Unlocking GPU Powers for Vector Search in OpenSearch

Chintan Agrawal | Solutions Architect, AWS

OpenSearchCon India 2026

Have You Hit This Wall?

You built a vector search index. It works great.
Then your data grows. You need to rebuild the index.
Suddenly your search latency spikes 3x while the rebuild runs.
Users complain. You schedule rebuilds at night. But now your data is stale during the day.

Or worse: your ML team retrains the embedding model weekly.
Every retrain means ALL vectors change. Full rebuild. Every. Single. Time.

The root cause: HNSW graph construction and search queries fight over the same CPU.

8.5hr
rebuild time

100%
CPU during build

3.3x
p95 latency spike

Why: HNSW Is CPU-Hungry by Design

```
HNSW Build Complexity:



For each new vector:
  1. Find entry point (top layer)
  2. Greedily descend layers
  3. Search neighbors at each layer
  4. Connect to m nearest neighbors
  5. Repeat for all layers

Complexity: O(n * m * log(n))
  n = number of vectors
  m = edges per node (typically 16)
  log(n) = layer traversal depth

At 1B vectors:
  n=1,000,000,000, m=16, log(n)=30
  = 480 billion distance calcs
  = sequential per-vector insertion
  = 32 hours on 16 CPU cores
```

What happens to search:

Write Load	CPU	P95	QPS
Idle	15%	12ms	1,200
2 clients	67%	75ms	1,200
4 clients	72%	80ms	1,150
8 clients	**96%**	**120ms**	**850**
16 clients	**100%**	**250ms**	**400**

Same JVM heap. Same CPU cores.
GC pauses spike. Search threads starve.

This Is Not a Hypothetical

Production systems today:

Amazon IP Detection

68 billion vectors
8B page changes/day
99% infringements auto-blocked

Every product listing change checked against known IP violation patterns in real-time.

Music Personalization

1.05 billion vectors
Rebuilt daily
Item-item collaborative filtering

Embedding model retrained every 24hr. All vectors change. Entire index must rebuild.

Enterprise AI Agents

100s of millions of vectors
1M updates daily
85% tickets auto-resolved

RAG over support, product, sales data. High-velocity updates with low-latency search.

The Core Insight

Indexing and search don't have to share the same hardware.

Offload the expensive part (graph construction) to purpose-built hardware (GPU).
Keep CPU free for what it's good at (search).

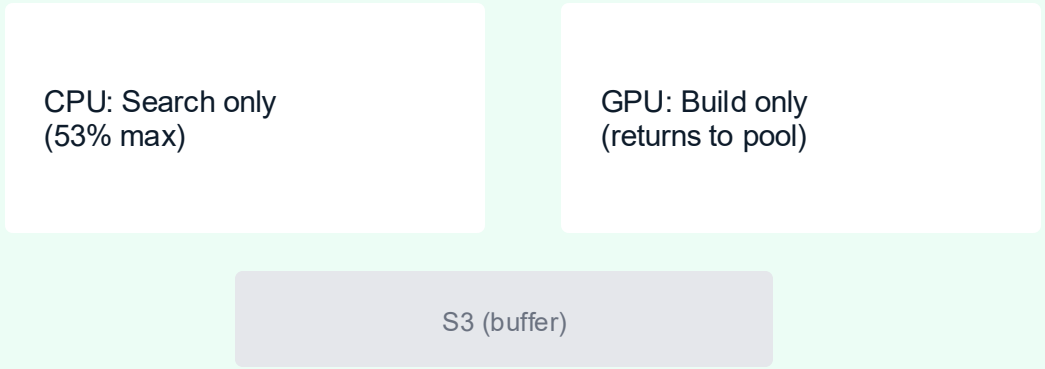
Two workloads. Two hardware types. Zero contention.

Before vs After

BEFORE: Shared CPU

- P95 latency: 250ms (3.3x spike)
- CPU: 100% utilization
- Search QPS drops 67%
- Must schedule off-hours builds

AFTER: Decoupled GPU



- P95 latency: <100ms (flat)
- CPU: never >53% during builds
- Search QPS drops only 12%
- Index anytime, no scheduling

Architecture: Three Decoupled Components

OpenSearch Cluster

(your existing nodes)

Bulk API ingestion
Lucene segment flush/merge
Search query execution
Threshold detection (>50MB)
Automatic CPU fallback on failure

Object Store

(S3, Azure, GCP, or BYO)

Raw vectors uploaded here
Built graph downloaded here
Enables retry on failure
Decouples cluster + GPU lifecycle
No data loss on GPU crash

Remote Build Service

(your GPU fleet, Docker images)

CAGRA graph construction
cuVS + Faiss integration
3 Docker images (base/core/API)
Hardware-agnostic, cloud-agnostic
Scale up/down as you need

Key: GPU fleet is separate from your search cluster. Deploy anywhere. Any cloud. Any hardware.

