GPU SMITH
COMPUTE ENGINEERING

GPU SMITH / RESEARCH & PRACTICAL GUIDANCE

vLLM CUDA Graph Memory Budget: Capture Sizes and KV Headroom

> Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-11-02 · vllm cuda graph memory · vllm cuda graph capture sizes · cudagraph_capture_sizes · vllm gpu_memory_utilization · vllm kv cache headroom · vllm enforce eager · cuda graph out of memory · vllm max_num_seqs · llm inference capacity planning · nvidia gpu memory

Original article: https://gpusmith.com/articles/en/vllm-cuda-graph-memory-budget-kv-headroom



In this article

1. Executive Summary
2. Introduction and Background
3. Key Changes: Graph Modes, Eligibility and Capture Shapes
4. Key Changes: Memory Estimation and KV Sizing
5. Implementation Considerations and Process Changes: The Lifecycle Worksheet
6. Data Analysis and Evidence
7. Implications and Future Directions
8. Frequently Asked Questions (FAQs)
9. Conclusion

Executive Summary

For an existing NVIDIA allocation, the **vLLM CUDA graph memory** question is not "how many bytes does a graph cost" but "which graph modes and capture shapes can be qualified without shrinking KV (key-value) cache below what the latency objective needs." Current vLLM documentation states that CUDA graphs are on by default and "take up extra memory in the GPU" (docs.vllm.ai), that the default mode is **FULL_AND_PIECEWISE**, which "requires the most memory and takes the longest to capture" (docs.vllm.ai), and that capture sizes can be reduced or graphs disabled with the eager flag. Those pages give no per-graph byte figures, so the number has to be measured on the reader's own model, backend, parallelism and revision.

Three facts shape the measurement plan. First, vLLM runs a dummy profiling forward pass, then a separate graph-memory estimation pass, and only then sizes KV cache (raw.githubusercontent.com); graph memory estimation has been on by default since v0.21.0 (raw.githubusercontent.com). Second, the estimate is not the actual: in the vLLM team's September 2026 GB300 experiment, one TP4DP4+MTP row reserved 7.56 GiB for graphs and used 9.90 GiB, while other rows reserved more than they used (vllm.ai). That is a scoped vendor observation, not a reserve rule. Third, the PyTorch allocator sees only its own allocations, so memory held by libraries such as NCCL appears only in device-wide counters (docs.pytorch.org).

The recommended method is a **lifecycle ledger**: per rank, record device-used, allocated and reserved memory at fixed stages (context, weights, profile run, graph estimate, capture, KV initialization, warm steady state, extreme requests), one fresh process per cell and one changed variable per cell. Run two tracks and label them separately: a fixed explicit KV budget, which skips profiling and the graph estimate and isolates execution overhead, and automatic KV allocation, which exposes the full deployment trade-off.

Selection should use observed shape coverage (eligible steps served by a captured graph divided by eligible steps), padding evidence from vLLM's graph metrics, and goodput under the reader's own TTFT (time to first token) and inter-token latency limits. Published cross-runtime evidence is directional only: TensorRT-LLM measured about 10 MB per additional graph on one DeepSeek-R1 configuration (nvidia.github.io), yet a 17.2% KV capacity reduction on another model (DeepSeek-V3.2, where 210 additional graphs beyond a +64 configuration were captured) (nvidia.github.io). Cold capture time, rolling-restart overlap and steady-state memory are separate costs. No GPU benchmark was produced for this report; all figures are quoted from named sources and scoped to their stated conditions.

Introduction and Background

Capacity engineers who own an NVIDIA inference platform often face the same sequence: a launch-overhead diagnosis has already justified testing CUDA graphs, the model and GPU allocation are fixed, and the open question is which graph modes and capture shapes to qualify. This report treats that as a measured execution-memory tuning decision, not a weights or bytes-per-token calculation. GPU Smith, an independent private AI infrastructure engineering consultancy, frames the work as a reproducible worksheet: the output is a measured non-KV budget that a KV capacity plan can consume, together with evidence for or against buying more HBM (high-bandwidth memory).

