



Private RAG Hardware Sizing: Embeddings, Reranker, Index

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 · 2026-10-29 · rag hardware requirements gpu · embedding model gpu throughput sizing · reranker gpu cost · cagra vs hnsw gpu vector search · cuvs cagra index memory

Original article: <https://gpusmith.com/articles/en/private-rag-hardware-sizing-embedding-reranker-vector-index>



In this article

1. The short answer
 2. RAG pipeline workload anatomy
 3. Embedding serving
 4. Choosing an embedding model: quality, dimension, memory
 5. Index memory math
 6. CPU HNSW
 7. GPU indexes with cuVS and CAGRA
 8. Reranker sizing
 9. Ingestion vs query capacity planning
 10. Reference configurations (derived examples)
 11. Benchmarking protocol
 12. When to co-locate with the LLM
 13. Sources
-

The short answer

In a private RAG deployment the retrieval components are usually smaller than the generator LLM, but they do not size themselves. Three numbers decide the hardware: the **vector index footprint** (vectors x dimensions x bytes per value, plus a graph overhead you can compute from your vector database's own documentation), the **ingestion compute** (corpus tokens x embedding-model FLOPs per token), and the **reranker compute per query** (candidates x tokens per candidate x reranker FLOPs per token). For a 10-million-chunk corpus with 1024-dimension fp32 embeddings, the index alone needs roughly 43 to 52 GB of RAM depending on the database, while a cross-encoder reranker scoring 50 candidates per query costs on the order of 800 times more compute than embedding the query.

Below: worked arithmetic from official docs and model cards, CPU HNSW vs GPU CAGRA, and a measurement protocol, since no published throughput figure will match your chunk lengths and recall target. For the generator, see our [LLM inference hardware sizing guide](#).

RAG pipeline workload anatomy

Two workloads share the same components:

Phase	Steps	Shape of load	What it is sized for
Ingestion	Chunk documents, embed every chunk, build or extend the index	Bulk, throughput-bound, schedulable	A batch window (hours to re-embed the corpus)
Query	Embed the query, ANN search, rerank candidates, generate	Latency-bound, bursty, concurrent	p95 latency at peak QPS

The embedding model is cheap at query time and expensive at ingestion; the reranker runs only on the query path and is usually its most expensive retrieval step; the index is memory-bound on both. Our [batch vs real-time sizing](#) article covers the same split for the generator.

Embedding serving

Hugging Face Text Embeddings Inference (TEI) serves open embedding and reranker models. Its docs list token-based dynamic batching, optimized inference via Flash Attention, Candle and cuBLASLt, Safetensors weight loading, no model graph compilation step, and Prometheus metrics plus OpenTelemetry tracing ([Hugging Face TEI docs](#)). The README covers rerankers (single-class sequence-classification cross-encoders) and an air-gapped path: download weights, then mount them into the container ([TEI GitHub README](#)).

The README for TEI v1.9.4 (released 2026-09-15) publishes 1.9 images for CPU (x86 and arm64), Turing (experimental), Ampere, Ada Lovelace, Hopper and, experimentally, Blackwell. The sizing-relevant flags (defaults verified in the v1.9.4 source) are:

- `--max-batch-tokens` (default 16384): total tokens in one batch, described as "one critical control to allow maximum usage of the available hardware".
- `--max-client-batch-size` (default 32): maximum inputs a client can send in one request.
- `--max-concurrent-requests` (default 512): requests beyond this are refused, not queued.

Because batching is by tokens, plan in **tokens per second**, not requests per second: a 512-token chunk costs roughly 16 times a 32-token query. Measure it with your real chunk-length distribution via TEI's Prometheus metrics.

Choosing an embedding model: quality, dimension, memory

MTEB is the standard reference for comparing models. The original paper covers 8 embedding tasks, 58 datasets and 112 languages, benchmarked 33 models and found that "no particular text embedding method dominates across all tasks" ([Muennighoff et al., MTEB, arXiv 2210.07316](#)). Shortlist on the retrieval subsets in your languages, then validate on your own documents.

