

Range-addressed KV cache blocks for the vLLM serving engine

The tagma backend behind the vLLM scheduler contract: per-request contiguous ranges replace the per-position block-table gather on the write path, measured at the software level on the SSCCS fork, with the GPU serve benchmark pending

Taeho Lee*
SSCCS Initiative
ssccs.org

August, 2026

Abstract

vLLM is the de facto LLM serving engine. Its scheduler interleaves prefill and decode steps across requests, and PagedAttention manages the KV cache memory that those steps consume: the cache is split into fixed-size blocks, and a per-sequence block table maps logical token positions to physical blocks. Every attention computation gathers K and V through that table, and the write path loads one table entry per token position. The indirection grows with sequence length and batch size.

This report documents the tagma KV cache backend on the SSCCS vLLM fork. The backend presents the same scheduler-facing KVCacheManager contract behind a new `--kv-cache-backend tagma` selector. A request holds its KV blocks as contiguous ranges, so the slot-mapping kernel derives the physical block from the range base plus the logical offset, with no per-position block-table load. Prefix caching, request references, and cache references live in the range allocator. The software level is implemented and measured on the host: the closed-form coordinate arithmetic, the allocator bookkeeping, and a data-structure-level model of both slot-mapping strategies. The host model measures parity between the two strategies; the GPU serve benchmark is the remaining measured question. The read path keeps the materialized block table in this version because the attention backends, including external ones with fixed ABIs, consume that tensor.

1 The block-table access pattern

vLLM's scheduler runs continuous batching: requests share the GPU in interleaved prefill and decode steps, and the KV cache manager hands each request its blocks for the step. The manager is the scheduler-facing contract, and the per-request block table is its data structure: logical token positions map to physical blocks through it.

The indirection carries a structural cost.

1.1 Block-table lookup per position

The slot-mapping kernel computes the physical position of every token by loading the request's block-table entry. On the write path this is one table load per token position, and the attention kernels repeat the same resolution on the read path. Each resolution is an extra memory access, and the count scales with sequence length, batch size, and the number of blocks each request touches.

*Founder & Architect; Corresponding author: lee@ssccs.org

1.2 Reported overhead figures

Third-party studies report a block-table preparation share of decode iteration latency and a kernel gap against specialized attention implementations. These figures are cited with their sources where possible and remain unverified in this work until the GPU serve benchmark measures them on the SSCCS fork. The structural cost, an extra lookup per position, does not depend on those figures.

1.3 Scaling degradation

The per-position lookup cost grows with context length. As context windows move toward 128K tokens and beyond, the cumulative indirection cost becomes a first-order consideration, which the GPU serve benchmark measures rather than assumes.

1.4 Existing optimizations stay inside the paradigm

Community efforts have compacted the table, for example by fusing table operations or flattening its layout. Those are optimizations of the table, and the structural cost of the indirection remains.

2 The Innovation: Range Arithmetic Replaces the Per-Position Gather

The tagma write path replaces the per-position block-table gather with range arithmetic. In the paged backend, the slot-mapping kernel computes the physical block for every token position by loading the request’s block-table entry:

```
physical_block = block_table[req][position // block_size]
```

In the tagma backend, the request’s KV blocks form contiguous ranges. The kernel loads the range descriptor once per position block and derives each physical block by arithmetic:

```
physical_block = range_start + (position // block_size - range_cum)
```

The load leaves the per-position path entirely: the paged kernel issues one table load per position, the tagma kernel issues the descriptor load per position block and arithmetic per position. Positions that fall outside every range are padded with the same pad id the paged kernel uses, so CUDA graph capture sees stable slot-mapping values across runs. This report measures that difference at the software level, and the serve benchmark measures it on-device.

The same structural pattern, an index-table lookup replaced by a closed form, was measured end to end in the CERN ROOT-Coord work on a production CMS dataset. Those numbers are the measured result of that workload; they do not project a vLLM speedup.

The structural parallel is the same index-table pattern in both workloads:

CERN ROOT TTree	vLLM Block Table
Branch, basket, and cache index traversal	Logical-to-physical block indirection
372,000 reads over roughly 14 hours	Per-block lookup in every attention computation
Coordinate arithmetic measured at 69.0x to 181.7x	Range arithmetic on the write path, GPU measurement pending

The vLLM column is the target of the serve benchmark, not a measured claim.