A CUDA graph records a sequence of GPU operations once and replays it with a single launch. NVIDIA documents that CPU launch cost is reduced because "much of the setup is done in advance" (docs.nvidia.com), and PyTorch documents that a replay skips argument setup and dispatch layers in Python, C++ and the CUDA driver (docs.pytorch.org). The benefit is largest when host overhead exceeds GPU run time, which PyTorch's engineers describe as the small-batch regime (pytorch.org). A preprint measuring batch-1 decode found a 1.259x latency improvement on an H100 at 2,048 context but only 1.028x on an L4 (arxiv.org), which illustrates why the diagnostic comes first and why gains are hardware specific.

The memory cost follows from replay mechanics. A captured graph acts on the same virtual addresses every time it replays (docs.pytorch.org), so static buffers and a graph-private allocator pool must persist (docs.nvidia.com). Capture-time allocations come from that private pool, which "stays alive" until the graph and its tensors go out of scope (docs.pytorch.org). By default each graph gets its own pool, which PyTorch notes "sometimes needlessly wastes memory" (docs.pytorch.org); pools may be shared only if graphs replay in capture order (docs.pytorch.org).

Scope and dating matter. The vLLM documentation pages read for this report are the "latest" builds and carry no version number, and source files were read from the main branch on 2 November 2026 without a recorded commit. Behavior below is therefore a checklist of what to verify in a pinned deployment, not a claim about any released version. The PyTorch 2.8 notes used here are an older pinned reference, not an assertion about the deployed PyTorch version.

Key Changes: Graph Modes, Eligibility and Capture Shapes

Change 1: Modes are separate from compilation and can be downgraded

The current design separates graph behavior from compilation: the redesign "made full CUDA Graphs support orthogonal to compilation" (docs.vllm.ai); full graphs work with or without compilation. Five modes exist. *Table 1* below summarizes what each captures and where its documented memory, startup or fallback implications sit.

Mode	What is captured	Documented memory and startup note	Eligibility and fallback
NONE	No graphs; eager execution.	Fastest startup, at the cost of steady-state decode performance.	Default when piecewise compilation is unavailable.

Mode	What is captured	Documented memory and startup note	Eligibility and fallback
PIECEWISE	Attention stays outside the graph; "everything else goes into CUDA Graphs" (docs.vllm.ai).	Optimization level -O1 uses PIECEWISE graphs.	Needs piecewise compilation; cannot cover whole-model replay.
FULL	Full graphs for all batches.	Largest per-graph scope.	Backend dependent; unsupported backends are downgraded.
FULL_DECODE_ONLY	Decode batches only; mixed prefill-decode batches run without graphs (raw.githubusercontent.com).	Saves the memory PIECEWISE graphs would need (docs.vllm.ai).	Prefill and mixed steps are eager, so measure their share.
FULL_AND_PIECEWISE	Full graphs for decode, piecewise graphs for prefill and mixed batches.	Most performant for most models and the default, with the most memory and longest capture.	Downgrades to the nearest supported mode.

The table shows why "requested mode" and "effective mode" are different ledger columns. The design document states that not all modes work with every attention backend and that vLLM downgrades to the closest supported mode (docs.vllm.ai); a backend that supports graphs only for uniform decode batches turns a FULL request into FULL_AND_PIECEWISE or FULL_DECODE_ONLY (docs.vllm.ai). For hybrid models the minimum capability across all backends decides the model's capability. A run whose startup log shows a different effective mode than the configuration file is a different experiment and must be recorded as such.

Two further eligibility rules affect the sweep. Attention-fusion custom passes that need the whole graph make vLLM disable piecewise compilation and use FULL-style modes. And at runtime, a central dispatcher is the "single source of truth" for which graph keys exist; it searches keys in priority order and, if no key matches, returns NONE, meaning eager execution (docs.vllm.ai). Shape coverage is therefore a dispatch outcome that must be observed, not inferred from the capture list.

Change 2: Default capture sizes are derived and capped

Capture-size defaults are generated, not fixed. When no list is given, sizes are inferred from the config; the source describes 1, 2 and 4, then multiples of 8 up to 256, then multiples of 16 (raw.githubusercontent.com). The ceiling defaults to 512, or 1024 on data center Blackwell GPUs, to avoid out-of-memory (OOM) in tight-memory cases, and the same text notes that larger graphs increase startup time and memory (raw.githubusercontent.com). A user-supplied list overrides the derivation (raw.githubusercontent.com), and a graph captured for one size can only serve that size, so batches between captured sizes are padded upward, which the design notes as a padded token length in the dispatch key.