The Build Flow

1	2	3	4	5	6
Ingest	**Detect**	**Upload**	**GPU Build**	**Download**	**Integrate**
Bulk API writes vectors to Lucene	Volume > 50MB per refresh	Raw vectors to S3 (durable)	CAGRA builds HNSW graph (parallel)	Built graph back to cluster	Merge into Lucene segments

Trigger: GPU path activates during flush/merge ops when segment > threshold. Not per-vector.
If GPU fails at any step: automatic CPU fallback. No data loss (S3 durable). Transparent to app.
Streaming: small flushes stay on CPU. GPU kicks in when merged segments grow large enough.

The Algorithm: CAGRA (cuVS)

CUDA ANNS GRaph-based | NVIDIA cuVS | Builds HNSW-compatible graphs on GPU

1 Build Initial k-NN Graph

GPU

NN-DESCENT or IVF-PQ on GPU. Goal: build a "good enough" graph fast (not perfect). Thousands of CUDA cores compute distances.

Not aiming for perfection here. Just a decent starting graph.

2 Optimize + Reverse Edges

GPU

Prune redundant edges. Then build REVERSE edge graph (B->A, C->B) and merge with forward graph. Ensures all nodes reachable.

Directed graph can have unreachable regions. Reverse edges fix this.

3 Convert to HNSW (base layer only)

CPU

graph_degree / 2 = M parameter. Copy nodes + neighbors to HNSW base layer. Search: random entry points + greedy descent. Same API as CPU-built.

Only base layer (no hierarchy). Top layers optional (+30% cost).

Integration Stack

OpenSearch k-NN Plugin (Java)	JNI bridge to native libs. Segment lifecycle, threshold detection, fallback logic.
Faiss Library (C++)	HNSW implementation, quantization codecs (SQ, PQ, binary), distance functions, IO format.
NVIDIA cuVS (C++/CUDA)	GPU memory management, CUDA kernel dispatch, multi-GPU support, device allocation.
CAGRA + IVF-PQ Kernels	Graph build kernels, pruning algorithms, HNSW format conversion, batch processing.
GPU Hardware (A10G / L4)	24GB VRAM, 600GB/s bandwidth, PCIe Gen4, FP16/FP32/INT8 tensor cores, 8960 CUDA cores.

You call the same OpenSearch API. The system decides GPU or CPU. Transparent.

Why Not Just Use GPU Instances for Everything?

	CPU Cluster (status quo)	GPU Cluster (always-on)	Decoupled (this approach)
Instances	3x r8g.12xlarge (384GB RAM)	6x g6.12xlarge (192GB + GPU)	**3x r6g.4xlarge + separate GPU fleet**
Cost/build (1B)	1x (baseline)	2.4x	**0.27x**
GPU idle time	N/A	70%+ (builds <30%)	**0% (scale down when done)**
Search impact	Severe (CPU saturated)	Still contention (shared JVM)	**Zero (decoupled)**
Scaling	Vertical only	Expensive	**Elastic (auto)**

Analogy: spin up GPU instances for 2 hours vs keeping them running 24/7 idle.

What This Means For You

As an engineer running vector search in production:

24/7

Rebuild Anytime

No more scheduling builds at 2am.
Retrain your model Monday morning,
rebuild index Monday morning.
Users never notice.

<100ms

Search Stays Fast

P95 stays under 100ms even during
heavy indexing. CPU never exceeds 53%.
No more "stale data" vs "slow search"
tradeoff.

7.8x

Pay Per Build

GPU instances return to pool when done.
No idle GPU costs. At 1B daily rebuild:
$48/day vs $375/day (7.8x savings).
Scales to zero between builds.

1 line

No GPU Expertise Needed

One index setting to enable.
No CUDA code. No GPU instance management.
Automatic fallback to CPU on failure.
Same API, same query syntax.