3 The Coordinate Approach

3.1 How coordinate addressing works for the KV cache

The coordinate space is defined by:

```
n(layer, block, token) = (layer * B + block) * T + token
offset = n * bytes_per_token
```

Where:

- `layer` = model layer index (0..L-1)
- `block` = KV cache block index (0..B-1)
- `token` = token offset within block (0..T-1)
- `bytes_per_token` = KV cache bytes per token

No hash, no block table, no index scan. The physical address is computed directly. This arithmetic is implemented in `csrc/tagma/kv_coord_map.h` on the SSCCS fork.

3.2 The allocator: contiguous ranges as the closed form

Assigning requests contiguous block ranges makes the physical block a closed form of the logical position. `csrc/tagma/kv_allocator.h` implements the contiguous-range allocator: aligned range allocation, a free list that is always coalesced, in-place growth for decode extension, prefix sharing by reference, and least-recently-used eviction of cache-only ranges. Request references and cache references are tracked independently, so the prefix cache can hold a range after every request releases it.

Two design decisions keep the write path cheap in practice. Decode growth extends the request's last range in place, and adjacent ranges merge on append, so the steady state is one range per request and the slot-mapping kernel's range scan is a single iteration per position block. The alignment contract applies to extensions as well: allocation counts must be multiples of the configured alignment, the hook the MLA storage block size uses.

Prefix caching in version 1 covers new-request hits, the same scope as the paged backend. The scheduler's hit lookup applies only to requests with no computed tokens, so decode-section hits are not part of the version 1 contract.

4 The Direction

The work tracks in milestones:

- M1: coordinate arithmetic and the contiguous-range allocator (C++ engine)
- M2: the `vllm.tagma` host extension (stable ABI 3.11, pure C++)
- M3: the `TagmaKVCacheManager` preserving the scheduler-facing contract, with prefix caching
- M4: the write-path slot-mapping kernel without a per-position block-table load
- M5: host benchmark, verification, and this report
- M6: GPU serve benchmark, paged against `tagma`, on ShareGPT at 16K and 128K context
- M7: the read-path decision, range arithmetic in vLLM-owned kernels, only if the gather profile justifies it

5 The Fork Implementation

The implementation is delivered on the SSCCS vLLM fork (ssccsorg/vllm, branch 52-tagma-kv-vllm):

- csrc/tagma/kv_coord_map.h: the closed-form coordinate arithmetic
- csrc/tagma/kv_allocator.h: the contiguous-range allocator with request and cache reference separation
- csrc/tagma/tagma_bindings.cpp: the vllm.tagma host extension, building through the real CMake target with USE_SABI 3.11
- vllm/v1/core/tagma_kv_cache_manager.py: the scheduler-facing manager, with fail-closed rejection of unsupported configurations
- vllm/v1/worker/gpu/block_table.py: the tagma write path, per-request range tables capped at 256 ranges with fail-closed rejection, and the slot-mapping kernel
- csrc/tagma/bench/bench_kv_engine.cpp: the host benchmark

The vLLM API and user experience stay unchanged: the same `vllm serve` command with a new `--kv-cache-backend tagma` selector, and the same scheduler-facing `KVCacheManager` interface. The ecosystem is not broken; only the interior is replaced. The baseline for the comparison is the paged backend, the default and mature production path of vLLM.

6 Benchmark Methodology

6.1 Host benchmark

The host benchmark `bench_kv_engine.cpp` measures the software-level primitives with a steady-clock harness: each scenario runs `iterations` calls per round over `rounds` rounds, with one warmup call per round, and reports the mean and standard deviation in nanoseconds per call. The JSON summary follows the syntagma export conventions. Conditions:

Item	Value
Host	Apple Silicon arm64, macOS 15.7
Compiler	clang++ -O2 -Wall -Wextra -Werror
C++ standard	C++17
Commit	54149fe5d (vllm fork, branch 52-tagma-kv-vllm)
Coordinate layout	32 layers, 64K blocks, 32 tokens per block, 2048 bytes per token

The slot-mapping model mirrors the two Triton kernels at host scale: the paged row loads one block-table entry per position, the tagma row loads the range descriptor once per request and derives the physical block per position by arithmetic. Both produce identical slot ids for the same physical layout.