One documentation discrepancy deserves recording. The conserving-memory example comments that the default list "goes up to max_num_seqs" (docs.vllm.ai), while the source docstring and configuration reference describe a cap of 512 or 1024. The two statements are compatible only when `max_num_seqs` is below the cap, so the ledger should store the actual captured list printed at startup. The same engine-argument reference says `max_num_seqs` "sizes the model runner" and graph capture (docs.vllm.ai), and a separate

`max_num_active_seqs` keeps batches smaller "without shrinking runner or graph capacity". That is the answer to the `max_num_seqs` and CUDA graph memory question: lowering it can shrink both runner buffers and the capture set, but also lowers admitted concurrency, so it is a workload decision before it is a memory one.

A field example shows the other direction. In an August 2026 vLLM post the default cap of 512 covered concurrency up to 3,072 for a decode endpoint, and the authors raised the cap for higher concurrency while stating they were "not certain" the change was required (vllm.ai). Treat that as a reminder to test whether a larger set earns its memory, not as a recommendation.

Change 3: Feature paths add graph variants

Several features change which graphs exist. For speculative decoding, uniform decode batches have query length 1 plus the number of speculative tokens, as in the validation pass (docs.vllm.ai), and many backends support full graphs only for uniform batches. The September 2026 vLLM experiment used MTP (multi-token prediction) with 3 speculative tokens (vllm.ai). Its authors advise treating the MTP concurrency rows as an upper bound rather than reachable concurrency (vllm.ai), a caution that applies to any derived-capacity column.

For LoRA (low-rank adaptation) serving, vLLM by default creates separate graphs for cases with and without active LoRA, at the cost of startup time and slightly higher memory (docs.vllm.ai). Multimodal encoders have their own opt-in graphs (raw.githubusercontent.com) that are orthogonal to decoder graphs. For data parallel (DP) serving, padding to the maximum token count across ranks applies when graphs are enabled (docs.vllm.ai), and it is enabled only when the synchronized mode across ranks is not NONE. Each of these is a refresh trigger in the checklist later in this report.

Key Changes: Memory Estimation and KV Sizing

Change 4: Graph memory is estimated before KV cache is sized

The startup order explains the measurement stages:

- **Profile run:** the worker profiles peak model memory to decide how much the KV cache can use (raw.githubusercontent.com), by executing a forward pass with dummy inputs (raw.githubusercontent.com).
- **Graph estimate:** it then profiles graph memory if graphs will be captured (raw.githubusercontent.com), using a temporary pool to avoid fragmenting the main pool (raw.githubusercontent.com).
- **KV sizing:** the cache is sized from what the budget leaves after those two measurements.
- **Warmup and capture:** kernel warmup precedes capture (raw.githubusercontent.com), and real capture goes large shapes first so smaller ones reuse the pool (raw.githubusercontent.com).

The change that made this default was a pull request merged on 21 April 2026 that enabled graph memory profiling and raised the default `gpu_memory_utilization` to 0.92 to compensate (github.com); the source documents it as enabled by default as of v0.21.0, and a later pull request extended it to Model Runner V2 (github.com). If the estimate is disabled, the code warns that graph memory is "not accounted for" in KV allocation (raw.githubusercontent.com).

The September 2026 vLLM post describes the mechanism in its own words. Because graphs use static addresses, vLLM is "forced to use that estimate to claim the whole amount beforehand" (vllm.ai); it estimates the largest graph and assumes later graphs add a similar extra increment (vllm.ai), because the second-largest graph still costs some extra memory. The consequence is direct: "KV cache size depends directly on how good that estimate is" (vllm.ai). The authors suggest estimating graph size manually when optimizing final KV percentages (vllm.ai).

Two accounting details matter for the ledger. The profiling context splits GPU memory into other processes, torch memory and non-torch memory in the instance (raw.githubusercontent.com), and measures total consumption with `mem_get_info` rather than allocator counters (raw.githubusercontent.com), adding only transient peak headroom so persistent allocations are not double counted (raw.githubusercontent.com). The activation pass runs in eager mode, because real batches may exceed the largest graph and run eagerly (vllm.ai).

