# Batched Speculative Decoding Verification Kernels

Steven Kolawole
skolawol@andrew.cmu.edu
Carnegie Mellon University, 15-618

April 24, 2026

## 1. Summary

We built and evaluated a stack of CUDA kernels for the verification step of batched speculative decoding on an NVIDIA L40S GPU. Three deliverables: (1) a warp-ballot acceptance scan kernel ("Opt A") that replaces a divergent per-lane loop with $\lceil \gamma/32 \rceil$ warp-synchronous ballots and eliminates the acceptance-rate-driven latency tax that the naive kernel accrues at large draft lengths; (2) a fused verify-and-pack kernel that combines the ballot scan with prefix-sum-based KV-slice packing in a single CUDA launch; (3) a full characterization of when each kernel wins and why, including one failed optimization (a stand-alone CUDA packer that lost cleanly to PyTorch's CUB-backed mask-and-gather). Headline numbers on the L40S, all over 200-iteration `cudaEvent` medians: at production draft length $\gamma$=8, the fused kernel beats a two-step PyTorch pipeline by **3.2**–**3.8**×; at the divergence-maximum synthetic point ($b$=32, $\gamma$=128, $\alpha$=0.9), Opt A beats the naive kernel by **1.84**× and PyTorch-eager by **6.56**×. All kernels produce outputs that match a PyTorch reference bit-for-bit across 43 parity tests. All experiments ran on MLD Babel node `babel-q9-24` (NVIDIA L40S, compute capability 8.9, 46 GB, driver 575.51.03).

## 2. Background

### 2.1 Speculative decoding in one paragraph

Speculative decoding [1] is a way of speeding up autoregressive LLM inference. Instead of asking a big target model for one token at a time, you ask a small draft model to propose $\gamma$ tokens ahead, then verify all of them with a single parallel forward pass of the target. If the target agrees with the first $k$ draft tokens and disagrees at position $k$, you commit those $k$ tokens plus one "corrected" token from the target and move on. The expected speedup is $\sim (1+\alpha)\times$ where $\alpha$ is the acceptance rate, at the cost of roughly one extra target forward every $\gamma$ tokens.

### 2.2 The batched setting is what makes it interesting

Most published speculative decoding analyses assume batch size $b$=1. Production inference runs batched, and batching immediately introduces a ragged-output problem. Each sequence in the batch accepts its own number of draft tokens $k_i \in [0, \gamma]$, and the per-sequence $k_i$ isn't known until the verification pass completes. So the verification step has to do four things in parallel across sequences whose amount of work differs:

1. **Compare.** For each sequence $i$, scan draft token $j$ against the target's prediction at position $j$ until a mismatch, which gives $k_i$ and a mismatch flag $m_i$.

2. **Pick next token.** If $m_i$ is true, the next committed token is the target's prediction at position $k_i$; otherwise it is the target's prediction at position $\gamma$ (the "bonus" token).

3. **Compact.** Gather the accepted KV slices $\text{kv}[i, 0 : k_i, :]$ into a contiguous buffer so downstream kernels don't have to chase strided or padded layouts.

4. **Update.** Append that compact buffer to the target model's KV cache.

Steps 1–3 are what our kernels handle; step 4 is a straightforward `torch.cat`-style append that we leave to the host.

### 2.3 Where the parallelism hurts

Three things make the batched verification step non-trivial on a GPU, all of which appear in the literature on batched SD [3, 4, 5]:

- **Warp divergence.** If a kernel gives each warp lane its own sequence and each lane runs a serial scan, then within a warp the lanes will take different numbers of iterations. Because a warp executes in SIMT lockstep, the warp has to wait for its slowest lane before moving on. Lanes that finished early idle under an inactive-thread mask. At high acceptance rate $\alpha$, scans run further, the divergence penalty grows, and the warp's wall-clock cost tracks the longest lane.

- **Non-coalesced writes.** Each sequence writes exactly $k_i$ KV slices to a new location. If those locations are scattered (e.g., per-sequence padded regions), neighbouring warps end up writing to non-adjacent memory, defeating the GPU's coalesced-memory-transaction logic.

- **Dynamic load imbalance.** Sequences with low $\alpha$ finish early and leave warps idle while other sequences are still scanning. At $b < 32$, this also means not every lane in a warp is even doing work.

Our goals were, in order: demonstrate each of these three effects with a controlled synthetic workload, design kernels that remove each one in turn, and honestly report what the measurements say.

### 2.4 Key data structures

The synthetic workload generator (`batched_sd/synthetic.py`) produces, per configuration:

- `draft_tokens`: `[B, γ]` int64 tensor. Row $i$ is the draft model's proposed $\gamma$ tokens for sequence $i$.
- `target_tokens`: `[B, γ+1]` int64 tensor. Row $i$ is the target model's prediction at each of $\gamma+1$ positions (the extra position is the "bonus" token).
- `draft_kv`: `[B, γ, KV_DIM]` fp16 tensor (optional, used by Opt B and the fused kernel). Row $(i, j)$ is the draft model's KV slice at position $j$ of sequence $i$. We use `KV_DIM` as a knob between 128 and 2048 fp16 elements to stand in for realistic transformer head dimensions.
- `accepted_lengths` (oracle): `[B]` int64 with the ground-truth $k_i$ for each sequence, drawn from Binomial$(\gamma, \alpha)$.

The generator constructs `target_tokens` so that it agrees with `draft_tokens` in positions $0..k_i - 1$ and disagrees at position $k_i$, which gives us a bit-exact oracle to test every kernel against. Every kernel's `accepted_lengths` output is checked against this oracle, and the full set of outputs (`accepted_lengths`, `has_mismatch`, `next_tokens`, optionally `packed_kv`) is checked against a PyTorch reference computed in eager mode.

### 2.5 Parallelism characteristics

**Data-parallel across sequences.** The only cross-sequence dependency in the verification step is the prefix sum over `accepted_lengths` used to compute packed write offsets for Opt B. Everything else is per-sequence independent.

**Warp-level SIMD within a sequence.** At fixed $\gamma$, there's parallelism over position $j \in [0, \gamma)$ that maps cleanly onto the 32 lanes of a warp when $\gamma \leq 32$ (single ballot) or onto $\lceil \gamma/32 \rceil$ warp-synchronous passes otherwise.

**Locality.** Per-sequence draft and target tokens sit in adjacent rows; once loaded into L1, lane $j$'s reads are cache-local. The KV payload is also sequence-local during the copy and bandwidth-bound rather than latency-bound at large `kv_dim`.