Two properties matter for hardware: parameter count (GPU memory, FLOPs) and output dimension (index memory). From model cards and Hugging Face file metadata, October 2026:

Model	Parameters	Dimension	Max tokens	Weight memory
BAAI/bge-base-en-v1.5	109,482,752 (F32 safetensors)	768	512	0.44 GB fp32
BAAI/bge-m3	567,754,752 (derived from config, see note)	1024	8192	2.27 GB fp32 checkpoint
BAAI/bge-reranker-v2-m3 (reranker)	567,755,777 (F32)	n/a (score)	512 in card examples	2.27 GB fp32, 1.14 GB fp16

Model	Parameters	Dimension	Max tokens	Weight memory
Qwen/Qwen3-Embedding-8B	7,567,295,488 (BF16)	up to 4096, user-defined 32-4096	32k	15.13 GB bf16

Sources: [bge-base-en-v1.5 card](#), [bge-m3 card](#), [bge-reranker-v2-m3 card](#), [Qwen3-Embedding-8B card](#). The bge-m3 card gives no parameter count. Its config.json (24 layers, hidden 1,024, FFN 4,096, 250,002-token vocabulary, 8,194 positions) gives 567,754,752 parameters for an XLM-RoBERTa encoder with pooler, which matches the 2,271,145,830-byte fp32 `pytorch_model.bin` ([bge-m3 files](#)) at 4 bytes each; the reranker's 567,755,777 swaps the pooler for a classification head. Weight memory is parameters x bytes per parameter; activations come on top and scale with `--max-batch-tokens`.

A BERT- or XLM-RoBERTa-large-class embedder fits on almost any data center GPU. An 8B embedder is closer to serving a small LLM, and its 4096-dimension output quadruples index memory versus 1024 unless you use its user-defined 32-4096 output dimension; check retrieval quality at the truncated size first.

Index memory math

The base term is always **vectors x dimensions x bytes per value**; databases differ in graph overhead and bookkeeping.

Assumptions (illustrative): 10,000,000 chunks, 1024-dimension embeddings (for example bge-m3), one vector per chunk, no replication.

Raw vectors

- fp32: $10,000,000 \times 1024 \times 4 \text{ B} = 40,960,000,000 \text{ B} = 40.96 \text{ GB}$
- fp16: $10,000,000 \times 1024 \times 2 \text{ B} = 20.48 \text{ GB}$
- Same corpus at 4096 dimensions fp32: 163.84 GB

FAISS IndexHNSWFlat. The FAISS guidelines give memory as " $(d \times 4 + M \times 2 \times 4)$ bytes per vector", with M between 4 and 64 links per vector, and list HNSW as not supported on GPU ([FAISS wiki](#), [Guidelines to choose an index](#)). With M = 32 (an assumption):

- Per vector: $1024 \times 4 + 32 \times 2 \times 4 = 4096 + 256 = 4352 \text{ B}$
- Total: $10,000,000 \times 4352 \text{ B} = 43.52 \text{ GB}$, of which the graph is 2.56 GB

Qdrant. Qdrant's capacity-planning guide gives `dense_size = base * dimensions * bytes_per_dim`, `hnsw_size = base * m * 2 * 4 bytes * 1.2` (the 1.2 covers lower graph layers and bookkeeping), `id_tracker_size = base * 52 bytes`, and recommends about 20% headroom on totals ([Qdrant capacity planning](#)). With m = 16 (the value in Qdrant's own example):

- Vectors: 40.96 GB
- HNSW: $10,000,000 \times 16 \times 2 \times 4 \times 1.2 = 1.536 \text{ GB}$
- ID tracker: $10,000,000 \times 52 = 0.52 \text{ GB}$
- RAM before headroom: 43.02 GB; with 1.2 headroom: 51.62 GB

- Payloads (chunk text, metadata) are extra and default to disk.

pgvector. The README (v0.8.7) gives storage per value as $4 * \text{dimensions} + 8$ bytes for vector, $2 * \text{dimensions} + 8$ for halfvec and $\text{dimensions} / 8 + 8$ for bit; HNSW indexes take vector up to 2,000 dimensions and halfvec up to 4,000; HNSW defaults are $m = 16$ and $\text{ef_construction} = 64$; and builds are "significantly faster when the graph fits into `maintenance_work_mem`" ([pgvector README](#)).

