Jasper: Updatable Approximate Nearest Neighbor Search on GPU

Prashant Pandey, Northeastern University
https://prashantpandey.github.io/

Hunter McCoy

Zikun Wang

Jasper: Updatable Approximate Nearest Neighbor Search on GPU

Prashant Pandey, Northeastern University
https://prashantpandey.github.io/

Hunter McCoy

Zikun Wang

Jasper: Updatable Approximate Nearest Neighbor Search on GPU

Prashant Pandey, Northeastern University
https://prashantpandey.github.io/

Hunter McCoy

Zikun Wang

The nearest neighbor search problem

Recommendation systems

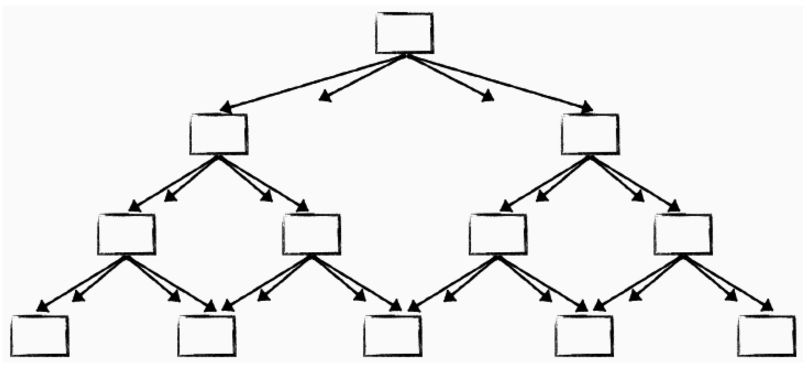
Image retrieval

RAG for LLMs

Anomaly detection

Finding *k* closest points to a query in high-dimensional space

Why approximate? The curse of dimensionality



k-d trees/quad
trees don’t scale

Distance concentration
in high dimension

Modern embedding:
>1000 dimensions

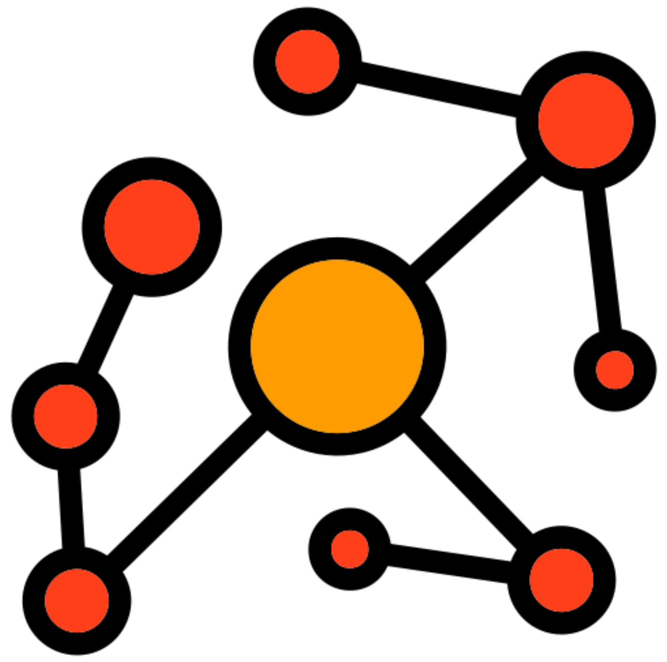
Three major ANNS paradigms

Inverted file index

Locality sensitive hashing

Navigable graph index

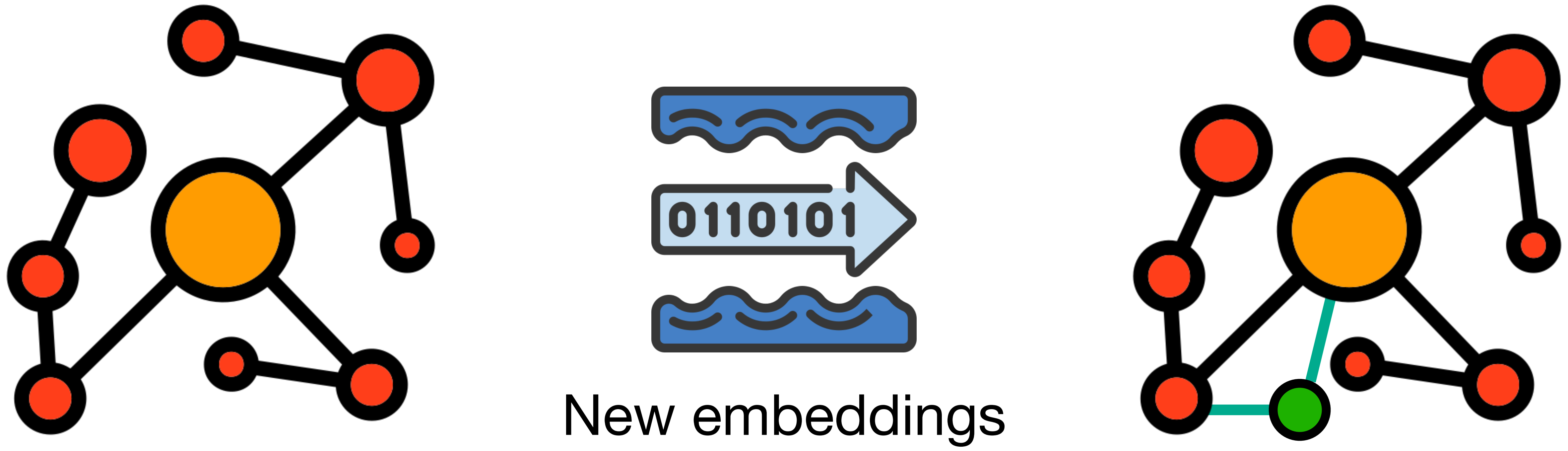
Graph-based ANNS dominate performance-accuracy tradeoff