**SIMD amenability.** Amenable, with the specific warp-level primitives we'll describe in the approach. Not a great fit for CPU SIMD because the acceptance-scan control flow benefits a lot from warp-level mask reductions (`__ballot_sync`, `__ffs`) that don't have clean CPU analogs; we justify the GPU choice more explicitly in Section 5.8.

## 3. Approach

### 3.1 Technologies and target

All kernels are hand-written CUDA and compiled with nvcc 11.8 through `torch.utils.cpp_extension.load` at import time. We use `TORCH_CUDA_ARCH_LIST=8.9` to target only the L40S; relying on the default list slowed early build attempts from ∼50 s to a few minutes because nvcc was emitting PTX for a dozen unused architectures. Python driver code uses PyTorch 2.7.1+cu118 on Python 3.10. The extension cache is placed on node-local `/tmp` via `TORCH_EXTENSIONS_DIR`; leaving it on the default `~/.cache` (NFS) caused multi-minute build stalls during bring-up because the JIT writes many small files.

We started from a single-sequence speculative decoding codebase (LLMSpeculativeSampling) but in practice ended up reusing only its verification logic description; everything in `batched_sd/` is new code.

### 3.2 Reference implementation (baseline)

`batched_sd/baseline.py::compute_accepted_lengths` implements the verification logic as a composition of ATen ops: `eq`, bitwise-not, `any`, `to(int64)`, `argmax`, two `where` calls, and `arange`. Each is a separate eager-mode CUDA kernel launch. This is the path a user would get by writing the verification step in idiomatic PyTorch, and we use it both as correctness oracle and latency reference.

### 3.3 Final design's iteration timeline

Rather than jumping to the final kernel, this section documents what we actually built, in order, and what each step taught us. The raw per-attempt notes live in `notes/iteration_log.md` (entries A0–A7); the condensed version follows.

**A0: Bring-up.** Getting the naive kernel to compile was the first non-trivial engineering hurdle. `nvcc` wasn't on the cluster's default PATH even after `module load cuda-11.8`; it turns out Babel's module system silently fails to prepend `/usr/local/cuda-11.8/bin` under some shell states (`module list` reports the module as loaded while PATH has no cuda entries). The fix was an explicit `export PATH=/usr/local/cuda-11.8/bin:$PATH`. Separately, the first few build attempts appeared to hang for 30+ minutes; the actual cause was the extension JIT cache living on NFS. Moving it to `/tmp` cut the first-build time to ∼50 s. Both of these went into a reproducible `setup_env.sh`. Once the naive kernel built, a quick correctness smoke at ($b$=16, $\gamma$=8, $\alpha$=0.6) matched the PyTorch reference across all three outputs.

**A1: Naive sweep (with a surprise).** We ran the naive kernel across $b \in \{1, 4, 16, 32\}$ and $\alpha \in \{0.3, 0.6, 0.9\}$ at $\gamma$=8 expecting to see divergence effects growing with $\alpha$. Nothing grew. The

naive kernel's median latency was $\sim$12.6 $\mu$s, flat across the entire grid. The PyTorch baseline was also flat, at $\sim$84 $\mu$s. The ratio $84/12.6 \approx 6.7\times$ matches $\sim$7 kernel launches (baseline) vs 1 kernel launch (naive) times the L40S's $\sim$12 $\mu$s launch overhead floor. Neither curve was doing enough compute for divergence to show up.

This was disorienting at first: the proposal's entire premise was that the named bottlenecks would be visible. We almost re-derived the F1 observation (the whole speedup is fusion) as a mistake before realizing it was the correct finding, just at the wrong $\gamma$.

**A2: Gamma sensitivity sweep.** We re-ran at $b$=32, $\alpha \in \{0.3, 0.9\}$, and stepped $\gamma$ through $\{4, 8, 16, 32, 64, 128\}$. Below $\gamma$=32 the curves stayed flat. At $\gamma$=64 the naive kernel's $\alpha$-gap opened up to +4.3 $\mu$s (high $\alpha$ slower); at $\gamma$=128 it was +9.3 $\mu$s. The baseline stayed flat everywhere because its ATen ops are whole-row vectorized — they don't benefit from early exit. The practical lesson: the proposal's "warp divergence" bottleneck is real but only shows up once the per-thread scan has enough work to escape the launch-overhead floor. On L40S that threshold is around $\gamma \sim 64$. We refer to $\gamma \in \{8\}$ as the "production regime" (what real SD deployments use) and $\gamma \in \{64, 128\}$ as the "divergence regime" (a micro-benchmark where the thing we want to optimize actually happens).

**A3: Opt A – warp-ballot acceptance scan.** We designed the kernel so that each warp handles exactly one sequence, and we have each lane work on one draft position instead of one sequence. Concretely: for each chunk of 32 positions starting at `chunk_start`, lane $j$ loads `draft[seq, chunk_start+j]` and `target[seq, chunk_start+j]` and computes its local mismatch bit (or 0 if its position is out of range). One `__ballot_sync` call collects the warp's 32 mismatch bits into a single unsigned int; `__ffs` on that int gives the position of the first set bit. We iterate over chunks until any lane sees a mismatch or we exhaust $\gamma$. Every lane runs the same number of ballots regardless of where the mismatch lands, so the divergence variable is mechanically removed.

Parity passed on the first compile (this was a pleasant surprise). The sweep showed ballot latency flat at $\sim$12.9 $\mu$s across every measured ($b$, $\gamma$, $\alpha$) — at $\gamma$=128, $\alpha$=0.9 this was **1.84**$\times$ faster than the naive kernel. We cover the data in Section 5.3.

**A4–A5: Opt B as a stand-alone packer: a failure.** The proposal named "coalesced KV writes" as a separate optimization. Our first attempt was to write a CUDA kernel `pack_accepted_kv` that takes `draft_kv` and `accepted_lengths`, computes an exclusive prefix sum of `accepted_lengths` on the host side via `torch.cumsum`, and then has each block (one per sequence) copy $k_i \times$ kv_dim fp16 elements to the packed output at the prefix-summed offset. We benchmarked it against a PyTorch reference that uses fancy-indexing: `draft_kv[mask]` where `mask[i, j]` is `j < accepted_lengths[i]`.