- `vector(1024)`: $4,104 \text{ B} \times 10\text{M} = 41.04 \text{ GB}$ of vector data
- `halfvec(1024)`: $2,056 \text{ B} \times 10\text{M} = 20.56 \text{ GB}$
- Binary quantized expression index on `bit(1024)`: $136 \text{ B} \times 10\text{M} = 1.36 \text{ GB}$, with re-ranking by the original vectors recommended for recall
- These exclude row/page overhead and the HNSW index, which the README does not quantify; measure a sample load.
- A 4096-dimension vector cannot be HNSW-indexed in pgvector without truncation.

Milvus. The Milvus HNSW page states that HNSW "requires high memory overhead to maintain its hierarchical graph structure", with M defaulting to 30 and efConstruction to 360, but gives no per-vector formula ([Milvus HNSW docs](#), v3.0.x; current release v3.0.4). Apply the FAISS-style $M * 2 * 4$ bytes per vector as an approximation and validate with a test load.

cuVS CAGRA (GPU). The cuVS CAGRA guide defines `dataset_size (device) = N x D x B` and `graph_size (host) = N x G x S_idx`, with defaults `graph_degree = 64` and `intermediate_graph_degree = 128`, and 4-byte neighbour IDs ([NVIDIA cuVS CAGRA guide](#)):

- Device dataset: 40.96 GB fp32 or 20.48 GB fp16
- Graph: $10,000,000 \times 64 \times 4 \text{ B} = 2.56 \text{ GB}$
- The guide recommends reserving headroom $H = 0.20$ for IVF-PQ graph builds and $H = 0.30$ for NN-Descent builds, applied as $M_{\text{usable}} = (M_{\text{free}} - M_{\text{other}}) \times (1 - H)$.

Summary for 10M x 1024 dimensions:

Store	Formula source	Estimated RAM / VRAM
FAISS HNSWFlat, $M=32$, fp32	FAISS wiki	43.52 GB
Qdrant, $m=16$, fp32, +20% headroom	Qdrant docs	51.62 GB
pgvector vector (data only)	pgvector README	41.04 GB + index + overhead
pgvector halfvec (data only)	pgvector README	20.56 GB + index + overhead
CAGRA, fp32 dataset + degree-64 graph	cuVS docs	40.96 GB device + 2.56 GB host

At 1024 dimensions the vectors are about 94% of the footprint. Precision and dimension move the bill far more than HNSW parameters.

CPU HNSW

HNSW builds a multi-layer structure of proximity graphs, with search starting at the top layer and descending, giving logarithmic complexity scaling ([Malkov and Yashunin, arXiv 1603.09320](#)). For most private RAG corpora it is enough. Sizing properties:

- **RAM-bound.** Qdrant advises against cold-tiering HNSW because traversal "does many small random reads that suffer badly from disk latency" ([Qdrant capacity planning](#)).
- **Build time is the CPU pain point.** Higher efConstruction "leads to longer indexing time and increased memory usage during construction" ([Milvus HNSW docs](#)); pgvector builds use 2 parallel workers by default and slow down when the graph exceeds maintenance_work_mem ([pgvector README](#)).
- **Query cost is modest** at typical RAG QPS; a CPU server with enough RAM often serves search while GPUs embed and rerank.

GPU indexes with cuVS and CAGRA

NVIDIA cuVS is "a GPU-accelerated library for vector search", and among its stated benefits are fast index build and "Interoperability (build on GPU, deploy on CPU)" ([NVIDIA cuVS docs](#)). CAGRA is its GPU-native graph index.

The CAGRA paper reports, on the authors' hardware and datasets ([Ootomo et al., CAGRA, arXiv 2308.15136, ICDE 2024](#)):

- graph construction 2.2 to 27 times faster than HNSW;
- large-batch query throughput 33 to 77 times higher than HNSW at 90-95% recall, and 3.8 to 8.8 times higher than prior GPU implementations;
- single-query search 3.4 to 53 times faster than HNSW at 95% recall.

This is a 2023 paper (revised July 2024) comparing a GPU with a CPU baseline; the ratio depends on both devices, the dataset and the recall point.