Navigable
graph index

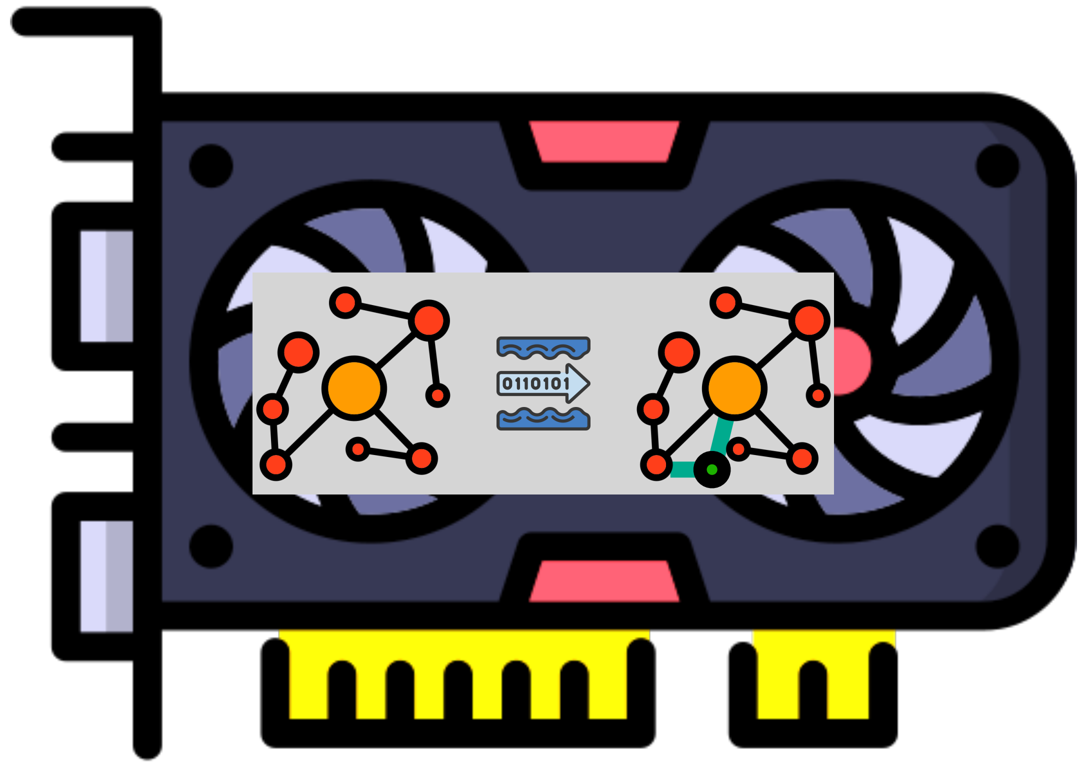
Navigable search graphs with greedy beam search traversal

The critical need for updatability



- Real-world data arrives continuously (100K new tracks/day on streaming, real-time e-commerce updates)
- RAG systems must incorporate fresh documents
- Static indices require costly full reconstruction

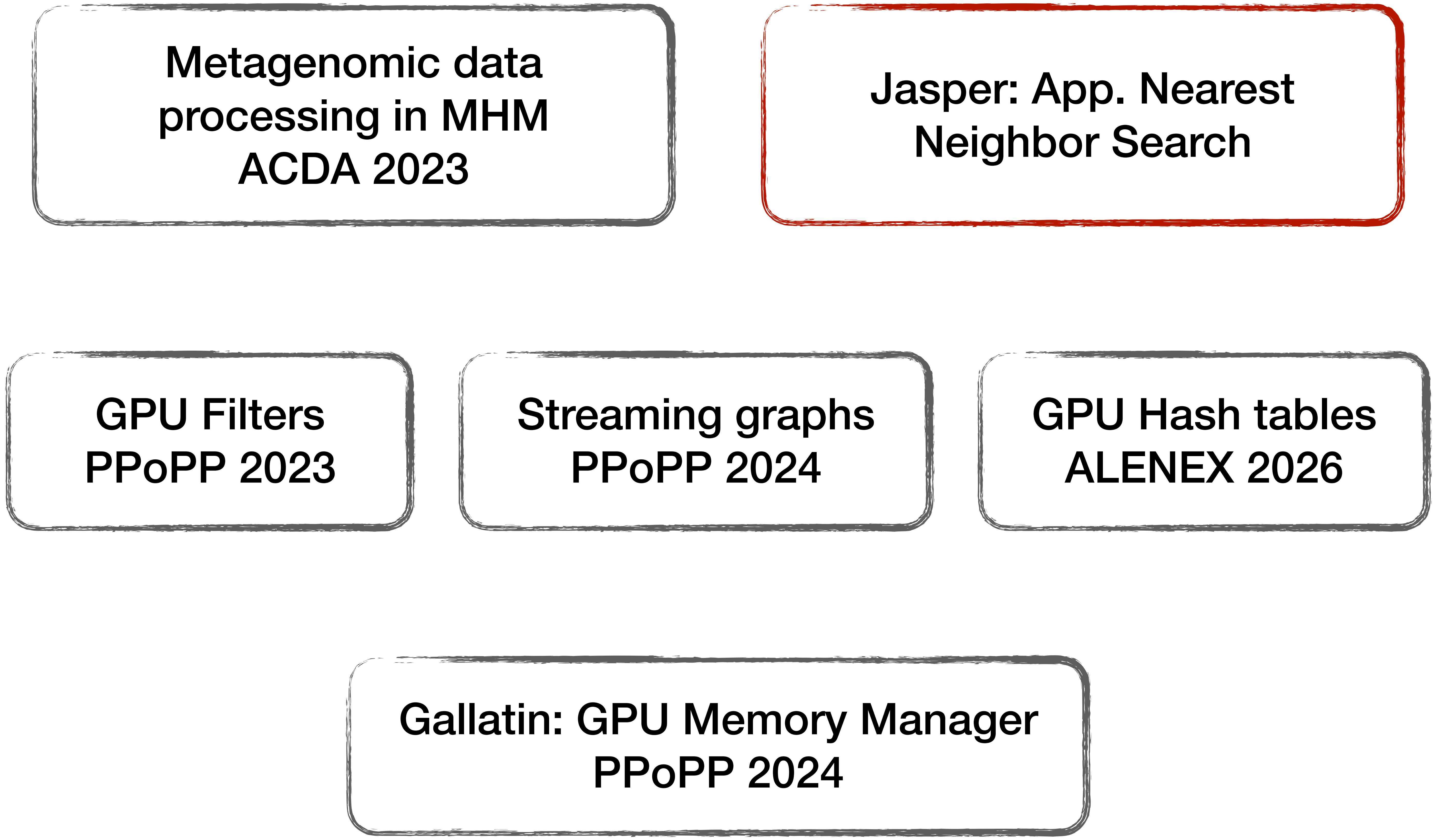
The case for GPU acceleration



GPU

- Applications require high throughput and low latency
- ANNS is compute-intensive and implicitly parallel
- GPUs already deployed for ML inference, colocation reduces data movement