When Does GPU Help? (Sizing Guidance)

GPU acceleration helps when:

- Write volume > 50MB per refresh interval
- Frequent full rebuilds (daily/weekly model retraining)
- Index size > 1M vectors (sweet spot: 10M+)
- Search latency SLA during indexing
- Cost matters (GPU builds cost 4-12x less)
- You use Faiss engine + HNSW algorithm

GPU may not be needed when:

- Small indexes (< 500K vectors)
- Infrequent writes (batch once/month)
- No latency SLA (batch offline OK)
- Using Lucene engine (not supported)
- Using nmslib (not supported)
- Write volume < 50MB/refresh (below threshold)

Threshold + streaming behavior:

GPU triggers during flush/merge, not per-vector. Threshold: 50MB (configurable). Upper bound: GPU VRAM limit.
50MB at 768D FP32 = ~16K vectors. At FP16 = ~32K. At 128D = ~98K vectors in that segment.
Streaming: small segments flush to CPU (or skip build entirely). As merges grow segments past threshold, GPU kicks in.
Batch rebuild: always hits GPU (segments are large). You don't have to switch anything. Automatic.

Proof: Benchmarks

All results reproducible via OpenSearch Benchmark + CDK templates on GitHub.

Build Time: CPU vs GPU (Managed Clusters)

Dataset	Dims	CPU Build	GPU Build	Speedup	Cost Savings
1M vectors	768	1.4 hr	**9.9 min**	**8x**	8x
10M vectors	768	8.5 hr	**36.8 min**	**14x**	12x
113M vectors	1024	28.7 hr	**4.5 hr**	**6x**	6x
1B vectors	128	31.9 hr	**2.8 hr**	**11x**	10x
1B (index-only)	**128**	**N/A**	**35 min**		

Cluster: 3x r6g.4xlarge (16 vCPU, 128GB) | GPU: 3x g5.2xlarge (A10G, 24GB VRAM)
Config: Faiss, HNSW m=16, ef_construction=256 | Multi-AZ, replication on
Embeddings: Cohere Embed V2 (768D) / SIFT-1B (128D) | Reproducible via GitHub CDK + OSB

Serverless: Cost Comparison

Dataset	OCU-hrs (CPU)	OCU-hrs (GPU)	Cost (CPU)	Cost (GPU)	Savings
1M (768D)	8	1.5	$1.92	**$0.36**	**5.3x**
10M (768D)	78	20.3	$18.72	**$4.87**	**3.8x**
113M (1024D)	2,721	304.5	$653	**$73**	**8.9x**
1B (128D)	1,562	201	$375	**$48**	**7.8x**

Why GPU is cheaper per build:

HNSW = embarrassingly parallel.
A10G: 8,960 CUDA cores vs 16 CPU cores.
Completes in 1/10th the time.
Total OCU-hours = far less.

Analogy: 10 workers x 1 hr
vs 1 worker x 10 hrs.

Daily 1B rebuild at production:

CPU: $375/day = $136K/year
GPU: $48/day = $17.5K/year

$119K/year savings

$0.24/OCU-hr (Vector Acceleration), us-east-1

Search Stays Fast During Rebuild

The real question: does search degrade while you index?

WITHOUT GPU (shared CPU)

Clients	CPU%	P95	QPS
2	67%	75ms	1,200
4	72%	80ms	1,150
8	96%	120ms	850
16	**100%**	**250ms**	**400**

WITH GPU (decoupled)

Clients	CPU%	P95	QPS
2	30%	75ms	1,200
4	35%	80ms	1,180
8	42%	90ms	1,100
16	**53%**	**100ms**	**1,050**

The difference:

Without GPU: at 16 write clients, search QPS drops 67% (1200 to 400). P95 goes 3.3x.
With GPU: QPS drops only 12% (1200 to 1050). P95 stays under 100ms. CPU never exceeds 53%.

No more "index at night, search during the day." Both run simultaneously. Users never notice.

What's Improved Since GA

OpenSearch 3.0 to 3.7: 13 months of rapid iteration (May 2025 to Jun 2026)

Release Timeline

3.0 — May 2025

Remote GPU build (GA)

FP32 only. Faiss engine + HNSW required. 50MB threshold trigger.

3.2 — Aug 2025

Full type support for GPU builds

FP16, byte (INT8), binary vectors now buildable on GPU.

3.5 — Feb 2026

Performance leap

Throughput: 450 to 1,500 req/s (+310%). P99 latency: 330ms to 91ms.

3.6 — Apr 2026

First LTS + 1-bit scalar quantization

32x compression, +24% recall vs binary. Faiss 40% faster. Prefetch 2x.

3.7 — Jun 9, 2026

1-bit scalar ON GPU + 5.5x retrieval

GPU builds with 1-bit quant. 5.5x faster vector retrieval. Base64 binary encoding.