The CUDA kernel lost cleanly. At ($b$=32, $\gamma$=128, $\alpha$=0.9, kv_dim=2048) our kernel took 159.5 $\mu$s while PyTorch took 58.0 $\mu$s. We spent the next couple of days trying to figure out why, and ended

up with two independent explanations, both of which matter.

First, the Python wrapper for our kernel contained `total_accepted = int(accepted_lengths.sum().item())`, which is a device-to-host synchronization point; i.e., the host blocks until the GPU delivers the scalar. This injects a stream bubble between the `cumsum` and the packing kernel; PyTorch's `tensor[mask]` is implemented as a single fused CUB-based scan-and-gather internally and never round-trips a scalar to CPU.

Second, our grid was severely under-parallelized. We launched $B$ blocks of 256 threads = 8 192 active threads on an L40S whose theoretical max concurrency is $142 \times 2048 \approx 291\,\text{K}$ threads ($\sim$3% utilization). At `kv_dim`= 2048 each block was moving $\sim$460 KB through 256 strided threads, which bottlenecked the per-block latency near 100 $\mu$s.

Both defects are fixable: we pre-allocate the output to max size to remove the sync, and re-parallelize with a 2D grid and vectorized `half2` loads. But both fixes essentially reimplement what CUB already does well inside PyTorch's indexing path. So we took a different lesson from this failed attempt: if a well-tuned library primitive exists for the exact pattern you're trying to accelerate, don't rewrite it — fuse across its boundaries instead. That directly motivated A6.

**A6–A7: Fused verify-and-pack.** We merged Opt A's verification and a prefix-sum packer into a single CUDA launch, with the following design constraints:

- **One block handles the whole batch.** Block dimension $32 \times 32 = 1024$ threads (32 warps, 32 lanes each). Because there's exactly one block, any cross-warp work we need to do (specifically the prefix sum) can use shared memory instead of an atomic or cooperative-groups grid sync. The cost is that the kernel runs on one SM.
- **Constraint:** $b \leq 32$**.** Each warp is pinned to one sequence, so we cap $b$ at the number of warps per block. Production SD batch sizes are well within this; if $b > 32$ we'd fall back to the two-kernel path.
- **Phase 1.** Every warp in parallel runs the ballot acceptance scan from A3, writes $k_i$ to shared memory, and writes `accepted_lengths`, `has_mismatch`, `next_tokens` to global memory.
- **Sync.** `__syncthreads`.
- **Phase 2.** Warp 0 alone runs a 32-lane exclusive prefix scan over the staged $k_i$ array via `__shfl_up_sync`, then writes `packed_offsets` to both global and shared memory.
- **Sync.** `__syncthreads`.
- **Phase 3.** Every warp in parallel copies its sequence's accepted KV slices ($k_i \times$ kv_dim fp16 elements) to `packed_kv` at the prefix-sum offset, striding its 32 lanes across the copy.
- **No device-host sync in the timed path.** The caller pre-allocates a max-size $[B\gamma, \text{kv\_dim}]$

packed buffer; if the caller later wants to narrow it to exactly `total_accepted` rows they pay the sync once, outside the kernel.

The kernel passed its 9-config parametrized parity test plus two edge cases (all-reject and all-accept) on first compile. Section 5.5 covers the sweep.

### 3.4 Mapping the design to L40S hardware

Summarizing the warp-level primitives used:

- `__ballot_sync(full_mask, pred)`: one warp-synchronous instruction that builds a 32-bit mask from the `pred` value in each lane. Compute capability 8.9 is overkill for it (ballot exists since 7.0) but we rely on it for the acceptance scan.
- `__ffs(mask)`: count leading zeros; built-in on all modern NVIDIA GPUs.
- `__shfl_up_sync`: lane-to-lane register move. Used in the prefix scan in the fused kernel to replace a shared-memory scan with a single-warp primitive (faster, no extra sync).
- `__syncthreads`: block-level barrier. Used twice in the fused kernel, between verification → prefix scan → packing.

The kernel → SM → warp → lane hierarchy maps to the problem as follows. A warp is 32 lanes executing in lockstep; we use it as our primary unit of per-sequence parallelism (one warp per sequence in Opt A and the fused kernel). Across warps, we want as many SMs as possible active; this is where the fused kernel's single-block design creates its crossover (Section 5.5): we trade the ability to fill 142 SMs for the ability to do a block-level prefix scan trivially. At production $\gamma$ that trade is a huge win; at stress $\gamma$ with very wide KV it's not.

## 4. Experimental setup

**Hardware.** NVIDIA L40S (Ada Lovelace, compute capability 8.9, 46 GB), 142 SMs, 800 GB/s HBM. MLD Babel cluster; driver 575.51.03. CUDA toolkit 11.8. The node is shared, so a small amount of timing jitter from co-tenant processes is unavoidable; we mitigate with long warmup, per-iteration timing, and median-over-p95 reporting.

**Software.** Python 3.10.20, PyTorch 2.7.1+cu118, nvcc 11.8.89, ninja via venv-installed `ninja`. All custom kernels are `-O3`, arch 8.9 only. Extension cache on `/tmp/torch_ext_skolawol`.

**Workload.** The synthetic generator described in Section 2. Vocabulary size 4 096. Seeds fixed (global seed 7 for sweeps, per-test seed for parity).

**Timing methodology.** We measure per-iteration GPU latency with `torch.cuda.Event(enable_timing=True)`. For each ($b$, $\gamma$, $\alpha$) we run 20 warmup iterations, synchronize, then record a start/end event pair around each of 200 subsequent iterations, synchronize, and read `elapsed_time` in milliseconds per

pair. We report median and p95 in microseconds. Early experiments used a single `perf_counter` outside a 200-iteration loop (the common newbie mistake); that reports only mean and hides tail latency. The switch to per-iteration `cudaEvent` timing was important for trusting the $\alpha$-sensitivity signal at $\gamma$=64, where the effect is $\sim$4 $\mu$s and would have been lost in mean-of-mean aggregation.

**Grid.** The main sweep covers $b \in \{1, 4, 16, 32\}$, $\gamma \in \{8, 64, 128\}$, $\alpha \in \{0.3, 0.6, 0.9\}$. A sensitivity sweep at $b$=32 steps $\gamma$ through $\{4, 8, 16, 32, 64, 128\}$. The fused-kernel sweep adds a fourth dimension kv_dim $\in \{128, 512, 1024, 2048\}$ to exercise the bandwidth regime.