Dynamic data structures and applications on GPUs



Metagenomic data processing in MHM
ACDA 2023

Jasper: App. Nearest Neighbor Search

GPU Filters
PPoPP 2023

Streaming graphs
PPoPP 2024

GPU Hash tables
ALENEX 2026

Gallatin: GPU Memory Manager
PPoPP 2024

https://github.com/saltsystemslab/

Jasper at a glance

- GPU-native ANNS with full updatability
  - GPU-native Vamana[Subramanya et al. 2019] design to maximize throughput
  - Adapted batch-parallel construction from ParlayANN[Manohar et al. 2024]
  - GPU-accelerated quantization based on RaBitQ[Gao et al. 2024]
  - Co-designed quantization, memory access, and kernel structure
- Performance numbers
  - Up to **1.93× higher query throughput** than CAGRA[Ootomo et al. 2024] (state-of-the-art GPU index)
  - **19–131× faster queries** than BANG[Karthik et al. 2025] (previous GPU Vamana implementation)
- Incremental construction numbers
  - An **order of magnitude faster updates** than systems requiring full reconstruction

Three challenges in GPU ANNS

Irregular memory access

Control flow divergence

Memory bound

Current state of the art GPU ANNS systems

- **CAGRA**[Ootomo et al. 2024]**: high performance, no updates**
  - State of the art query throughput via optimized NN-Descent
  - Batch-oriented construction with no incremental update mechanism
- **BANG**[Karthik et al. 2025]**: limited scalability due to Product Quantization, No GPU updates**
  - Uses pre-built Vamana index to query on GPUs
  - Uses Product Quantization (PQ) to save space in GPUs
  - Low GPU compute and memory throughput

The Product Quantization (PQ) problem on GPUs

Small, scattered
codebook lookups

8x read
amplification

GPU memory organized in 32-byte sectors

Throughput strictly worse than unquantized for all PQ sizes tested

Fundamental mismatch between PQ access patterns and GPU architecture

Other GPU ANNS systems

- GANNS (HNSW)[Yu et al. 2022]: Sequential construction dependencies
- SONG[Zhao et al. 2020]: Fixed-rank graphs with different tradeoff profiles
- GTS[Zhu et al. 2024]: Tree-based, requires full reconstruction on updates

None achieve: SOTA performance + efficient construction + streaming updates

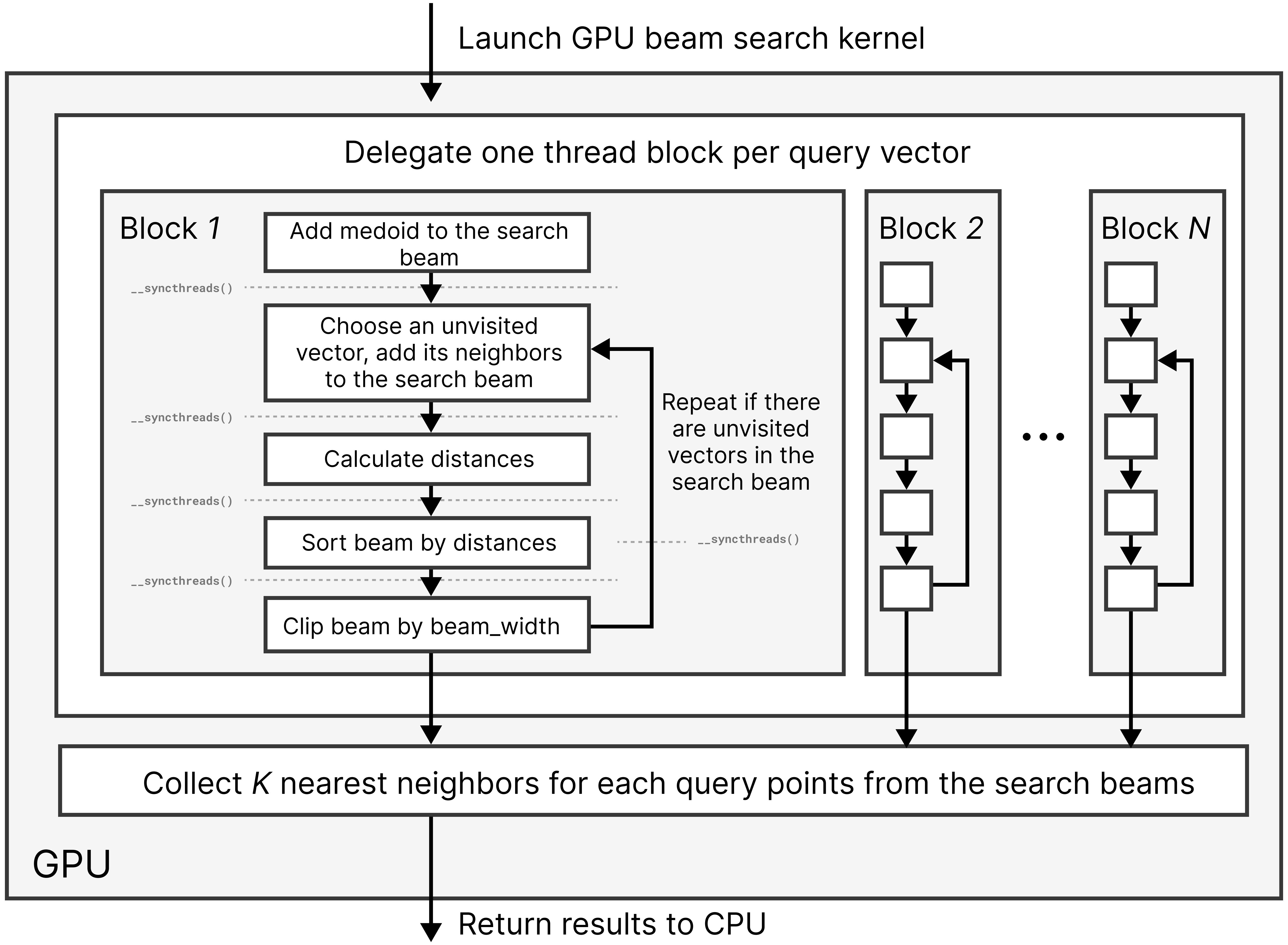
Why Vamana[Subramanya et al. 2019]?

