

Isaac Oravec
Independent Researcher
*

January 31, 2026

Abstract

In a previous note I reported IRON engine GPU kernel throughput of ∼108 MB/s (node-centric) and ∼280 MB/s (Hybrid block-per-node reduction). Here I show that the batch XOR and batch potential contribution are *identical for all nodes*: they depend only on the token stream. Computing them once and applying per node yields a *reduce-then-apply* design with $O(\text{tokens}) + O(\text{nodes})$ memory traffic instead of $O(\text{nodes} \times \text{tokens})$. Kernel 1 reduces the token buffer to two scalars (batch XOR, batch potential) via strided, coalesced reads and block-level reduction; Kernel 2 applies those scalars to every node with lattice quantization. Effective throughput reaches **326 GB/s** for a ∼3 MB buffer on RTX 2080 SUPER, achieving **66% of theoretical peak bandwidth** (496 GB/s), verified by Nsight Compute hardware profiling. This 1000× improvement over the Hybrid kernel comes from reading the token buffer exactly once with high parallelism and only 256 atomics globally. **Keywords:** CUDA, GPU, memory bandwidth, parallel reduction, reduce-then-apply, token processing, IRON engine, lattice quantization, roofline analysis.

1 Context and motivation

The IRON engine maintains $N = 4096$ nodes, each with an XOR accumulator and a potential value, updated from a stream of tokens. In Part I [1] I fixed alignment (`uint32_t` for XOR accumulators) and replaced the 2D atomic-heavy kernel with a *Hybrid* kernel: one block per node, 256 threads, strided token reads, warp/shared reduction, one write per node. Throughput increased from ∼5 MB/s (broken 2D) to ∼108 MB/s (node-centric) and ∼280 MB/s (Hybrid).

The Hybrid kernel still has each block read the *entire* token buffer. So total token memory traffic is $N \times \text{buffer\_size}$ reads. For 4096 nodes and a 3 MB buffer, that is ∼12 GB read. Throughput is limited by this amplification once the buffer exceeds L2. The question is: can we read the token buffer only *once* and still update all nodes correctly?

2 Key observation: batch contribution is node-invariant

For each batch of tokens, the update to node j uses:

- XOR: combine (positioned) tokens with XOR; then merge with node j's current XOR and quantize to the lattice.
- Potential: sum over tokens of a scalar contribution, scale by node j's activation, then add to node j's potential and quantize.

The *positioned* token stream (after the position-dependent hash) is the same for every node. So:

1. The XOR of all positioned tokens in the batch is a single value, say $\mathsf{batch\_xor}$. Every node will combine this same $\mathsf{batch\_xor}$ with its own current XOR (and then quantize).
2. The sum $\sum_t \big((\mathsf{token}_t \bmod \mathsf{vocab}) - \mathsf{half}\big) \cdot \mathsf{inv}$ is a single value, say $\mathsf{batch\_pot}$. Every node scales this by its activation and adds to its potential (then quantizes).

Thus the *batch contribution* is node-invariant. We can compute $\mathsf{batch\_xor}$ and $\mathsf{batch\_pot}$ once from the token buffer, then apply them to all nodes. Memory traffic becomes $O(\text{tokens})$ for the reduce phase and $O(\text{nodes})$ for the apply phase, instead of $O(\text{nodes} \times \text{tokens})$.

*Verified with Nsight Compute 2025.4.1, CUDA 13.1, driver 591.74 on RTX 2080 SUPER.

3 Design: two kernels

3.1 Kernel 1: `ob1_reduce_tokens`

Goal: Reduce the full token buffer to two scalars batch_xor, batch_pot.

Grid: 256 blocks × 256 threads (65 536 threads). Stride = 256 × 256.

Per-thread: For $i = \mathsf{idx}, \mathsf{idx} + \mathsf{stride}, \ldots$ with $i < \mathsf{total\_tokens}$, load `d_buffer[i]`, apply position hash position_hash(i), accumulate `local_xor ^= positioned_token` and `local_pot += ...` (potential formula). So each thread walks the buffer in a *strided*, coalesced-friendly pattern.

Block reduction: Write `local_xor` and `local_pot` to shared memory; tree-reduce in shared memory (XOR and sum) to one value per block.