Change 5: An explicit KV budget skips profiling

The `kv_cache_memory_bytes` setting defaults to None, meaning vLLM infers the size. When it is set, the documented behavior has four parts:

- **Profiling is skipped:** passing the logged value back on the next boot skips both memory profiling and the graph estimation pass (docs.vllm.ai).
- **Utilization is ignored:** the source logs that profiling was skipped and that the setting does not respect `gpu_memory_utilization` (raw.githubusercontent.com).
- **Errors move:** a conservative value "caps batch concurrency (and therefore throughput)" (docs.vllm.ai), while an optimistic one fails at allocation time.
- **Portability is limited:** the value is valid only on the same GPU with the same initial free memory (docs.vllm.ai).

A separate extensible-KV option sizes the cache from memory free after graph capture rather than a profiling estimate (docs.vllm.ai), and the documented default utilization changes to 1.0 with it. Whether that option is available or enabled in a pinned deployment is a fact to verify.

Change 6: The allocator does not see everything

The utilization fraction is applied to total device memory: the request is computed from total memory and validated against free memory (raw.githubusercontent.com), and the setting is per instance. Inside a process, PyTorch's snapshot tools "only have visibility" into memory obtained through its allocator (docs.pytorch.org), memory allocated directly from CUDA APIs is not seen (docs.pytorch.org), and NCCL is the documented example (docs.pytorch.org). `memory_allocated` counts tensor-occupied memory (docs.pytorch.org), `memory_reserved` counts what the caching allocator manages (docs.pytorch.org), and `mem_get_info` reports device-global free and total memory (docs.pytorch.org). The consequence for graph memory is that the allocator can account for a private pool while device-wide usage rises by more, with the residual attributable to libraries and metadata.

Implementation Considerations and Process Changes: The Lifecycle Worksheet

Step 1: Fix the experiment conditions

Before any cell runs, freeze and record four groups of inputs:

- **Model and runtime:** weights, tokenizer, precision, and the vLLM and PyTorch revisions.
- **Hardware stack:** GPU, driver, CUDA version, attention backend and effective runner.
- **Parallelism and features:** tensor parallel (TP) and DP layout, speculative decoding and LoRA settings.
- **Traffic and targets:** the traffic profile, the quality gate, and the TTFT and inter-token latency (ITL) requirements.

The metric definitions must also be fixed, since vLLM's documentation warns that "metric terminology is not standardized" across tools (docs.vllm.ai). ITL approximates time per output token (TPOT) only when each streamed output holds one token (docs.vllm.ai); with speculative decoding a streamed output can bundle several tokens, so ITL and TPOT diverge and a latency objective written in ITL and one written in TPOT are different gates.

Step 2: Read what is actually captured

For every cell, store four items:

- **Mode:** requested and effective, including any downgrade message.
- **Sizes:** the captured size list printed at startup.
- **Backend:** the graph capability of each attention backend in use.
- **Dispatch:** padded and unpadded token counts and runtime dispatch modes, from the graph metrics option (docs.vllm.ai).

Separate compilation time from capture time and from replay behavior, since the two features are orthogonal.

Step 3: Measure memory across the lifecycle

Table 2 below is the blank ledger; cells marked "(fill)" are left empty for the reader to complete. One copy is filled per rank and per cell; device-used is total minus free from `mem_get_info`, and the last column keeps the non-torch residual explicit.

Stage	Allocated (GiB)	Reserved (GiB)	Device-used (GiB)	Non-torch residual (GiB)	Source and accounting trap
S0 CUDA context, before weights	(fill)	(fill)	(fill)	(fill)	Context and lazily loaded modules are outside the allocator; lazy loading is the default since CUDA 12.3 (docs.nvidia.com).
S1 Weights loaded	(fill)	(fill)	(fill)	(fill)	Record per rank; TP and DP change per-GPU weights.

Stage	Allocated (GiB)	Reserved (GiB)	Device-used (GiB)	Non-torch residual (GiB)	Source and accounting trap
S2 Profile run (eager, peak)	(fill)	(fill)	(fill)	(fill)	Reset peak counters first; `max_memory_allocated` is a peak since program start unless reset (docs.pytorch.org).
S3 Graph estimate reported	n/a	n/a	n/a	n/a	Copy the logged estimate and its window; the estimate pass uses a temporary pool (raw.githubusercontent.com).
S4 After graph capture	(fill)	(fill)	(fill)	(fill)	Compare with the logged actual pool figure (raw.githubusercontent.com) and the capture log line (raw.githubusercontent.com).
S5 KV cache initialized	(fill)	(fill)	(fill)	(fill)	Record the engine's available-KV figure and block count from the log.
S6 Warm steady state	(fill)	(fill)	(fill)	(fill)	Run the production mix long enough for lazy loads and buffers to settle.
S7 Extreme supported requests	(fill)	(fill)	(fill)	(fill)	Longest context, largest batch, LoRA, multimodal and speculative paths; late module loads can fail if VRAM was fully claimed (docs.nvidia.com).