Algorithm	Query performance	Construction	Updatability
HNSW	Very good	Moderate	Yes (slow)
NSG	Very good	Slow (requires k-NN graph first)	No
CAGRA (NN-Descent)	Excellent (GPU)	Fast (GPU)	No
Vamana	Excellent	Fast	Yes (incremental)

Supports incremental insertions via beam search + robust prune [ParlayANN]

Lock-free batch-parallel construction possible

Block-based GPU beam search in Jasper



Launch GPU beam search kernel

Delegate one thread block per query vector

Block *1*

Add medoid to the search beam

__syncthreads()

Choose an unvisited vector, add its neighbors to the search beam

__syncthreads()

Calculate distances

__syncthreads()

Sort beam by distances

__syncthreads()

Clip beam by beam_width

__syncthreads()

Repeat if there are unvisited vectors in the search beam

Block *2*

...

Block *N*

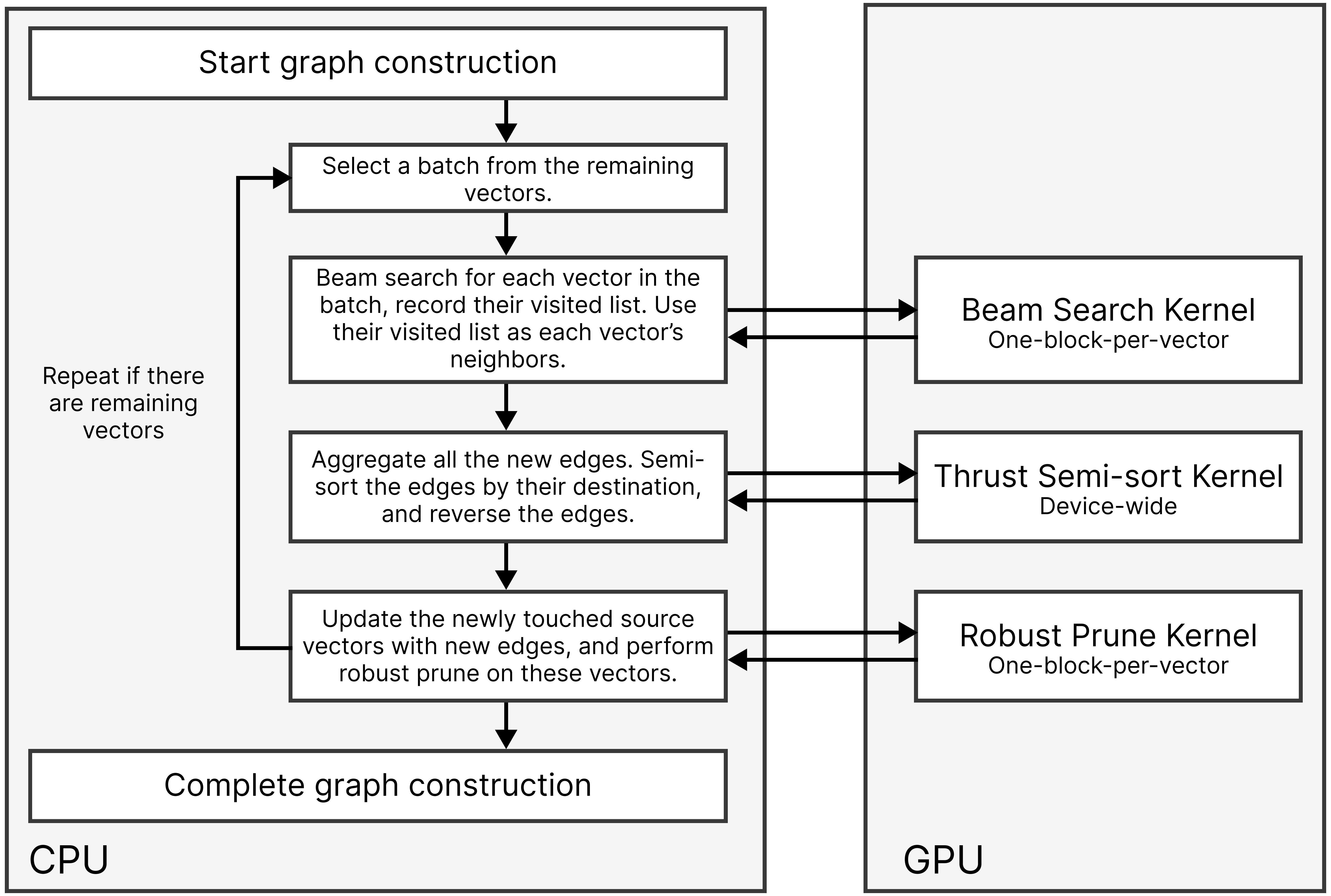
Collect *K* nearest neighbors for each query points from the search beams

GPU

Return results to CPU

- Independent beam searches per vertex (no locks)
- Number of blocks per CTA is based on the dataset dimension