Global write: One thread per block (tid = 0) does `atomicXor(d_batch_xor, s_xor[0])` and `atomicAdd(d_batch_pot, s_pot[0])`. So only **256 atomics** in total (one per block), not millions.

The token buffer is read *once*, in a coalesced pattern. No per-token atomics.

3.2 Kernel 2: `ob1_update_lattice`

Goal: Apply batch_xor and batch_pot to every node with the same lattice quantization semantics as the Hybrid kernel.

Grid: $\lceil N/256 \rceil$ blocks × 256 threads.

Per-thread: Each thread owns one node index. Read batch_xor and batch_pot from global (same two scalars for all). Read node's current XOR and activation. Compute new XOR = $\mathsf{old\_xor} \oplus \mathsf{batch\_xor}$, quantize to 12-point lattice, write back. Compute contribution = $\mathsf{activation} \times \mathsf{batch\_pot}$, quantize potential, add to `node_potentials[idx]`. All writes are one-per-node, no atomics (each thread owns one node).

4 Throughput and roofline analysis

4.1 Test environment

- **GPU:** NVIDIA GeForce RTX 2080 SUPER (Turing, SM 7.5)
- **Theoretical Peak BW:** 496.1 GB/s
- **Driver:** 591.74
- **CUDA:** 12.6 (WSL2) / 13.1 (Windows)
- **Buffer:** 768K tokens = 3,145,728 bytes

Table 1: Nsight Compute hardware-verified results.

Metric	Value
Mean kernel time	10.50 μs
Buffer size	3,145,728 bytes
Memory Throughput (Nsight)	**65.78%**
Compute Throughput (Nsight)	29.66%
Achieved Occupancy	68.3%
Theoretical peak	496.1 GB/s
Achieved BW	**326 GB/s**
Efficiency	**66%**

Table 2: Throughput progression (same engine, same algorithm).

Mode	Description	Throughput
2D (broken)	23M atomics	~5 MB/s
Node-centric (Part I)	16 bl., 1 th./node	~108 MB/s
Hybrid (Part I)	4096 bl., reduce/node	~280 MB/s
Reduce-then-apply (Part II)	1 read of buffer + apply	**326 GB/s**

4.2 Results

Throughput measured via Nsight Compute hardware counters (memory throughput percentage × theoretical peak bandwidth).

4.3 Why 66% efficiency is excellent

The kernel performs per-element:

- Position hash (2 muls, 2 adds, 1 shift)
- Integer modulo (expensive GPU operation, 20–80 cycle latency)
- Float arithmetic (sub, mul, accumulate)
- Shared memory reduction with synchronization

Pure memory-copy achieves 80–90% efficiency. Kernels with integer division typically achieve only 25–45%. This kernel achieves 66% *with* division—matching division-free sum reductions.

4.4 Comparison with previous designs

Kernel 2 is light (one read of two scalars per thread, plus per-node reads/writes); it does not dominate total time for typical batch sizes.

5 Implementation notes

- **Position hash:** Tokens are combined with a position-dependent hash, e.g. `position_hash =`

`(i*137) + (i>>4)*17`, before XOR and potential contribution, so the reduce kernel matches the semantics of the Hybrid kernel.

- **Lattice:** The 12-point lattice and quantization (scale to [7, 511], nearest lattice point) are unchanged; Part II only changes how the batch contribution is computed and applied.

- **Correctness:** Reduce-then-apply is mathematically equivalent to applying the same batch contribution to every node; the Hybrid kernel did that per-node reduction internally, here we do it once and then apply. Numerical results match when the same lattice and vocab are used.

6 Conclusion

By observing that the batch XOR and batch potential are the same for all nodes, I reduced token-buffer memory traffic from $O(N \times T)$ to $O(T) + O(N)$. The reduce-then-apply design (Kernel 1: reduce buffer to two scalars; Kernel 2: apply to all nodes) achieves **326 GB/s** effective throughput (66% of theoretical peak), hardware-verified with Nsight Compute, instead of the 280 MB/s of the Hybrid kernel—a **1000× improvement**—while preserving the same lattice semantics.

References

[1] I. Oravec, "IRON Engine GPU Kernel Optimization: Alignment Fix and Hybrid Parallelism for Token-Processing Throughput," (Part I). Throughput: node-centric ∼108 MB/s, Hybrid ∼280 MB/s.