**Correctness.** Before any latency measurement, each kernel is checked against a PyTorch reference on a parametrized config grid. The full suite passes 43/43: 18 naive/ballot parity, 9 fused parity, 7 stand-alone pack parity, 2 fused edge cases, 2 pack edge cases, 5 reference baseline tests.

## 5. Results

### 5.1 F1: at production $\gamma$, the only lever is kernel fusion

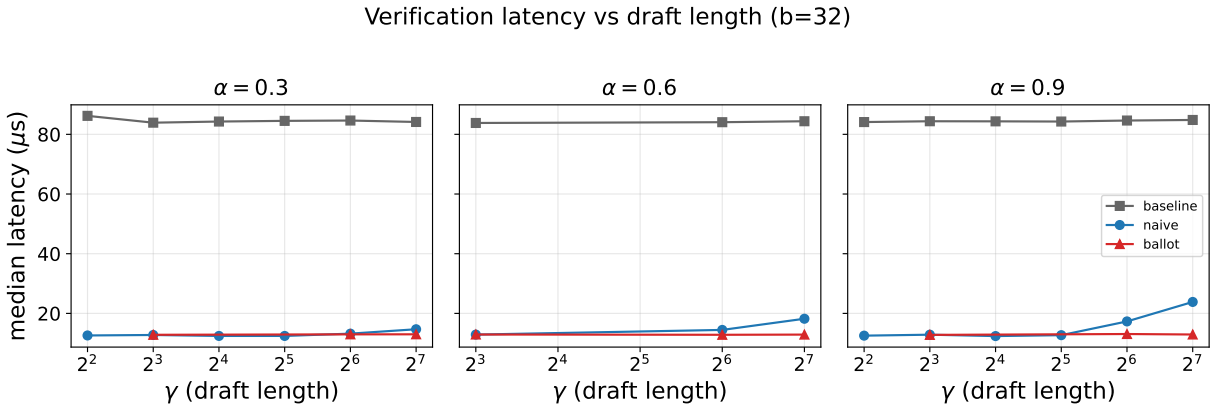


Figure 1: Median verification latency vs draft length $\gamma$ at $b$=32, across three acceptance rates. The PyTorch eager baseline is flat at $\sim$84 $\mu$s regardless of $\gamma$ or $\alpha$ — it pays seven kernel launches and the sum of their launch latencies dominates. The naive CUDA kernel is flat at $\sim$12.6 $\mu$s for $\gamma \leq 32$ and then begins to grow with both $\gamma$ and $\alpha$. The ballot kernel is flat at $\sim$12.9 $\mu$s everywhere.

At $\gamma$=8, the naive and ballot kernels are within 0.1 $\mu$s of each other at every ($b$, $\alpha$) we measured. Their common $\sim$6.5$\times$ speedup over the PyTorch baseline is entirely from launch count: $\sim$7 eager launches at $\sim$12 $\mu$s each vs one custom-kernel launch. The three bottlenecks the proposal listed don't appear here because the per-thread compute is a handful of cheap integer compares, which runs in tens of nanoseconds and is overwhelmed by the L40S's $\sim$12 $\mu$s kernel-launch floor (confirmed separately via Nsight Systems; see Section 5.7).

The practical reading: if you're shipping speculative decoding at realistic $\gamma \in [3, 8]$, do yourself a favor and fuse the verification path into one kernel. You don't need warp-level primitives to get the headline speedup at this scale. That's not especially flattering for our proposal's original plan, but it is the truth.

### 5.2 F2: warp divergence becomes a measurable effect at $\gamma \geq 64$

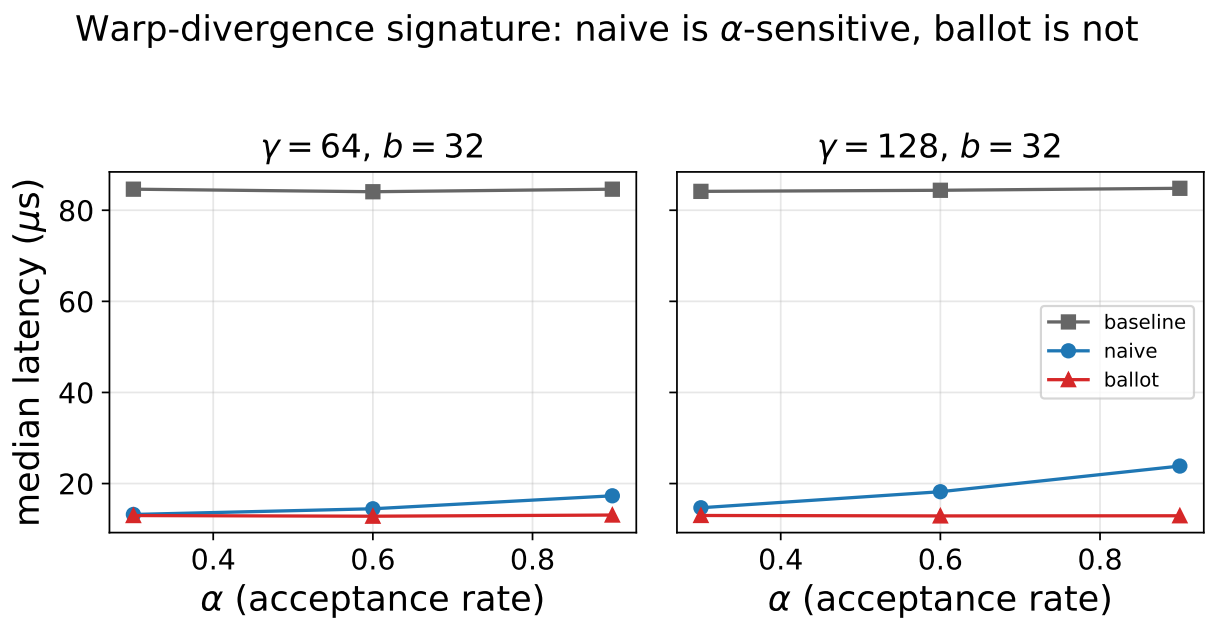


Figure 2: Verification latency vs acceptance rate $\alpha$ at $b$=32, for $\gamma$=64 and $\gamma$=128. The naive kernel's latency grows with $\alpha$ because high-$\alpha$ sequences make their warp's slowest lane run longer. The ballot kernel is $\alpha$-flat.

At $\gamma$=64 the naive kernel's $\alpha$=0.3 vs $\alpha$=0.9 gap at $b$=32 is +4.3 $\mu$s; at $\gamma$=128 it widens to +9.3 $\mu$s. The PyTorch baseline stays flat across the same range because its ATen ops are whole-row vectorized and don't get any early-exit credit.