6.2 GPU serve benchmark

The serve benchmark (`run-vllm-tagma-benchmark.sh` in the syntagma devlogs) measures the end-to-end comparison on an NVIDIA H100 or A100, the same card for both runs, on ShareGPT at 16K and 128K context: time to first token, decode latency, throughput, and KV cache footprint. This is milestone M6 and is pending.

The measurement targets, with the baseline condition explicit:

Metric	Baseline (vLLM)	Tagma Target	Expected Improvement
Write-path slot mapping	Block-table gather per token position	Range arithmetic, no gather	Measured per-step kernel cost

Metric	Baseline (vLLM)	Tagma Target	Expected Improvement
Read-path block resolution	Materialized table in v0	Range arithmetic for vLLM-owned kernels	Deferred until the benchmark shows the gather matters
Throughput (128K context)	vLLM baseline on the same hardware	Tagma backend	Target to be measured; no prior claim
Memory footprint	Paged blocks	Contiguous ranges	Measured under concurrent load

7 Measured Results

The software level is measured on the host; the GPU level is pending.

7.1 Coordinate arithmetic and allocator bookkeeping

The closed form and the allocator bookkeeping are single-threaded map and arithmetic work on the scheduler thread. The chart shows the per-op cost on a log scale with standard-deviation error bars across rounds; the context walk bar is per token, the others per op.

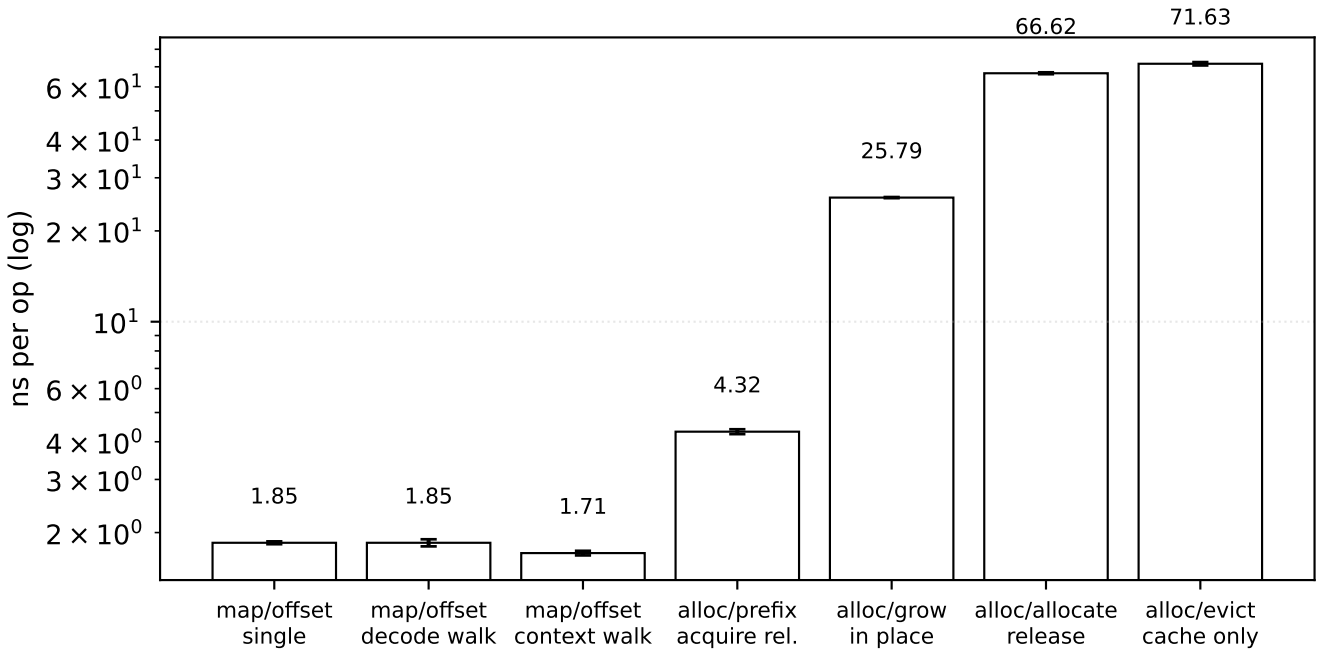


Figure 1: Host-side scheduler-thread costs: coordinate arithmetic and allocator bookkeeping, log scale, standard-deviation error bars across rounds. The context-walk bar is per token; the others are per op.