More useful for private RAG is the hybrid mode: converting a CAGRA graph to HNSW lets "the GPU build the graph while the CPU handles search later", and the guide recommends CAGRA "when the index fits in GPU memory" ([cuVS CAGRA guide](#)); see `hnsw.from_cagra` in the [cuVS HNSW Python API](#). NVIDIA lists integrations including FAISS (`GpuIndexCagra`, convertible via `IndexHNSWCagra`), Milvus, Elasticsearch, OpenSearch and Solr ([cuVS integrations](#)). Milvus 2.6.4+ can convert a GPU_CAGRA index into "a CPU-executable format (HNSW-like)" at load via `adapt_for_cpu` ([Milvus GPU_CAGRA docs](#)).

Build memory, worked. For the 10M x 1024 corpus:

- During build the dataset must be in GPU memory: 40.96 GB fp32 or 20.48 GB fp16.
- An NN-Descent initial graph peaks at $N \times (D \times 2 + 276)$ B on device = $10M \times 2,324$ B = 23.24 GB, and $N \times (13 \times I + 912)$ B on host = $10M \times 2,576$ B = 25.76 GB with $I = 128$ ([cuVS CAGRA guide](#)).
- Assumption: a GPU with 48 GB free and nothing else resident. With $H = 0.20$: $48 \times 0.8 = 38.4$ GB usable, which does not hold the fp32 dataset but does hold fp16. With 80 GB free: 64 GB usable, enough for fp32.

- Larger corpora: IVF-PQ initial-graph builds can run in batches over datasets larger than GPU memory, and cuVS ships multi-GPU CAGRA APIs.

When GPU indexing pays: frequent full rebuilds, corpora where CPU builds take hours, or high-QPS batch retrieval. For a static 1M-chunk corpus at tens of QPS, CPU HNSW is simpler.

Reranker sizing

A cross-encoder reads the query and each candidate together, so its cost scales with **candidates x tokens per pair** on every query.

A standard approximation for a Transformer forward pass is $C_{\text{forward}} \approx 2N + 2 n_{\text{layer}} n_{\text{ctx}} d_{\text{attn}}$ FLOPs per token, where N is non-embedding parameters ([Kaplan et al., arXiv 2001.08361, Table 1](#)), with $N \approx 12 n_{\text{layer}} d_{\text{model}}^2$. bge-reranker-v2-m3 uses 24 layers with hidden size 1024 ([model card and config](#)).

Worked estimate (assumptions labelled):

- Non-embedding parameters: $12 \times 24 \times 1024^2 = 301,989,888$ (about 302M; the rest of the 568M total is the 250,002-token vocabulary embedding, which costs a lookup, not matrix FLOPs).
- FLOPs per token at 512-token pairs: $2 \times 301,989,888 + 2 \times 24 \times 512 \times 1024 = 629,145,600$ (about 0.63 GFLOP).
- Assumption: 50 candidates per query, each pair padded or truncated to 512 tokens. Tokens per query: $50 \times 512 = 25,600$.
- Rerank compute per query: $25,600 \times 0.629 \text{ GFLOP} \approx 16.1 \text{ TFLOP}$.
- For comparison, embedding a 32-token query with bge-m3 (same backbone): $32 \times (2 \times 301,989,888 + 2 \times 24 \times 32 \times 1024) \approx 19.4 \text{ GFLOP}$. The rerank step is roughly 830 times more compute.
- At an assumed 20 QPS peak: $20 \times 16.1 \approx 322 \text{ TFLOP/s}$ of sustained demand, before any efficiency loss.

Consequences: at TEI's default `--max-batch-tokens` of 16384, one query's 25,600 rerank tokens span at least two batches, and 50 inputs exceed the default `--max-client-batch-size` of 32 per request ([TEI GitHub README](#)), so tune both or split requests. The cheapest optimizations are on the workload: fewer candidates, shorter chunks or a smaller reranker. The bge-reranker-v2-m3 card notes that fp16 "speeds up computation with a slight performance degradation" ([bge-reranker-v2-m3 card](#)).

Converting FLOPs to GPUs needs an achieved throughput only measurement gives: send real query-plus-candidate sets at rising concurrency and record p95 latency and tokens per second.