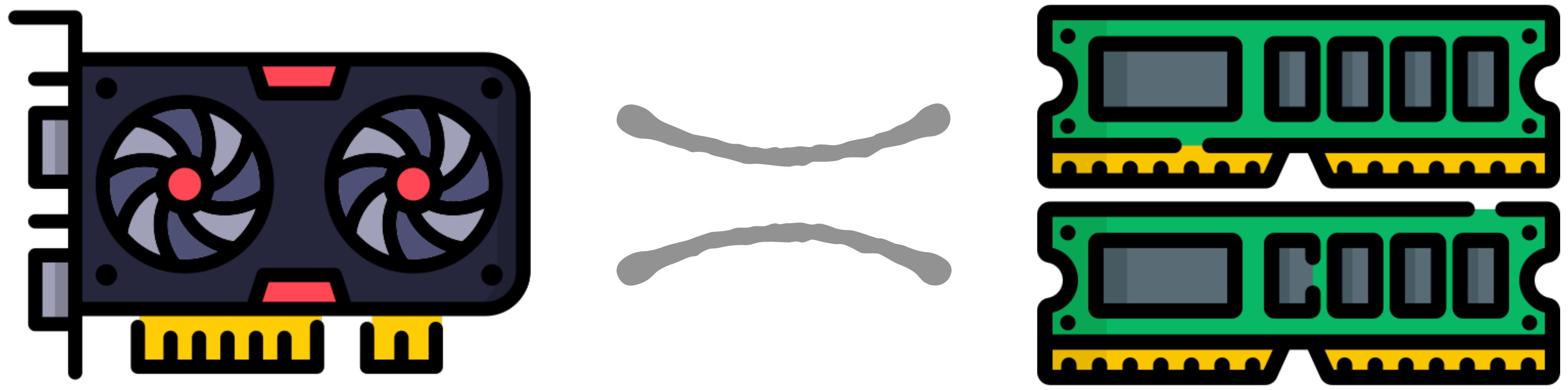
Batch parallel GPU construction in Jasper



Start graph construction
Select a batch from the remaining vectors.
Beam search for each vector in the batch, record their visited list. Use their visited list as each vector's neighbors.
Aggregate all the new edges. Semi-sort the edges by their destination, and reverse the edges.
Update the newly touched source vectors with new edges, and perform robust prune on these vectors.
Complete graph construction
Repeat if there are remaining vectors
CPU
Beam Search Kernel
One-block-per-vector
Thrust Semi-sort Kernel
Device-wide
Robust Prune Kernel
One-block-per-vector
GPU

- Independent beam searches per vertex (no locks)
- Accumulate candidate edges → sort by target vertex → parallel pruning
- Eliminates serialization bottleneck at high-degree vertices (e.g., medoid)

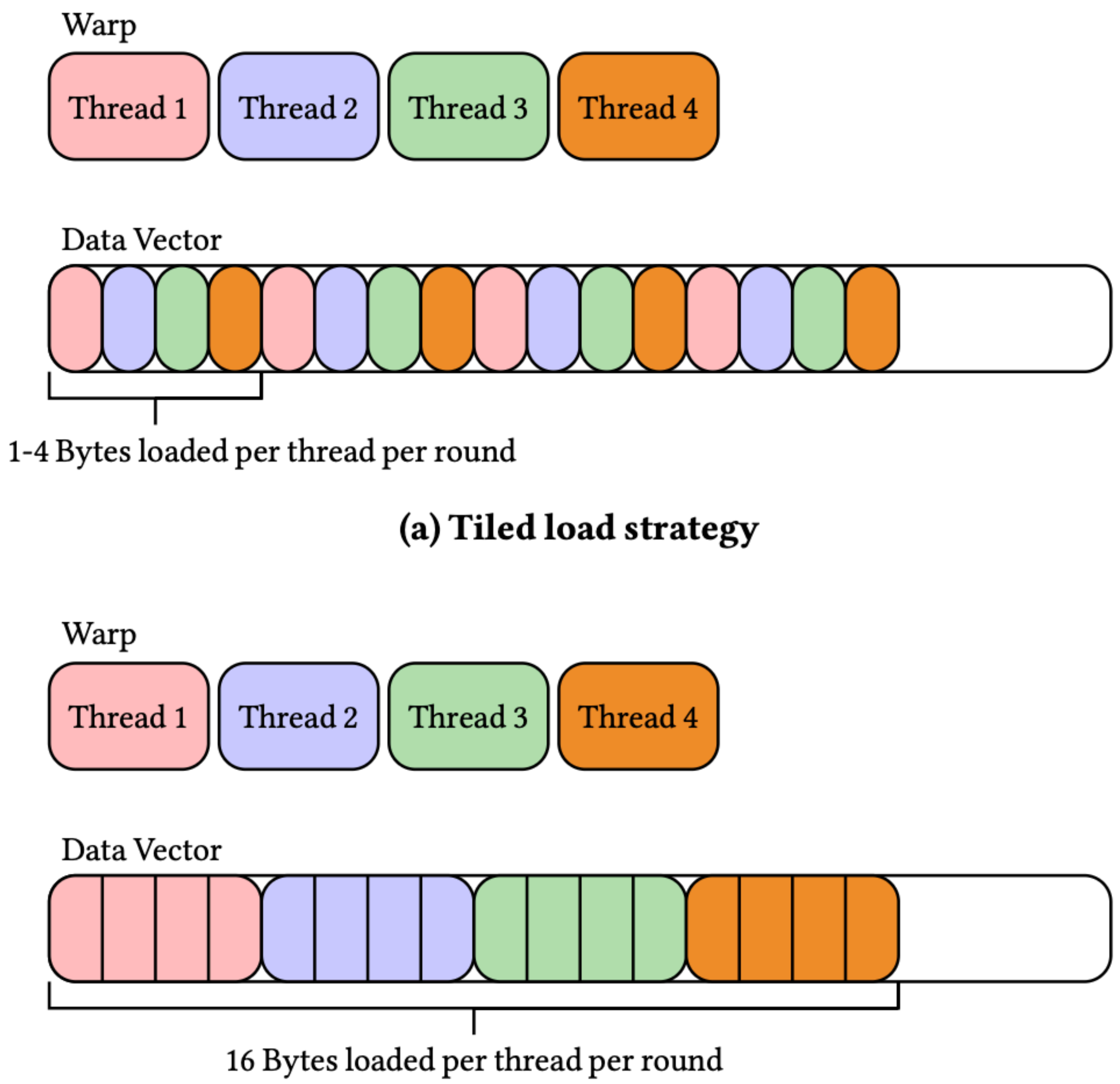
GPU memory bandwidth challenge



- ANNS is memory-bound: each dimension read once, used once, discarded
- E.g., 128-dim float32 vector: 256 FLOPs but 512 bytes loaded