The ledger works only if windows do not overlap. Reserved memory is the sum of allocator segments (docs.pytorch.org), and allocated memory is a subset of it, so adding allocated to reserved double counts. Subtracting a peak from one window from a baseline from another is equally misleading; take stage deltas from the same process and clear peaks between stages. The NVIDIA troubleshooting guide notes that even one graph implies two pools, global and graph-private, whose fragments cannot be shared (docs.nvidia.com), and that each additional pool adds an isolated region (docs.nvidia.com). Freed tensors inside capture remain reserved but unusable (docs.nvidia.com), which is why a graph delta in `memory_reserved` can exceed what the graph "needs".

Define the quantities explicitly:

- **Graph estimate error** E = reported graph reservation estimate minus measured graph-pool quantity, computed only when both cover the same rank, window and pool definition; otherwise leave E blank.
- **Capture residual** = device-used delta across capture minus reserved delta across capture; attribute it to non-torch memory only after confirming no unrelated process changed.
- **Observed captured-shape coverage** = eligible observed steps served by a captured graph divided by eligible observed steps, with padding evidence from the dispatcher metrics.
- **Padding waste** = padded tokens minus unpadded tokens, divided by padded tokens, from the same metrics.
- **Goodput** = completed requests per second meeting the stated service-level objectives (haoailab.com).

NVIDIA also documents a graph memory footprint query for graph memory nodes, with reserved and used attributes (docs.nvidia.com) and a high watermark that can be reset (docs.nvidia.com). Those attributes describe graph memory nodes owned by the driver (docs.nvidia.com); whether they cover PyTorch private pools in a given deployment must be verified before they are used as a cross-check.

Step 4: Sweep a small informed set

Keep the matrix small and run every cell in a fresh process, changing one variable at a time, with identical inputs and enough repeats to see local variability. A defensible first set is:

- **Baseline production captures:** the effective mode and the full captured list from the current deployment.
- **Reduced shape set:** captures concentrated where the shape histogram says traffic lives, with the maximum covering the admitted batch.
- **Justified alternate mode:** for example FULL_DECODE_ONLY for decode-dominant instances, where the docs say it saves the memory PIECEWISE graphs need.
- **Eager control:** graphs disabled, to measure what the production set buys and costs.

Run each cell on two tracks, labeled separately. **Track A (fixed explicit KV budget)** holds KV capacity constant so differences are execution overhead, with the profiling and estimate steps skipped; verify what else the setting overrides in the pinned release. **Track B (automatic KV allocation)** lets each cell size KV itself, so graph memory shows up as lost KV, which is the full deployment trade-off. Comparing eager and graph cells on Track B without recording the KV each received would confuse the two effects; the vLLM post shows why, since its KV capacity depends on the estimate.

Step 5: Choose the capture set and plan rollback

Choose from coverage, padding and SLO-qualified goodput, not from the lowest memory figure. PyTorch's compiler guide says re-recording is profitable when shapes are limited, such as inference batch sizes (docs.pytorch.org), and suggests padding to a few fixed shapes when shapes vary (docs.pytorch.org). Before promotion, require four checks:

- **OOM test:** the extreme cases from S7 complete without allocation failure.
- **Fallback test:** observed coverage and the eager share meet the target, with the dispatcher evidence attached.
- **Correctness test:** outputs match the baseline cell on the agreed quality set.
- **Rollback:** the previous configuration stays deployable.

Treat three costs separately:

- **Cold capture time:** a preprint states capture "often dominates startup" (arxiv.org), and TensorRT-LLM found capture time scales linearly with graph count there (nvidia.github.io). vLLM's optimization levels explicitly trade startup time for performance, and a release note describes opt-in reuse of the memory-profiling result across boots (github.com).
- **Rolling-restart overlap:** when two instances share a GPU during a restart, each instance's utilization setting is independent, so the overlap window needs its own memory check.
- **Steady-state memory:** stages S6 and S7 of the ledger, which decide whether the chosen set survives production traffic.