Ingestion vs query capacity planning

Ingestion compute, worked. Using the same bge-m3 backbone:

- Assumption: 10M chunks x 512 tokens = 5.12 billion tokens.
- Compute: $5.12\text{e}9 \times 0.629 \text{ GFLOP} \approx 3.22\text{e}18 \text{ FLOP}$.
- If a deployment sustained an assumed 50 TFLOP/s effective: $3.22\text{e}18 / 5\text{e}13 \approx 64,400 \text{ s} \approx 17.9 \text{ hours}$.
At an assumed 200 TFLOP/s effective: about 4.5 hours.

These effective rates are placeholders, not GPU specifications; achieved FLOP/s depends on precision, kernels, padding and length mix. The useful conclusion: a full re-embed scales linearly with corpus tokens and roughly with non-embedding parameters, so an 8B embedder costs over an order of magnitude more.

Plan capacity in three buckets:

1. **Steady query load:** query embedding, ANN search (RAM), reranking (GPU at p95 target).
2. **Incremental ingestion:** usually small; can share query GPUs off-peak.
3. **Full re-embed and rebuild** after a model or chunking change: size for your window; old and new index coexist during cutover.

See [CPU vs GPU inference cost](#) for moving small-model work, such as query embedding, to CPU.

Reference configurations (derived examples)

Computed from the formulas above (1024 dimensions, 512-token chunks, bge-m3-class models); starting points for a test, not validated builds.

Corpus	Raw vectors fp32 / fp16	HNSW RAM (FAISS M=32, fp32)	Full re-embed compute	Suggested starting layout
1M chunks	4.10 / 2.05 GB	4.35 GB	3.2e17 FLOP	Single server; index in RAM; embedder and reranker on one GPU or a GPU slice shared with the LLM
10M chunks	40.96 / 20.48 GB	43.52 GB	3.2e18 FLOP	Dedicated index RAM (Qdrant estimate 51.6 GB with headroom); reranker GPU sized to p95 at peak QPS; consider GPU build for frequent rebuilds
100M chunks	409.6 / 204.8 GB	435.2 GB	3.2e19 FLOP	Quantization or fp16 storage, sharding or replication, GPU (possibly multi-GPU) index builds with CPU serving

Replication multiplies the index figures: Qdrant's formulas start from `number_of_points * replication_factor` ([Qdrant capacity planning](#)).

Benchmarking protocol

Before buying hardware, measure:

1. **Ground truth.** Exact top-k for a sample of real queries by brute-force search.
2. **Recall@k vs QPS curves.** Sweep search parameters (ef for HNSW, itopk_size for CAGRA) and plot recall against throughput, the quality-performance trade-off ANN-Benchmarks was built to evaluate ([Aumüller et al., ANN-Benchmarks, arXiv 1807.05614](#)). Report QPS at your recall target, not peak QPS.
3. **Quantization check.** Verify recall against unquantized results; Qdrant notes some embedding models quantize poorly ([Qdrant capacity planning](#)).
4. **Embedding and reranker throughput** in tokens per second, with p95/p99 latency at rising concurrency.
5. **End-to-end quality** after reranking, on a labelled set from your own documents.

6. **Peak memory** during builds; the CAGRA guide's $(\text{observed_peak} - \text{formula}) / \text{formula}$ headroom method works for any engine.

When to co-locate with the LLM

Sharing GPUs with the generator works when the embedder and reranker need a few GB of weights and their bursts do not collide with prefill peaks, or you isolate them. Options are MIG partitions (hard memory and compute isolation) or MPS and time-slicing (softer); see [GPU sharing with MIG and MPS](#). The cuVS guide warns that running other GPU work beside a persistent CAGRA search kernel "can be fragile" and offers `persistent_device_usage` to reserve less than the full GPU ([cuVS CAGRA guide](#)). Move the reranker to its own GPU or slice once its p95 tracks LLM load, and keep full re-embeds off serving GPUs.

GPU Smith helps teams size and benchmark private retrieval stacks against their own corpora; [contact us](#).

Sources

All URLs accessed 2026-10-28.