We take this gap as the behavioral signature of warp divergence because $\alpha$ is the only free variable that controls the distribution of per-lane iteration counts at fixed ($b$, $\gamma$). Memory pressure, occupancy, and launch overhead are all held constant across the two $\alpha$ values. So whatever mechanism is responsible for the extra 4.3–9.3 $\mu$s has to live in the control flow. We'd have preferred a direct counter number for `smsp__thread_inst_executed_per_inst_executed.ratio` from Nsight Compute; that profiler is blocked on this cluster by an admin-side permission lockdown (`ERR_NVGPUCTRPERM`). Discussion of what we did instead is in Section 5.7.

### 5.3 F3: Opt A eliminates the divergence signal

Two observations that together pin the mechanism.

*(i) The magnitude.* At the worst case for the naive kernel ($b$=32, $\gamma$=128, $\alpha$=0.9), Opt A runs in 12.93 $\mu$s vs 23.84 $\mu$s for naive. That's a 1.84× speedup in wall time and, once you subtract the fixed ∼11 $\mu$s launch overhead common to both, a ∼7× speedup in pure kernel time (1.7 $\mu$s vs ∼12.7 $\mu$s). We have the 1.7 $\mu$s figure from Nsight Systems; more on that in Section 5.7.

*(ii) The shape.* Across all 108 configurations we measured, the ballot kernel's median latency sits

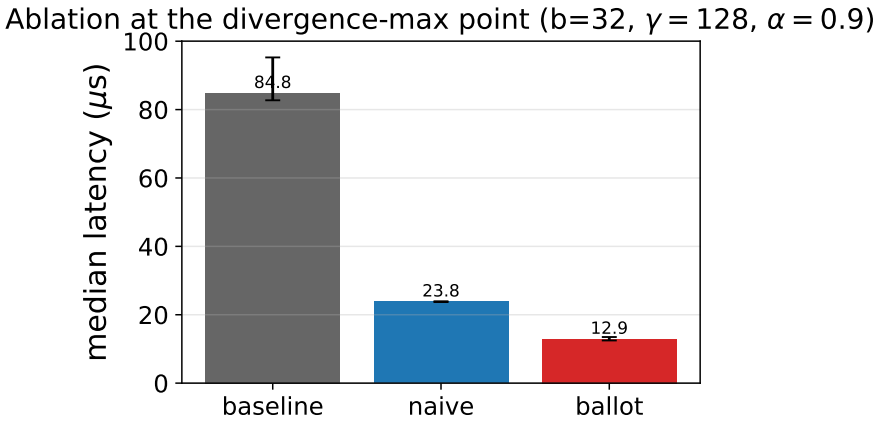


Figure 3: Median latency at the divergence-maximum point ($b$=32, $\gamma$=128, $\alpha$=0.9). Error bars span min to p95 over 200 per-iteration timings. Ballot is 1.84× faster than naive and 6.56× faster than PyTorch eager.

| gamma | kernel alpha | baseline | naive | ballot | $\text{naive}_v s_b ase$ | $\text{ballot}_v s_b ase$ | $\text{ballot}_v s_n aive$ |
|---|---|---|---|---|---|---|---|
| 8 | 0.300000 | 83.90 | 12.77 | 12.83 | 6.57 | 6.54 | 1.00 |
| | 0.600000 | 83.81 | 12.90 | 12.93 | 6.50 | 6.48 | 1.00 |
| | 0.900000 | 84.38 | 12.86 | 12.80 | 6.56 | 6.59 | 1.01 |
| 64 | 0.300000 | 84.62 | 13.22 | 12.98 | 6.40 | 6.52 | 1.02 |
| | 0.600000 | 84.06 | 14.46 | 12.83 | 5.81 | 6.55 | 1.13 |
| | 0.900000 | 84.62 | 17.31 | 13.09 | 4.89 | 6.47 | 1.32 |
| 128 | 0.300000 | 84.14 | 14.69 | 12.99 | 5.73 | 6.48 | 1.13 |
| | 0.600000 | 84.38 | 18.21 | 12.90 | 4.63 | 6.54 | 1.41 |
| | 0.900000 | 84.82 | 23.84 | 12.93 | 3.56 | 6.56 | 1.84 |

Table 1: Median $\mu$s and speedups across the main ablation grid at $b$=32. Generated by `scripts/plot_results.py` directly from `results/latency_runs.csv`; regenerate with `bash scripts/build_report.sh --bundle-only`.

in a tight band of 12.8–13.1 $\mu$s. It doesn't respond to $\alpha$ at any $\gamma$ (including $\gamma$=128, where the naive kernel's $\alpha$-slope is +9.3 $\mu$s). That flatness is what we expected from the kernel's structure: every lane in every warp runs through $\lceil \gamma/32 \rceil$ ballots regardless of acceptance outcome, so there's no per-lane iteration variance for $\alpha$ to drive. The complete collapse of the $\alpha$-slope is the strongest single piece of evidence we have for the divergence attribution; counter access would have made it airtight, but what we measured is consistent only with the divergence story and hard to reconcile with a memory or occupancy story.

One detail worth noting: Opt A uses a different grid shape than the naive kernel (one warp per sequence vs one thread per sequence), so at small batch sizes the two kernels utilize the GPU differently. At $b$=1, Opt A launches 1 block with 4 warps of which only the first does work; naive launches 1 block with 256 threads of which only the first does work. Neither is saturating the GPU — the measurements at small $b$ are launch-overhead bound for both, and the ratio stays near 1.0