Two strategies: optimize transfer speed AND reduce data volume

Tile-based loading scheme



(a) Tiled load strategy

(b) Chunked load strategy

- Partition vectors into 16-byte chunks
- Each thread loads one chunk from query AND candidate in parallel
- Multiple 16-byte loads issued simultaneously at maximum stride

- 14% improvement at low beam widths (latency-bound regime)
- No throughput penalty at high beam widths

Achieves both low latency AND peak throughput

Block size optimizations: Dataset-dependent tradeoff

- **Smaller blocks** → more queries per CTA → higher memory-level parallelism
- **Larger blocks** → more threads per query → higher compute throughput
- Low-dim vectors (BigANN): memory-parallelism dominates → block size 32 optimal
- High-dim vectors (GIST): compute dominates → block size 128+ optimal

Memory load optimizations enable >80% utilization for both compute and memory

Graph reordering: minimal benefit on GPUs

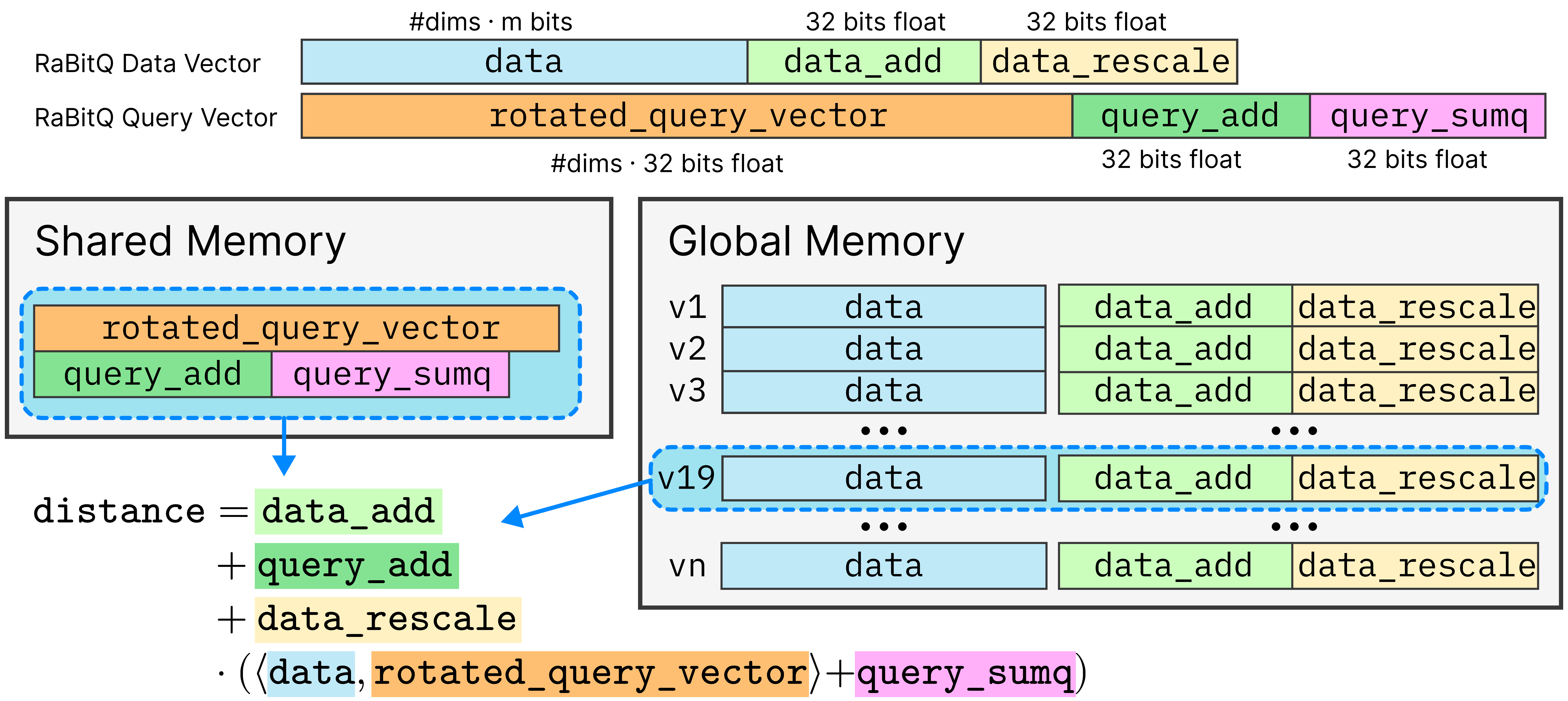
- Contrary to CPU-focused prior work
- When index resides entirely in GPU memory at high utilization, reordering provides negligible benefit
- Cache behavior differs fundamentally between CPU and GPU

RaBitQ [Gao et al. 2024]: quantization without random access

- RaBitQ: Randomized Binarization Quantization
- Randomized rotation + scalar quantization
- Johnson-Lindenstrauss: rotated dimensions cluster tightly around 0
- Encode each dimension with 1-8 bits + rescale factor
- Estimate L2 distance via inner product between quantized data and rotated query