Scenario	Mean	Stddev	Note
map/offset single	1.85 ns	0.02	one checked closed-form offset
map/offset decode walk	1.85 ns	0.05	one new token per step
map/offset context walk 4k	1.71 ns/token	0.03	prefill walk of one request
alloc/allocate release	66.6 ns/cycle	0.48	one allocate plus one release

Scenario	Mean	Stddev	Note
alloc/grow in place	25.8 ns/step	0.11	decode growth, rollover every 32 steps
alloc/prefix acquire release	4.3 ns/cycle	0.08	one prefix share cycle
alloc/evict cache only	71.6 ns/evict	0.92	LRU eviction of a cache-only range

7.2 Slot-mapping structural model

The model runs the same logical slot mapping through both strategies at host scale. The two charts are complementary dimensions of the same comparison: the measured per-position cost (left) and the deterministic load count behind it (right).

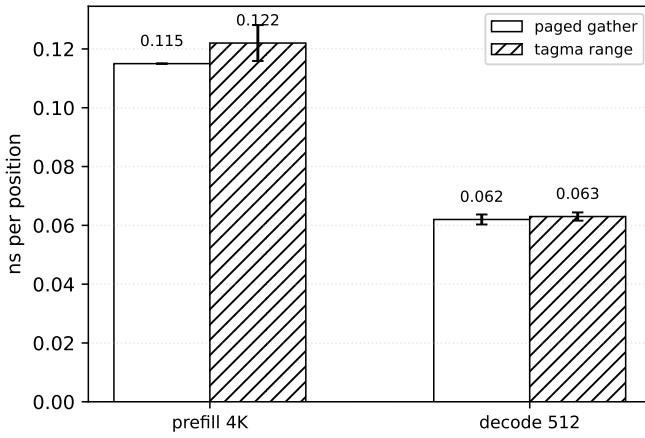


Figure 2: Per-position slot-mapping cost on the host, paged gather against tagma range arithmetic. prefill and decode, standard-deviation error bars. The model measures parity within noise; the on-device profile decides.

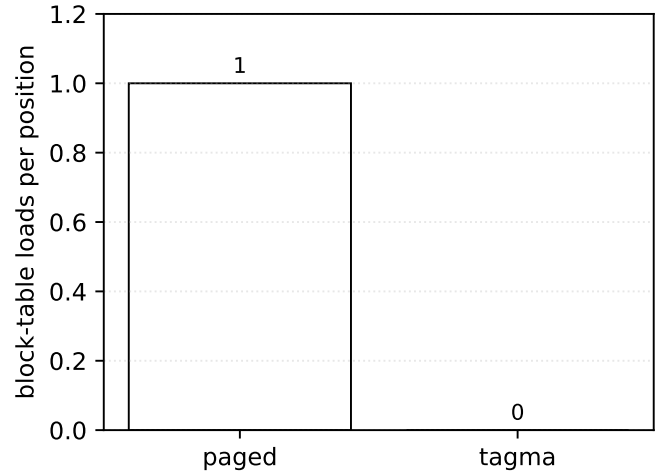


Figure 3: Block-table loads per position by construction. The paged kernel loads one table entry per position; the tagma kernel loads the range descriptor once per position block and derives the physical block by arithmetic. This is the mechanism the host model and the GPU profile measure. It is a deterministic property of the two kernels, not a timing result.

Scenario	Paged gather	Tagma range	Note
prefill, 4K context	0.115 ns/position	0.122 ns/position	parity within noise
decode, 512 requests	0.062 ns/position	0.063 ns/position	parity

8 Cost Analysis

The measured numbers separate two kinds of claims.