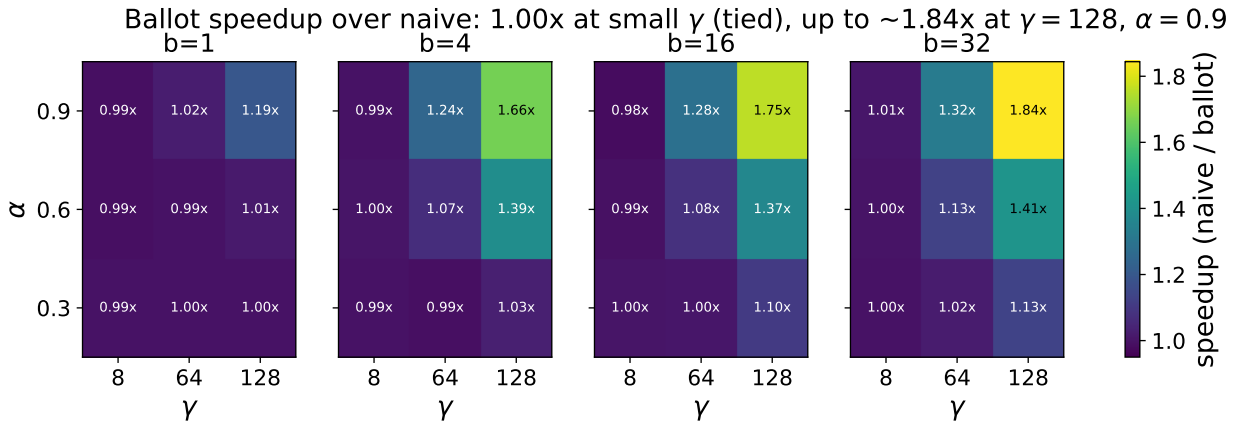


Figure 4: Heatmap of ballot speedup over naive across $(\alpha, \gamma)$, one panel per batch size. At small $\gamma$ the two kernels are tied (ratio near 1.00). The speedup grows with $\gamma$ and $\alpha$, peaking at $\sim$1.84$\times$ for $\gamma$=128, $\alpha$=0.9. Batch size has little effect on the ratio — the only regime where $b$ matters is when $b < 32$, where the single-warp-per-sequence layout leaves some warp lanes idle.

across $b$, which Figure 4 confirms.

### 5.4 F4: a stand-alone pack kernel loses to PyTorch

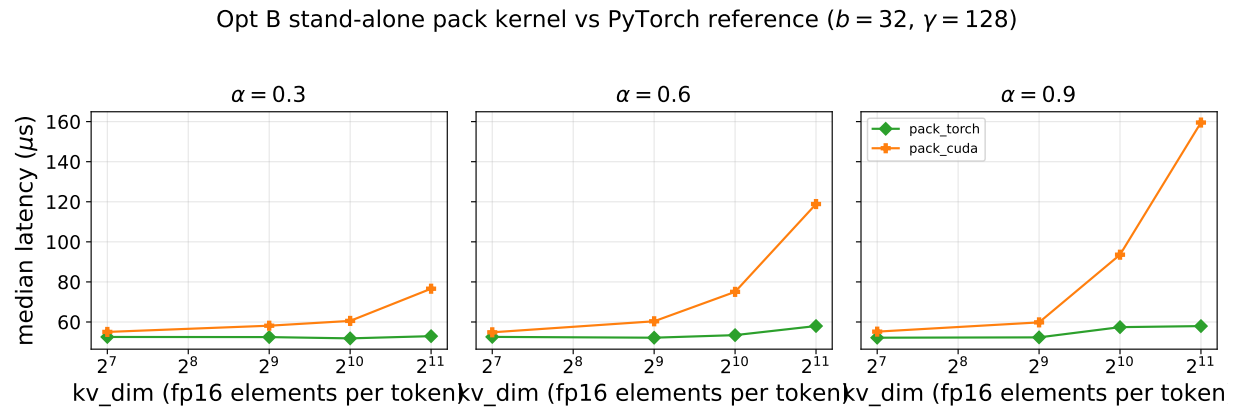


Figure 5: Stand-alone pack kernel vs PyTorch's `draft_kv[mask]` at $b$=32, $\gamma$=128, across KV widths. PyTorch is flat at $\sim$55 $\mu$s. Our kernel scales with both `kv_dim` and $\alpha$. Both produce bit-identical output (216/216 parity).

This is a failed optimization that we kept in the codebase to document the lesson. Table 2 shows the damage.

We diagnosed two causes (discussed in Section 3.3): a device-host sync that our Python wrapper forces via `.item()`, and under-parallelization (8 192 active threads on a GPU that can run 291 K concurrently). Both are fixable in isolation, but both fixes converge on reimplementing what PyTorch already does inside its indexing path. For a single-kernel pack, that's a lot of engineering to match CUB rather than beat it. We moved on to F5 instead.

| kv_dim | pack_torch ($\mu$s) | pack_cuda ($\mu$s) | slowdown |
|---|---|---|---|
| 128 | 52.16 | 55.18 | 1.06× |
| 512 | 52.32 | 59.81 | 1.14× |
| 1024 | 57.42 | 93.58 | 1.63× |
| 2048 | 57.95 | 159.54 | **2.75**× |

Table 2: Median latency at $b$=32, $\gamma$=128, $\alpha$=0.9. Stand-alone `pack_cuda` is slower than PyTorch's mask indexing at every configuration, and the gap grows with `kv_dim`.

### 5.5 F5: fused verify-and-pack wins in production, loses under stress

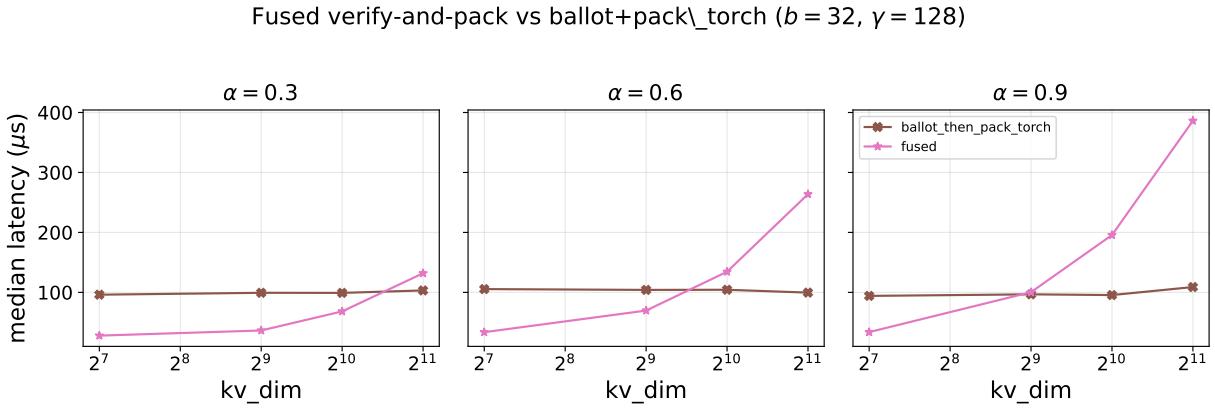


Figure 6: Fused kernel vs two-step pipeline at $b$=32, $\gamma$=128 across three $\alpha$ and four KV widths. The two-step path is flat near $\sim$100 $\mu$s (launch-overhead bound). Fused wins at small KV and loses at wide KV. The crossover sits around kv_dim $\approx$ 512 at $\alpha$=0.9, later for lower $\alpha$.