3.5 Performance Leap: How

Throughput

450 → 1,500 req/s

+310% improvement

P99 Latency

330ms → 91ms

3.6x reduction

What changed (cumulative through 3.6):

- FP16 bulk SIMD operations (AVX-512 / SVE on Graviton3) for distance computation
- Segment-level parallelism: concurrent graph builds across shards
- Smarter graph traversal: pruned candidate lists, better entry point selection
- 3.6: Hardware prefetch hints for graph hops (2x prefetch improvement)
- 3.6: Faiss quantization kernels rewritten (40% faster distance computation)
- 3.6: Lucene-on-Faiss hybrid engine (77-154% QPS improvement for hybrid search)

GPU Builds: Full Precision Spectrum

Sep 2025: FP32 only. Today: every vector type builds on GPU.

Type	Size/Vector (D dims)	Compression	GPU Since	Best For
FP32	4D bytes	**1x (baseline)**	3.0	Maximum precision, no loss
FP16	2D bytes	**2x**	3.2	Default for most models
Byte (INT8)	D bytes	**4x**	3.2	Quantization-aware training
Binary	D/8 bytes	**32x**	3.2	First-stage retrieval + re-rank
1-bit Scalar	**D/8 bytes**	**32x**	**3.7**	**+24% recall vs binary, same cost**

1-bit Scalar Quantization (3.6 search / 3.7 GPU build):

Maps each float to {-1, +1} using per-dimension median as threshold (adapts to YOUR data distribution).
Same 32x compression as binary, but 24% better recall because threshold is learned, not fixed (sign bit).
5.5x faster vector retrieval in 3.7. Eliminates 50-70% redundant vector transfer (in progress).

Beyond Dense Vectors: New Search Modes

Late Interaction (ColBERT)

Native lateInteractionScore function (3.3+)
Multi-vector per document, token-level MaxSim
Better for long passages, tables, structured docs
No external re-ranker needed

Higher relevance
for complex queries

Neural Sparse v3

Learned sparse representations (3.3+)
15 languages supported
0.546 NDCG@10 (BEIR average)
Lower cost than dense (fewer dimensions active)

3-5x cheaper
than dense retrieval

Lucene-on-Faiss Hybrid

BM25 scoring + Faiss vector compression (3.5+)
77-154% QPS improvement over pure Lucene
Best of keyword + semantic in one query
Automatic score normalization

2.5x QPS vs
pure Lucene hybrid

Ecosystem: Agentic AI + Competitive Landscape

OpenSearch for AI Agents (2026):

- MCP Server (open source): works with Claude, Cursor, Kiro. NL to OpenSearch queries.
- OpenSearch Launchpad: AI builds full search apps from natural language description.
- Relevance Agent: multi-agent system for automated relevance tuning (NDCG, MRR, P@k).
- NextGen Serverless (May 2026): GPU ON by default, scale-to-zero, auto-quant selection.

GPU Vector Search Landscape (Jun 2026):

Engine	GPU Approach	Status	Differentiation
OpenSearch	**cuVS/CAGRA, decoupled**	**GA Dec 2025**	**Decoupled + serverless + auto-optimize + OSS**
Milvus	Full cuVS integration	GA 2025	GPU search path available, 98% recall@6ms
Elasticsearch	cuVS plugin	Announced May 2026	6 months behind, plugin vs native
Qdrant	Vulkan (vendor-agnostic)	Available	NVIDIA + AMD + Intel Arc support

What's next: GPU-accelerated search (not just build), eliminate redundant vector transfer (graph-only download), build indexes larger than GPU VRAM.

Try It Yourself

One setting to enable. Open source. CDK templates on GitHub.

Enable GPU Builds: One Setting

```
PUT /my-vector-index/_settings
{
  "index.knn": {
    "remote_index_build.enabled": true
  }
}

# Requirements:
#   Engine: Faiss (not Lucene or nmslib)
#   Algorithm: HNSW
#   OpenSearch >= 3.0
#   Threshold: ~50MB vectors per refresh
#
# That's it. GPU fleet is fully managed.
# Fallback to CPU is automatic.
```

Resources:

- github.com/opensearch-project/k-NN

- Remote Vector Index Builder repo
(3 Docker images: base, core, API)

- Base: cuVS + CAGRA + Faiss GPU
Core: builder + GPU->CPU conversion
API: conforms to OpenSearch contract

- BYO hardware: swap base image
BYO cloud: swap core image
Multi-tenant: customize API layer

- OpenSearch Benchmark workloads
(reproduce all numbers in this talk)

- CDK templates for full deployment

On AWS: It Just Works

Amazon OpenSearch Service handles the undifferentiated heavy lifting:

Managed Clusters

- GPU warm pool pre-allocated (no cold start)
- S3 object store managed for you
- Auto-scaling GPU fleet (scales with write load)
- Automatic fallback to CPU on failure
- Auto-Optimize integrated (one-click)
- Multi-AZ, encryption, IAM built-in
- OpenSearch 3.0+ (GA since Dec 2025)

You manage: nothing extra.
Enable the setting. Done.