Data Analysis and Evidence

Published estimate versus actual graph memory

The only first-party vLLM table found that separates the graph estimate from actual use is the September 2026 GB300 post. Its conditions are specific: measurements ran on a GB300 cluster connected via NVLink72 (vllm.ai), with the default 8% reserve of a 276.62 GiB baseline (vllm.ai), and graph rows labeled by topology and MTP setting. The measurement runs used aggregated decode topologies with the FULL_DECODE_ONLY mode (vllm.ai). The authors state the estimate "often exceeds" actual need "by a wide margin" (vllm.ai), and one row shows the opposite (vllm.ai). *Table 3* reproduces the rows as read from the post's table (the fragment links point to the adjacent prose, since table cells are not addressable text), and computes estimate error as estimate minus actual.

Topology (MTP)	Graphs	Estimate (GiB)	Actual (GiB)	Error E (GiB)	E / actual
TP8 (MTP)	11	0.91	0.59	+0.32	+54%
TP8 (MTP)	17	1.28	0.82	+0.46	+56%
TP8 (MTP)	37	1.87	1.88	-0.01	-1%
TP8 (MTP)	85	7.93	6.04	+1.89	+31%
TP4DP4 (MTP)	37	8.35	6.52	+1.83	+28%
TP4DP4 (MTP)	85	7.56	9.90	-2.34	-24%
TP4DP4 (no MTP)	52	8.29	7.51	+0.78	+10%
TP4DP4 (no MTP)	60	8.95	8.07	+0.88	+11%

The spread, from a 24% shortfall to a 56% surplus, is the reason a fixed percentage reserve is an unsafe general rule: sign and size changed with topology, speculation and graph count in a single experiment. For scale only, the post's default 8% reserve of 276.62 GiB is about 22.1 GiB, so the largest shortfall of 2.34 GiB was roughly a tenth of that reserve; this arithmetic describes one fleet and sets no reserve for another. The post also reports that memory outside weights and KV cache comes to about 20 GiB in every case it measured, and it calls out that results are conditioned on its hardware and software. Estimate and actual may not share a window or definition, so the E column is informative only to the extent the post's definitions align with the reader's logs.

Cross-runtime evidence on shape sets

Other runtimes show the same tension, with different numbers. SGLang advises reserving memory for activations and graph buffers when raising its static memory fraction (docs.sglang.ai), enables graphs by default only for small batch sizes (docs.sglang.ai), and warns that larger graph sets consume more memory, so the static fraction may need lowering (docs.sglang.ai). Its static fraction covers weights and the KV pool only (docs.sglang.ai). TensorRT-LLM pads an incoming batch to the next captured size (nvidia.github.io), and its older default captured 23 sizes in a roughly doubling progression (nvidia.github.io), which could cause up to

50% padding waste at high concurrency (nvidia.github.io). It measured about 10 MB per extra graph on DeepSeek-R1 with TP=4 (nvidia.github.io), and found a larger graph footprint cut KV token capacity by 17.2% on DeepSeek-V3.2 against 8.6% on R1 (nvidia.github.io). The lesson is not a transferable number; it is that per-graph cost, padding waste and KV loss depend on the model, so each must be measured. PyTorch's own compiler documentation adds that a separate pool for graph allocations can raise a program's memory (docs.pytorch.org) and that per-launch metadata can be significant when graphs are re-recorded (docs.pytorch.org).

Latency evidence

Latency benefits are conditional. The batch-1 preprint above found the benefit shrank from 1.259x to 1.028x between GPUs (arxiv.org), and a separate cold-start study of vLLM describes startup as predominantly CPU bound (arxiv.org). Latency and throughput trade-offs therefore must come from the reader's service-level objective. vLLM's benchmark CLI accepts goodput objectives as key-value pairs in milliseconds (docs.vllm.ai). A smaller batched-token limit also improves ITL because fewer prefills slow decodes (docs.vllm.ai), a confounder to hold constant across cells.