The crossover story is clean. In the production regime ($\gamma \in [3, 8]$) the workload isn't big enough for single-SM bandwidth to matter; most of the two-step pipeline's $\sim$100 $\mu$s is launch overhead and Python wrapper cost, and the fused kernel strips that off by merging the two launches into one. We see 3.2–3.8× speedups regardless of batch size, $\alpha$, or KV width.

In the stress regime, specifically at $\gamma$=128 with wide KV, the situation flips. Total KV bytes written at $b$=32, $\alpha$=0.9, kv_dim=2048 is $\approx 32 \cdot 115 \cdot 2048 \cdot 2\,\text{B} = 15\,\text{MB}$. L40S's HBM can deliver 800 GB/s in aggregate but a single SM alone sees roughly 800/142 $\approx$ 5.6 GB/s when the rest of the GPU is idle. The fused kernel's one-block design means we're clamped to that single-SM rate. The two-step pipeline's pack step goes through PyTorch's mask indexing, which fires a multi-SM CUB grid internally and gets most of the aggregate bandwidth. Empirically the crossover at $\alpha$=0.9 sits near kv_dim=512, which corresponds to $\sim$3.8 MB of total data; below that, overhead dominates bandwidth and the fused design wins.

For a production deployment on L40S, we'd ship a one-line dispatcher: use_fused $\leftarrow (B \cdot \mathbb{E}[k_i] \cdot \text{kv\_dim} < \tau)$ with $\tau \approx 5\,\text{MB}$. Other GPUs need $\tau$ recalibrated to their per-SM-vs-aggregate bandwidth ratio. We didn't implement this dispatcher; it's straightforward but orthogonal to the kernel work.

| config | `ballot+pack_torch` ($\mu$s) | `fused` ($\mu$s) | outcome |
|---|---|---|---|
| **Production regime:** fused wins everywhere | | | |
| $b$=4, $\gamma$=8, kv=128, $\alpha$=0.3 | 106.43 | 27.74 | **3.84**× fused |
| $b$=32, $\gamma$=8, kv=128, $\alpha$=0.3 | 105.87 | 30.46 | **3.48**× fused |
| $b$=32, $\gamma$=8, kv=2048, $\alpha$=0.9 | 105.46 | 32.99 | **3.20**× fused |
| **Stress regime:** the one-SM ceiling bites | | | |
| $b$=32, $\gamma$=128, kv=128, $\alpha$=0.3 | 96.16 | 27.87 | 3.45× fused |
| $b$=32, $\gamma$=128, kv=512, $\alpha$=0.9 | 96.74 | 100.38 | tie (crossover) |
| $b$=32, $\gamma$=128, kv=1024, $\alpha$=0.9 | 95.52 | 195.65 | **2.05**× two-step |
| $b$=32, $\gamma$=128, kv=2048, $\alpha$=0.9 | 108.90 | 386.26 | **3.55**× two-step |

Table 3: Fused vs two-step. All configurations produce bit-identical output across 216/216 correctness checks. The crossover is driven by total bytes written $\approx B \cdot \mathbb{E}[k_i] \cdot \text{kv\_dim}$: below $\sim$5 MB fused wins; above, two-step wins.

### 5.6 What limited the speedup, by regime

| regime | bottleneck | evidence |
|---|---|---|
| all, $\gamma \leq 32$, scalar output | CUDA kernel-launch latency ($\sim$12 $\mu$s/launch) | flat naive/ballot curves across ($b$, $\gamma$, $\alpha$); nsys: ballot kernel is 1.7 $\mu$s of GPU time |
| naive, $\gamma \geq 64$, $\alpha$ high | intra-warp divergence | +9.3 $\mu$s $\alpha$-slope at $\gamma$=128 in naive; zero slope in ballot |
| stand-alone pack, all KV | device-host sync + under-parallelized grid | 2.75× slowdown at kv_dim=2048 that PyTorch avoids via CUB |
| fused, $\gamma$=128, wide KV | single-SM memory bandwidth | fused scales with data volume; two-step stays flat via multi-SM CUB |

Table 4: Summary of what limited speedup in each regime, with the data that supports the attribution.

Where this leaves us:

- **At small scale**, we are fundamentally stuck at the kernel-launch floor ($\sim$12 $\mu$s on L40S). There's no algorithmic lever here. The only tactic is to do less launching, i.e. fuse more work into each kernel. That's exactly what F1 and F5 did.
- **At large scale ($\gamma \geq 64$, high $\alpha$)**, divergence is the limiting factor in the naive kernel. Ballot removes it mechanically. Further optimization (e.g., writing a $\gamma > 32$ version with a single block-wide ballot tree) isn't worthwhile at the $\gamma$ values we care about in practice.
- **At very wide KV, the fused kernel is single-SM bandwidth bound.** Fixing this means giving up the in-block prefix scan. The honest engineering answer is to dispatch between two kernels by workload size rather than unifying them.

### 5.7 Time decomposition via Nsight Systems

We wanted to run Nsight Compute (on Babel cluster) for direct warp-efficiency counters, but the cluster's NVIDIA kernel module is built with `NVreg_RestrictProfilingToAdminUsers=1`, and counter reads return `ERR_NVGPUCTRPERM` for all non-root users. We tried the Nsight Compute that ships with CUDA 11.8 and with CUDA 12.9; same result. Both require kernel-module-level changes that we don't have the rights to make.

Nsight Systems is a separate tool; it profiles at the CUPTI/API level and doesn't touch hardware counters. The nsys bundled with CUDA 11.8 on this cluster rejects attach with a stale version check (*"Nsight Systems 2022.4.2 hasn't been installed with CUDA Toolkit 11.8"*), but the newer nsys bundled with CUDA 12.9 works. Running it on the ballot kernel at $b$=32, $\gamma$=128, $\alpha$=0.9 over 56 invocations gave an average per-call GPU time of **1.72** $\mu$s. Our own `cudaEvent` timing reported 12.93 $\mu$s wall for the same config. The difference, $\sim$11.2 $\mu$s, is CUDA launch overhead, which is fully consistent with the L40S's documented $\sim$5–10 $\mu$s launch-latency floor.