The allocator and coordinate numbers bound the per-step bookkeeping of the scheduler thread and the cost of the closed form. They hold on any platform because they are single-threaded map and arithmetic work.

The slot-mapping numbers measure parity on the host: a dense block table is L1-resident, so the gather and the range arithmetic cost the same per position. The host does not reproduce the GPU gather profile: memory divergence, L2 and TLB behavior, and the per-position load instruction in the Triton kernel. The structural claim of the

tagma write path is the mechanism, the absence of a per-position table load, not a measured speedup. The GPU serve benchmark measures that profile, and it decides whether the write-path difference matters on-device and whether the read-path kernels should move to range arithmetic.

The GPU serve benchmark may measure parity. If it does, the conclusion is that the write-path block-table indirection is not the dominant cost of vLLM inference: the structural claim stands as a measured mechanism, not an end-to-end speedup, and the write-path integration is not pursued as a performance contribution.

9 Correctness Verification

The software level is verified by tests that run without a GPU: 16 C++ engine tests (ASan and UBSan clean), 4 config tests, 13 manager integration tests, and 6 write-path range-compression tests, all passing. The four write-path kernel and CUDA integration tests are written and skipped without a GPU: kernel equivalence with the paged kernel, multi-range requests, out-of-range padding, and dummy zeroing. The `vllm.tagma` extension builds through the real CMake target and imports.

10 Status

10.1 Software level: complete

The engine, the host extension, the manager, the write path, the tests, and the host benchmark are implemented and verified. This report records the measured host numbers.

10.2 GPU level: pending

The CUDA machine is the remaining measurement and optimization surface:

- Run the four GPU write-path tests on a CUDA machine
- Run the serve benchmark (`run-vllm-tagma-benchmark.sh`) on an NVIDIA H100 or A100, paged against tagma on the same card, on ShareGPT at 16K and 128K context, and record time to first token, decode latency, throughput, and KV cache footprint
- If the benchmark measures parity, the write-path integration is recorded as a measured negative result and is not proposed upstream as a performance feature
- Profile the block-table gather in the attention kernels on the baseline run: the fraction of kernel time loading `block_table` entries, and the L2 and TLB behavior of the gather against sequential range access
- Profile the per-step materialization copy `_gather_block_tables_kernel` in tagma mode: the staged `block_tables` tensor is written and then copied to `input_block_tables` every step, and the copy is redundant because the range tables already hold the request layout. If the profile shows the copy is a measurable share, replace it with a range-expansion kernel that fills `input_block_tables` directly from the range tables. The execution plan is the vLLM tagma GPU plan in the `syntagma devlogs`
- The read-path optimization, range arithmetic inside the vLLM-owned Triton kernels (`unified_attention`, `context_attention_fwd`), is implemented only if that profile shows the gather is a measurable share; the host model measures parity and does not justify it alone
- MLA and hybrid cache support remain planned; the backend fails closed on those configurations
- Open the upstream pull request after the measured comparison and a human review of every changed line, per the vLLM contribution policy

11 Boundaries

- All numbers in this report are measured on the stated host and reported as they are, with no projection ahead of measurement
- The slot-mapping comparison is a host model; the GPU kernel profile is the on-device evidence
- The tagma backend complements the paged backend; the materialized block table and the external attention backends remain in place
- The tagma backend fails closed on unsupported configurations; MLA and hybrid caches are out of scope for this version

12 Conclusion

The coordinate-indexed write path replaces the per-position block-table gather with range arithmetic. The software level is implemented on the SSCCS vLLM fork and verified at the unit, integration, and host-measurement level: 16 C++ engine tests, 23 Python tests, 11 host benchmark scenarios, and the real CMake extension target. The host slot-mapping model measures parity, which is the honest pre-GPU state of the write-path claim: the mechanism is in place, the on-device evidence is not yet measured. The GPU serve benchmark is the next measured question, and the runbook, the runner script, and the fork are the starting point for it.

13 References

- vLLM KVCacheManager
- PagedAttention: vLLM's memory management
- CERN ROOT-Coord report
- Tagma Map documentation
- SSCCS vLLM fork
- Host benchmark
- Benchmark runbook and runner