Cost per accepted output

Cost should be computed from observed accepted output and reader-supplied GPU-hour prices, never from a graph delta divided by theoretical KV bytes. The formula is: cost per million accepted output tokens = (GPU count × price per GPU-hour) ÷ (accepted output tokens per hour ÷ 1,000,000). **(Hypothetical Example)** Suppose cell A (production captures) sustains 36 million SLO-passing output tokens per hour on 8 GPUs and cell B (reduced captures) sustains 33 million, with a reader price of $p per GPU-hour. Cost per million tokens is 8p ÷ 36 = 0.222p for A and 8p ÷ 33 = 0.242p for B, so B would need its freed memory to buy more than 9% extra qualified output to win. These numbers are illustrative, not measurements.

Implications and Future Directions

Capacity decisions. A shortfall justifies more HBM only after the reader has shown that a qualified smaller capture set, or a justified mode change, still leaves KV below what the latency objective needs. For scale, a 2023 paper reported about 65% of a 40 GB A100's memory going to weights and close to 30% to dynamic KV state for a 13B model (arxiv.org), which shows how thin the non-KV margin can be on small-memory parts; NVIDIA describes the H200 as the first GPU with 141 GB of HBM3e (www.nvidia.com). These are context, not purchase advice, and the older paper's setup differs from current serving stacks.

Hypothesis, unproven here. Smaller, coverage-driven capture sets may defer some HBM purchases, but this research produced no fleet measurement. The worksheet exists to test exactly that claim per workload.

Where the ground is moving. Graph memory estimation has changed defaults within 2026, including the utilization default (github.com), a Model Runner V2 extension (github.com) and a release-note item on lower memory for capturing large graph sizes (github.com). Other vendors document similar mechanisms: AMD's HIP graphs documentation describes graph memory pools and notes graphs pay off only for workloads that replay many iterations (rocm.docs.amd.com). Refresh the ledger, not the conclusion, when any of these inputs change.

Refresh checklist. Re-run the affected cells whenever any of the following changes:

- **vLLM, PyTorch, CUDA or driver revision:** estimate logic, defaults and allocator behavior move between releases.
- **Attention backend or effective runner:** eligibility and fallback change which graphs exist.
- **Model or quantization:** per-graph activation size and weights change.
- **Speculative decoding settings:** uniform decode query length changes.
- **LoRA or multimodal enablement:** added graph variants.
- **TP, DP or expert layout:** per-rank weights, padding across ranks and communication buffers change (docs.nvidia.com).
- **Traffic shape histogram:** coverage and padding waste shift with the workload.

Frequently Asked Questions (FAQs)

How much memory do vLLM CUDA graphs use? The documentation says only that they take up extra memory. Published numbers are scoped: one vendor experiment shows 0.59 to 9.90 GiB depending on topology and graph count (Table 3), and another runtime measured about 10 MB per additional graph in one configuration. Measure your own.

Which capture sizes does vLLM use, and can they be changed? By default sizes are inferred from the config; setting `cudagraph_capture_sizes` captures only the specified sizes (docs.vllm.ai), and the conserving-memory guide shows a reduced list.

How does `gpu_memory_utilization` interact with graphs? The fraction is applied to total device memory and is per instance; with graph estimation on, the estimate is part of the budget before KV is sized. Raising the fraction gives KV more space (docs.vllm.ai) but also removes the cushion that absorbs estimate error.

How do graphs change KV memory headroom? With automatic sizing, the graph estimate is deducted before KV is sized, so "KV cache size depends directly on how good that estimate is" (vllm.ai). A larger or denser capture set can therefore reduce KV capacity; read the engine's available-KV figure per cell on Track B instead of converting a graph delta into concurrency.

Does `--enforce-eager` free more KV cache? It skips compilation and capture for the fastest startup. How much KV you gain depends on the estimate and the run's accounting, so compare eager and graph cells on Track B and record the KV each received.

Why does CUDA OOM appear after startup or at capture? NVIDIA lists seven main causes of OOM with CUDA graphs (docs.nvidia.com), including persistent static inputs (docs.nvidia.com) and cached memory that cannot be released during capture (docs.nvidia.com). Late-loaded modules can also fail if VRAM was fully claimed at startup (docs.nvidia.com). Use the ledger to see which stage moved.

What does `max_num_seqs` do to graph memory? It sizes runner buffers and graph capture, so lowering it shrinks both but caps concurrency.