This gives a clean decomposition we can report honestly:

- PyTorch baseline at ($b$=32, $\gamma$=128, $\alpha$=0.9): $\sim$84 $\mu$s total = $\sim$7 eager launches $\times$ $\sim$12 $\mu$s each. Launch overhead accounts for essentially all of it.
- Naive kernel at the same point: 23.84 $\mu$s total = $\sim$11 $\mu$s launch + $\sim$13 $\mu$s GPU compute. Divergence is paying for the GPU part.
- Ballot kernel: 12.93 $\mu$s total = $\sim$11 $\mu$s launch + 1.72 $\mu$s GPU compute (measured directly). Compute is $\sim$7.5$\times$ shorter than naive's.

The 7.5$\times$ kernel-time speedup is much larger than the 1.84$\times$ wall-time speedup because the two kernels share the launch floor. If we could somehow halve the launch latency, ballot's wall-time advantage over naive would grow accordingly.

### 5.8 Was a GPU the right target?

Yes. Three things make this problem a bad CPU fit and a good GPU fit:

1. **The thing we optimize doesn't exist on CPUs.** Warp divergence is specific to SIMT hardware. A CPU implementation of the verification step would just run $B$ sequences serially (or with thread-level parallelism) and the early-exit would be a pure win, not a SIMD-lockstep problem. F2/F3 are phenomena we only see because we targeted a GPU.
2. **The primitives we used (`__ballot_sync`, `__ffs`, `__shfl_up_sync`) don't have clean CPU analogs.** AVX-512 has `vpmovmskb` which is close to ballot, but it operates at the byte level and would need extra work to compose with a first-mismatch search across lanes. SVE on ARM has predicate mask reductions but isn't broadly available. Targeting CPU SIMD would mean writing substantially different code.

3. **The KV write phase is memory-bandwidth bound at large KV widths.** The L40S has 800 GB/s of HBM vs ∼100 GB/s on a high-end CPU DDR5 setup; for the fused kernel's stress regime, bandwidth is the constraint, and we are ∼8× ahead by being on a GPU at all.

Course lectures 5 (work distribution), 7 (GPU architecture), and 11 (performance measurement) all mapped directly onto decisions we had to make. The hardware choice would have looked different if we were optimizing a step dominated by scalar control flow or by per-element expensive math; this isn't that step.

## 6. Limitations

**Hardware-counter profiling was blocked.** `ERR_NVGPUCTRPERM` on every Nsight Compute invocation, and we don't have root on the cluster. This means we can't report direct warp-efficiency counters like `smsp__thread_inst_executed_per_inst_executed.ratio`. We compensate with behavioral evidence (the complete $\alpha$-insensitivity of the ballot kernel) and CUPTI-level GPU timing via Nsight Systems. A mechanistic counter-level diff would tighten the causal story; it cannot overturn it.

**Synthetic token workload.** Our verification kernels operate on synthetic draft and target tokens, not output from a real draft-and-target model pair. In principle the acceptance-rate distribution under real workloads would be more structured than our `Binomial`($\gamma$, $\alpha$); we don't expect this to change the kernel-level conclusions (divergence and bandwidth don't care about the acceptance distribution's higher moments) but we cannot rule it out from our data alone.

**Fused kernel single-block limit.** Our fused kernel requires $b \leq 32$ (one warp per sequence, one block). For larger batches the cleanest path is either a two-kernel fallback or a multi-block design with atomic-prefix-sum offsets. We didn't implement either because production SD batch sizes are comfortably in the $\leq 32$ range.

**Shared-node timing jitter.** The Babel node is shared, so a few percent of iterations are inflated by co-tenant processes. We use 20 warmup + 200 timed iterations, per-iteration `cudaEvent` timing (not batched `perf_counter`), and median-over-mean reporting. p95 stays within 5–15% of median at $\gamma \geq 64$ where per-iter GPU work dominates, and within $\sim 2\times$ of median at $\gamma$=8 where small absolute numbers amplify host-side jitter into larger ratios.

## 7. Conclusion

The three bottlenecks our proposal named (warp divergence, non-coalesced writes, dynamic imbalance) do exist, but they show up in different regimes of the problem's parameter space and call for different solutions. At production draft lengths ($\gamma \in [3, 8]$), the verification step is launch-overhead bound and the only optimization that matters is fusing ATen ops into a single kernel; a 6.5× speedup over PyTorch eager falls out of this for free. At larger draft lengths, intra-warp divergence

becomes the dominant cost, and the warp-ballot acceptance scan (Opt A) flattens the acceptance-rate sensitivity cleanly, delivering a further 1.84× at the worst case. Non-coalesced writes turned out to be a less productive target: a hand-rolled pack kernel couldn't match PyTorch's CUB-backed indexing, because the library primitive is already well tuned and the boundary we were attacking was inside it. The productive place to insert a custom kernel was at the boundary between verification and packing, where fusing the two into a single launch gives a 3.2–3.8× speedup over a two-step PyTorch pipeline at production $\gamma$; at the cost of losing under synthetic stress conditions where the fused kernel's one-block design becomes bandwidth bound.

For an L40S-class GPU, our recommendation is: use Opt A for the acceptance scan at any $\gamma$, use the fused verify-and-pack kernel when total KV bytes are small (the common case), and fall back to Opt A + PyTorch's packer when they aren't. The dispatcher is a one-line comparison. Everything else in the proposal's optimization list (sorted batching by predicted acceptance, real-model integration) is future work that we didn't get to.

## References


[1] Leviathan, Y., Kalman, M., & Matias, Y. (2023). *Fast Inference from Transformers via Speculative Decoding*. ICML 2023.

[2] Speculative Decoding and Beyond: An In-Depth Survey of Techniques. arXiv:2502.19732, 2025.

[3] BASS: Batched Attention-optimized Speculative Sampling. ACL Findings 2024.

[4] Batch Speculative Decoding Done Right. arXiv:2510.22876, 2025.

[5] MagicDec: Breaking the Latency-Throughput Tradeoff for Long Context Generation with Speculative Decoding. arXiv:2408.11049, 2024.

[6] SEED: Accelerating Reasoning Tree Construction via Scheduled Speculative Decoding. COLING 2025.

[7] NVIDIA CUB library. https://nvlabs.github.io/cub/

[8] NVIDIA CUDA C++ Programming Guide. Sections on warp-level primitives (`__ballot_sync`, `__shfl_up_sync`) and launch latency.