Sequential memory access only—no branching, no random lookups

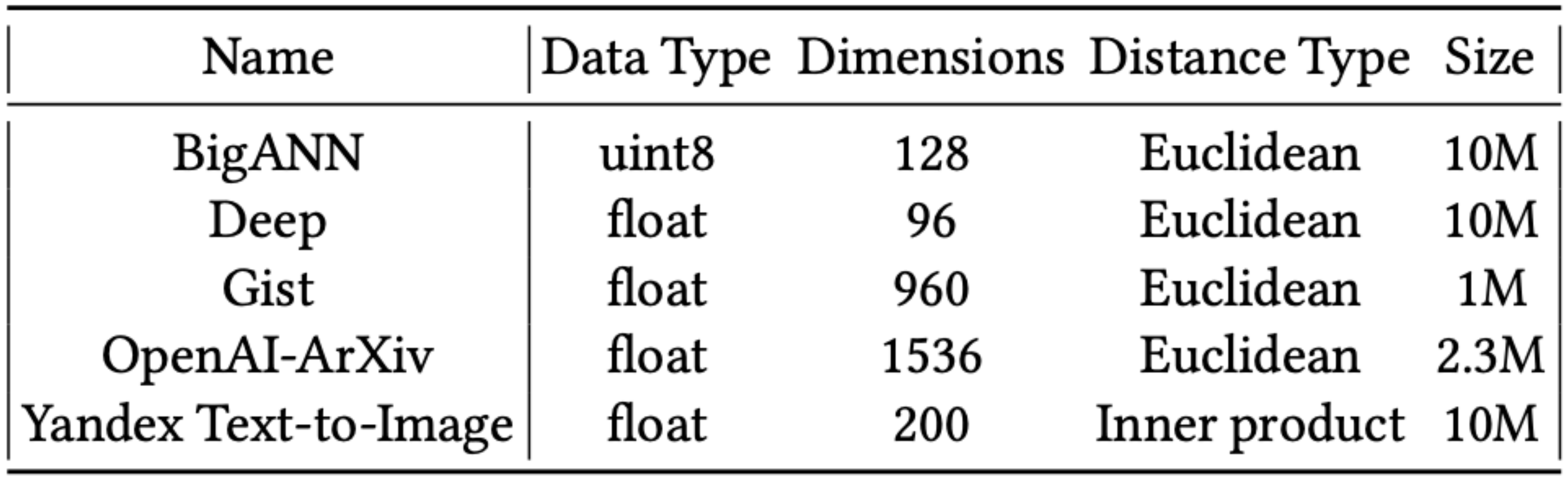
GPU-accelerated RaBitQ in Jasper



$$\text{distance} = \texttt{data\_add} + \texttt{query\_add} + \texttt{data\_rescale} \cdot (\langle \texttt{data}, \texttt{rotated\_query\_vector} \rangle + \texttt{query\_sumq})$$

During distance computations, query vector is stored in shared memory and quantized data vectors are loaded from global

Evaluation: datasets and GPU system

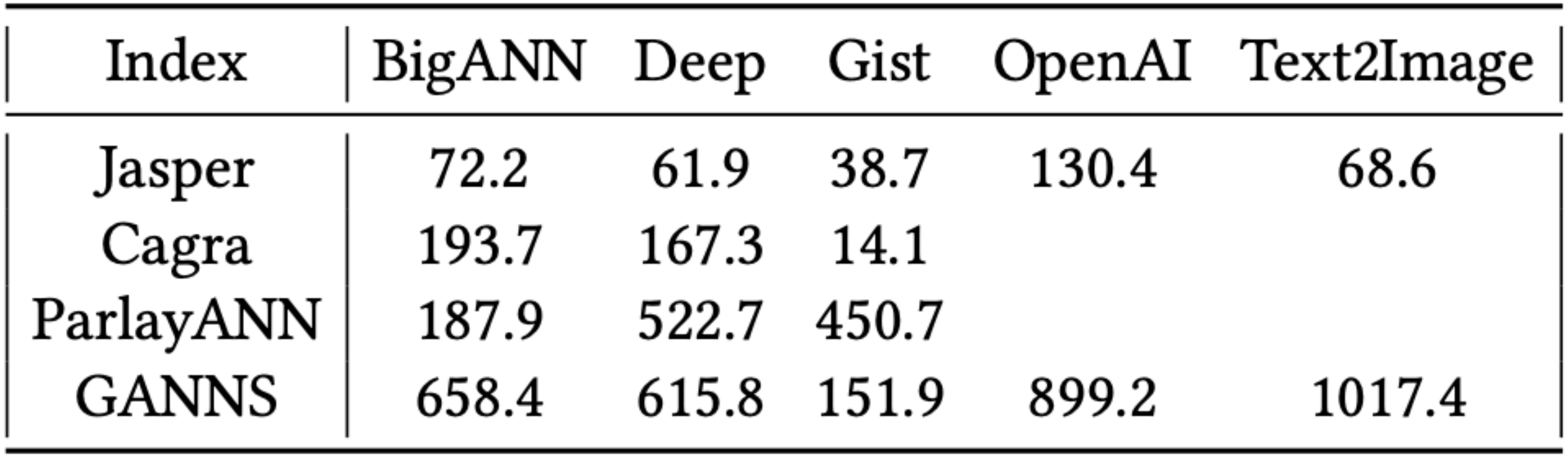


Name	Data Type	Dimensions	Distance Type	Size
BigANN	uint8	128	Euclidean	10M
Deep	float	96	Euclidean	10M
Gist	float	960	Euclidean	1M
OpenAI-ArXiv	float	1536	Euclidean	2.3M
Yandex Text-to-Image	float	200	Inner product	10M

Table 2: A comparison of the different datasets tested against.