Serverless Collections

- GPU acceleration ON by default (NextGen)
- Scale-to-zero (no min OCU when idle)
- No cluster sizing decisions
- Automatic quantization selection
- Vector ingestion pipeline built-in
- Pay per OCU-hour consumed
- $0.24/OCU-hr for vector acceleration

You manage: nothing.
Create collection. Ingest. Search.

Your Choice: Self-Hosted vs Managed

Same engine, same algorithm, same results. Different operational burden.

	Self-Hosted (OSS)	Managed Clusters	Serverless
GPU fleet	You deploy + scale (Docker images on EC2/EKS)	Managed warm pool (pre-allocated)	Fully transparent (no fleet concept)
Object store	You configure S3/Azure/GCP	Managed for you	Managed for you
Threshold tuning	You configure	Defaults work, tunable	Automatic
Auto-Optimize	Not available (OSS)	Integrated, one-click	Automatic by default
Multi-tenancy	You build (API layer)	Per-domain isolation	Per-collection isolation
Cost model	EC2 instances (you size)	Instance hours	Pay per OCU-hr used
Best for	Full control, multi-cloud, custom hardware	Production workloads, control over sizing	Zero-ops, variable workloads, fast start

Auto-Optimize

You tell it your constraints. It finds the optimal config in 30 minutes.

The Configuration Problem

```
Parameter space (~2,400 combos):



HNSW Algorithm:
  m: [4, 8, 16, 32, 64]
  ef_construction: [64, 128, 256, 512]
  ef_search: [50, 100, 200, 500, 1000]

Quantization:
  None | Scalar (4x) | Binary (32x)
  Product (8-64x) | 1-bit Scalar (32x)

Engine Mode:
  Memory-optimized
  Disk-optimized (with mmap)
  On-disk with Faiss SQ/PQ

Manual exploration: weeks of eng time
Get it wrong: 4x overcost or poor recall
```

Impact of wrong config (1B vectors):

Metric	Default Config	Optimized
RAM usage	4,646 GB	**1,161 GB**
Instances needed	12x r6g.4xl	**3x r6g.4xl**
Monthly cost	~$25,000	**~$6,000**
Recall	0.92	**0.97**
P95 latency	45ms	**12ms**

Published: 80.9% RAM reduction, +8.4% recall gain.

4x cost difference between "default" and "optimized"
for the same data, same hardware.

How Auto-Optimize Works

1

You Provide

Your data (Parquet in S3)
+ Constraints:
  min_recall >= 0.95
  max_latency <= 20ms
  optimize_for: cost

→

2

System Samples

Samples your data distribution
Computes exact k-NN (ground truth) on sample
Runs on GPU in parallel

→

3

Bayesian Search

Explores config space via Bayesian optimization
Builds candidate indexes on GPU, evaluates each

→

4

Delivers

3 recommendations:
  - Best recall config
  - Best cost config
  - Balanced
+ Instance sizing

GA since OpenSearch 3.0 (May 2025). Serverless job, flat-rate pricing. Completes in 30-60 min. No vector search expertise needed.

Key Takeaways

1. **The problem is real: HNSW builds saturate CPU and kill search latency**

   100% CPU, 3.3x p95 spike, 67% QPS drop during indexing

2. **Solution: decouple build (GPU) from search (CPU) via object store**

   Ephemeral GPUs, warm pool, automatic fallback, zero lock-in

3. **9-14x faster builds, 3.75-12x cost reduction**

   1B vectors: 32hr to 2.8hr. Index-only: 35 min. $119K/yr savings at scale.

4. **Full precision: FP32, FP16, byte, binary, 1-bit scalar (3.7)**

   32x compression with +24% recall vs binary. GPU builds for all types.

5. **Auto-optimize: 30 min vs weeks of manual config tuning**

   80.9% RAM reduction, +8.4% recall gain. Set constraints, get answers.

6. **Open source, CDK-deployable, one setting to enable**

   github.com/opensearch-project/k-NN. Try it today.

Thank You

Questions?

github.com/opensearch-project/k-NN

opensearch.org/blog

opensearch.org/docs/latest/search-plugins/knn/

Chintan Agrawal | achintan@amazon.com