What is the latency and throughput trade-off? Graphs reduce host launch overhead, with gains that depend on GPU and batch size (arxiv.org); the cost is memory and startup time. Decide with SLO-qualified goodput (haoailab.com).

Conclusion

The CUDA graph memory decision on a fixed allocation is an empirical one. vLLM's documentation defines the modes, fallbacks and size derivation clearly enough to design a sweep, but it does not supply a per-graph byte cost, and the one first-party table that compares graph estimates with actual use shows estimates that are both above and below actuals within a single experiment. The defensible response is a lifecycle ledger: record requested and effective mode, captured sizes, observed coverage and padding, then memory per rank at fixed stages in fresh processes, with a fixed-KV track for execution cost and an automatic-KV track for deployment cost.

Qualify a capture set by SLO-qualified goodput and cost per accepted output, test OOM, fallback and correctness paths including extreme requests, and keep a rollback. Treat cold capture time, restart overlap and steady-state memory as separate line items. Share the resulting non-KV budget with the KV planning process, and rerun the affected cells whenever the runtime, backend, model, speculation, LoRA or parallelism changes. Until a cell is measured, any fixed percentage reserve is a guess.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://docs.vllm.ai/en/latest/configuration/conserving_memory
2. https://docs.vllm.ai/en/latest/design/cuda_graphs
3. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/v1/worker/gpu_worker.py
4. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/envs.py
5. https://vllm.ai/blog/2026-09-21-qwen38-pd-serving
6. https://docs.pytorch.org/docs/2.8/torch_cuda_memory.html
7. https://nvidia.github.io/TensorRT-LLM/blogs/tech_blog/blog20_Tuning_CUDA_Graph_Batch_Sizes_for_Higher_Output_Throughput.html
8. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/cuda-graphs.html
9. https://docs.pytorch.org/docs/2.8/notes/cuda.html
10. https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs
11. https://arxiv.org/abs/2605.30571
12. https://docs.nvidia.com/dl-cuda-graph/latest/troubleshooting/memory-issues.html
13. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/config/compilation.py
14. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/config/vllm.py
15. https://docs.vllm.ai/en/latest/configuration/engine_args
16. https://vllm.ai/blog/2026-08-06-qwen35-25k-tps
17. https://docs.vllm.ai/en/latest/api/vllm/config/compilation
18. https://docs.vllm.ai/en/latest/api/vllm/v1/worker/dp_utils
19. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/v1/worker/gpu_model_runner.py
20. https://github.com/vllm-project/vllm/pull/38284

21. https://github.com/vllm-project/vllm/pull/53306
22. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/utils/mem_utils.py
23. https://docs.vllm.ai/en/latest/configuration/optimization
24. https://raw.githubusercontent.com/vllm-project/vllm/main/vllm/v1/worker/utils.py
25. https://docs.pytorch.org/docs/2.8/generated/torch.cuda.memory.memory_allocated.html
26. https://docs.pytorch.org/docs/2.8/generated/torch.cuda.memory.memory_reserved.html
27. https://docs.pytorch.org/docs/2.8/generated/torch.cuda.memory.mem_get_info.html
28. https://docs.vllm.ai/en/latest/benchmarking/cli
29. https://docs.vllm.ai/en/latest/design/metrics
30. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/lazy-loading.html
31. https://docs.pytorch.org/docs/2.8/generated/torch.cuda.memory.max_memory_allocated.html
32. https://haoailab.com/blogs/distserve
33. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__TYPES.html
34. https://docs.pytorch.org/docs/stable/user_guide/torch_compiler/torch.compiler_cudagraph_trees.html
35. https://arxiv.org/abs/2604.06664
36. https://github.com/vllm-project/vllm/releases
37. https://docs.sglang.ai/advanced_features/hyperparameter_tuning.html
38. https://docs.sglang.ai/advanced_features/server_arguments.html
39. https://arxiv.org/abs/2606.07362
40. https://docs.vllm.ai/en/latest/cli/bench/serve
41. https://arxiv.org/pdf/2309.06180
42. https://www.nvidia.com/en-us/data-center/h200
43. https://rocm.docs.amd.com/projects/HIP/en/latest/how-to/hip_runtime_api/hipgraph.html
44. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/bufferreg.html
45. https://docs.vllm.ai/en/latest/design/torch_compile

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the hardware library, GPU references, networking references and vendor research. Contact GPU Smith with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: https://gpusmith.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.