GPU: NVIDIA A100 with architecture 80, 80 GB of GPU RAM, and CUDA 12.9

Construction throughput

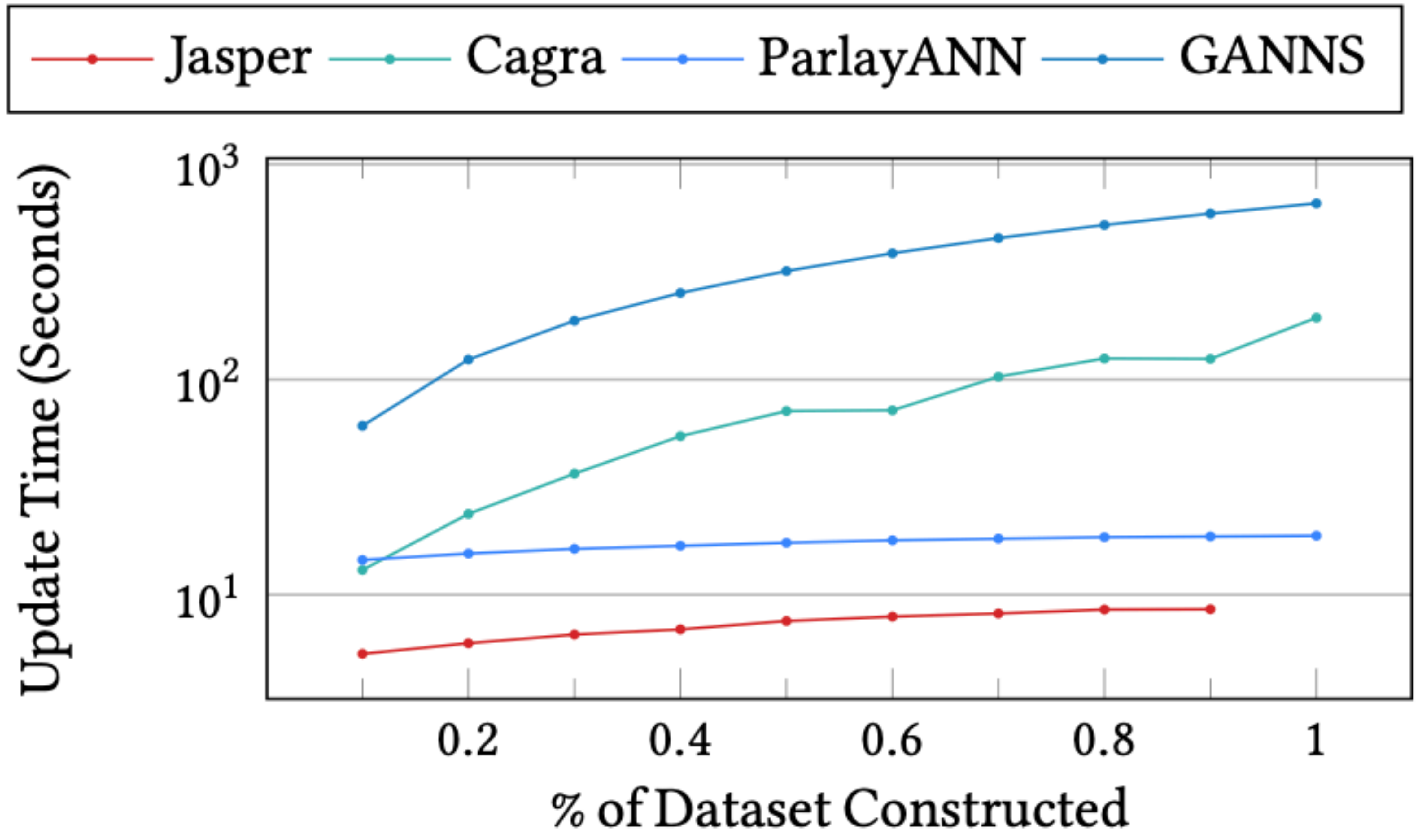


Index	BigANN	Deep	Gist	OpenAI	Text2Image
Jasper	72.2	61.9	38.7	130.4	68.6
Cagra	193.7	167.3	14.1		
ParlayANN	187.9	522.7	450.7		
GANNS	658.4	615.8	151.9	899.2	1017.4

Seconds

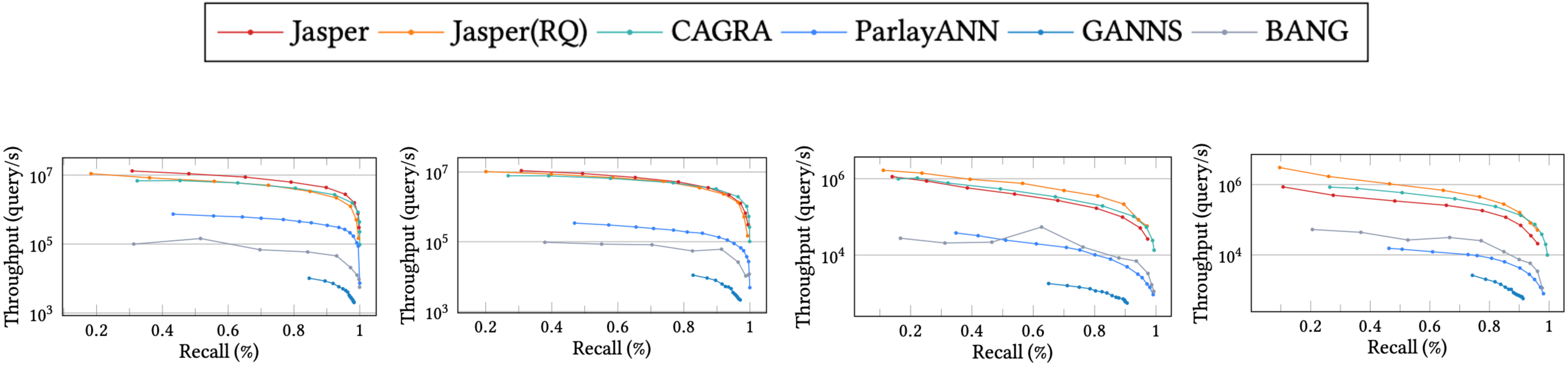
Jasper is fastest for BigANN and Deep even when using exact distance computation!

Incremental construction



Jasper incremental construction is up to 100X faster

Recall vs query throughput



(a) BigANN 10M 1@1 **(d) Deep 10M 1@1** **(g) Gist 1M 1@1** **(j) Openai 2M 1@1**

Jasper exact is faster for low-dimensional datasets, RaBitQ is faster for high-dimensional datasets

Evaluation summary

- BigANN/Deep: Jasper fastest up to ~0.96 recall
- GIST/OpenAI: Jasper RaBitQ fastest across all recall values
- Consistent 19-131× improvement over BANG
- Fastest construction on BigANN and Deep (177K inserts/sec peak)
- Incremental updates: order of magnitude faster than full reconstruction

Key takeaways

Jasper GPU-native ANNS with full updatability

- Piecemeal optimizations leave performance on the table
- Must co-design: quantization scheme + memory access + kernel structure
- Tile-based loading scheme (low latency + high throughput)
- Achieved ~80% utilization of both compute AND memory bandwidth

Hunter McCoy

Zikun Wang

Key takeaways

Jasper GPU-native ANNS with full updatability

- Piecemeal optimizations leave performance on the table
- Must co-design: quantization scheme + memory access + kernel structure
- Tile-based loading scheme (low latency + high throughput)
- Achieved ~80% utilization of both compute AND memory bandwidth

Future work:

- GPU-native algorithm design for ANNS
- Scaling out to host memory for billion-scale datasets
- Scaling to multi-GPU environments

Hunter McCoy

Zikun Wang