1. Text Embeddings Inference documentation. Hugging Face. <https://huggingface.co/docs/text-embeddings-inference/index>
2. text-embeddings-inference README (v1.9 images, flags, rerankers, air-gapped). Hugging Face, GitHub. <https://github.com/huggingface/text-embeddings-inference>
3. Muennighoff, Tazi, Magne, Reimers. MTEB: Massive Text Embedding Benchmark. arXiv 2210.07316, 2022 (rev. 2023). <https://arxiv.org/abs/2210.07316>
4. Malkov, Yashunin. Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. arXiv 1603.09320, 2016 (rev. 2018). <https://arxiv.org/abs/1603.09320>
5. Ootomo, Naruse, Nolet, Wang, Feher, Wang. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. arXiv 2308.15136, 2023 (rev. 2024-07-09), ICDE 2024. <https://arxiv.org/abs/2308.15136>
6. NVIDIA cuVS documentation. NVIDIA. <https://docs.nvidia.com/cuvs/>
7. CAGRA indexing guide (memory footprint, parameters, HNSW interoperability). NVIDIA cuVS. <https://docs.nvidia.com/cuvs/user-guide/api-guides/indexing-guide/cagra.md>
8. cuVS Integrations. NVIDIA. <https://docs.nvidia.com/cuvs/getting-started/integrations.md>
9. cuVS Python API: Neighbors HNSW. NVIDIA. <https://docs.nvidia.com/cuvs/api-reference/python-api-neighbors-hnsw.md>
10. Guidelines to choose an index. FAISS wiki, Meta. <https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index>
11. Capacity planning. Qdrant documentation. <https://qdrant.tech/documentation/capacity-planning/>
12. pgvector README. pgvector, GitHub. <https://github.com/pgvector/pgvector>
13. HNSW index. Milvus documentation. <https://milvus.io/docs/hnsw.md>
14. GPU_CAGRA index. Milvus documentation. <https://milvus.io/docs/gpu-cagra.md>
15. BAAI/bge-m3 model card. Hugging Face. <https://huggingface.co/BAAI/bge-m3>
16. BAAI/bge-m3 repository files. Hugging Face. <https://huggingface.co/BAAI/bge-m3/tree/main>

17. BAAI/bge-reranker-v2-m3 model card. Hugging Face. <https://huggingface.co/BAAI/bge-reranker-v2-m3>
18. BAAI/bge-base-en-v1.5 model card. Hugging Face. <https://huggingface.co/BAAI/bge-base-en-v1.5>
19. Qwen/Qwen3-Embedding-8B model card. Hugging Face. <https://huggingface.co/Qwen/Qwen3-Embedding-8B>
20. Kaplan et al. Scaling Laws for Neural Language Models. arXiv 2001.08361, 2020. <https://arxiv.org/abs/2001.08361>
21. Aumüller, Bernhardsson, Faithfull. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. arXiv 1807.05614, 2018. <https://arxiv.org/abs/1807.05614>

Source links

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

1. <https://huggingface.co/docs/text-embeddings-inference/index>
2. <https://github.com/huggingface/text-embeddings-inference>
3. <https://arxiv.org/abs/2210.07316>
4. <https://huggingface.co/BAAI/bge-base-en-v1.5>
5. <https://huggingface.co/BAAI/bge-m3>
6. <https://huggingface.co/BAAI/bge-reranker-v2-m3>
7. <https://huggingface.co/Qwen/Qwen3-Embedding-8B>
8. <https://huggingface.co/BAAI/bge-m3/tree/main>
9. <https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index>
10. <https://qdrant.tech/documentation/capacity-planning>
11. <https://github.com/pgvector/pgvector>
12. <https://milvus.io/docs/hnsw.md>
13. <https://docs.nvidia.com/cuvs/user-guide/api-guides/indexing-guide/cagra.md>
14. <https://arxiv.org/abs/1603.09320>
15. <https://docs.nvidia.com/cuvs>
16. <https://arxiv.org/abs/2308.15136>
17. <https://docs.nvidia.com/cuvs/api-reference/python-api-neighbors-hnsw.md>
18. <https://docs.nvidia.com/cuvs/getting-started/integrations.md>
19. <https://milvus.io/docs/gpu-cagra.md>
20. <https://arxiv.org/abs/2001.08361>
21. <https://arxiv.org/abs/1807.05614>

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.