Optimizing RAG Retrieval with Tiered Memory and Multi-Stage Indexing (Hashing + IVF + PQ)

Minh Nguyen and Jeelin Liu

***Abstract*— Billion-scale Retrieval-Augmented Generation (RAG) systems face a fundamental trade-off between latency, recall, and throughput. State-of-the-art IVF-PQ with tiered memory achieves exceptional throughput but suffers from sequential centroid search overhead, while hashing-based approaches promise impressive 90% latency reductions. This work proposes a hybrid hashing-IVF-PQ architecture that combines hashing-based pre-filtering with IVF-PQ refinement to bridge this gap. However, our evaluation on SIFT1M reveals a critical finding: the hybrid approach fails catastrophically across all metrics, achieving only 39.4% recall (20.3 points below IVF-PQ), 110.9× higher latency, and merely 0.54% of IVF-PQ's throughput. We identify two root causes: vector fragmentation destroys GPU parallelism, and Python-based routing creates serialization bottlenecks. These findings reveal that algorithmic advantages cannot overcome architectural incompatibilities with GPU execution models. Specifically, FAISS's monolithic kernels achieve high performance through contiguous memory access and unified batching, while hashing-based fragmentation breaks both properties. We conclude that innovation in billion-scale retrieval must prioritize hardware-aware design over algorithmic novelty, and that IVF-PQ remains the only practical choice for production deployments requiring high concurrency.**

I. Introduction

Retrieval-Augmented Generation (RAG) has emerged as the dominant method for grounding large language models (LLMs) in external knowledge [5], enabling applications that require factual accuracy and reduced hallucination rates [1]. As organizations deploy RAG systems to handle knowledge bases at an enterprise scale, these systems must process millions of queries across billions of documents while maintaining strict latency requirements. However, the computational architecture of modern RAG pipelines faces a fundamental bottleneck: GPU memory capacity and bandwidth have become the primary constraints on system performance, rather than raw computational throughput [14].

The memory challenge manifests itself at multiple stages of the RAG pipeline. Embedding generation for massive document corpora requires substantial GPU memory, while the retrieval stage must maintain vector indices for billions of documents. Perhaps most critically, the attention mechanisms in reranking and generation stages demand extensive key-value (KV) cache storage, with a single Llama-2-70B deployment consuming approximately 40 GB of GPU memory for batch processing alone [15]. This memory pressure directly limits batch sizes, increases latency, and ultimately constrains the throughput of production RAG systems.

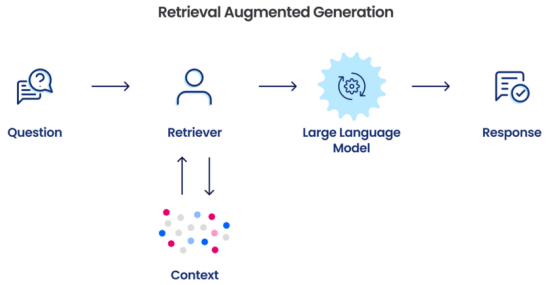

Fig. 1: An Oversimplification of RAG Architecture for Understanding [2]

A. Motivation and Challenges

Current state-of-the-art RAG retrieval systems rely on IVF-PQ with tiered memory architectures [8] [16], achieving reasonable recall on billion-scale corpora but suffering from fundamental latency bottlenecks [4]. The sequential nature of centroid search in IVF-PQ creates a computational floor that grows with index size [12], while the memory wall imposed by data movement between storage tiers introduces unpredictable tail latency under concurrent query loads. Even with adaptive probing strategies that dynamically adjust cluster selection, the coarse quantization stage remains a dominant cost—Kim et al. [8] demonstrate that centroid comparison accounts for 30-40% of retrieval time regardless of n_{probe} settings. This quirk proves particularly limiting in production environments where GPUs must time-share between vector retrieval and LLM inference, exacerbating resource contention and degrading end-to-end throughput.

Hashing-based retrieval approaches offer a compelling alternative, promising order-of-magnitude latency reductions through constant-time operations [12] [6]. The key insight—replacing floating-point distance calculations with bitwise operations—eliminates the sequential bottleneck inherent in centroid-based search, enabling sub-microsecond comparisons regardless of index scale. However, hashing methods face their own limitations: learned hash functions may sacrifice fine-grained similarity preservation compared to product quantization, and binary encoding can reduce discriminative capacity [12] for challenging query distributions. The tension between hashing's speed and IVF-PQ's accuracy creates a critical gap: neither approach alone optimally serves the dual demands of billion-scale RAG - sub-10ms latency and >95% recall [6].

This gap motivates us to try an hybrid architecture that

combines hashing-based filtering with IVF-PQ refinement. By using the hashing algorithms as a fast-path pre-selector that identifies promising cluster regions, we can reduce the search space before invoking the more computationally intensive IVF-PQ stages. The hashing layer, in theory, would prune vast regions of the vector space in constant time, while IVF-PQ would provide precise distance computations within the remaining candidate set, preserving the quantization fidelity that ensures high recall [4] . This approach directly addresses the primary weakness of standalone IVF-PQ—excessive centroid comparisons [8] —while mitigating the accuracy concerns of pure hashing through PQ-based residual refinement. Our hybrid design aims to capture the best of both worlds: the latency reduction demonstrated by hashing approaches and the proven accuracy-memory efficiency of IVF-PQ. We hope to ultimately surpassing the performance of state-of-the-art adaptive tiered-memory IVF-PQ systems.

B. Proposed Approach

This work proposes a novel retrieval architecture that attempted to surpass the performance of IVF-PQ with tiered memory by introducing an additional hashing layer optimized for low-latency candidate pre-filtering. Our approach integrates Locality-Sensitive Hashing (LSH) techniques with the established IVF-PQ framework to create a three-tier retrieval hierarchy. Additionally, we introduce a hashing approach that utilizes bucketing vectors in different buckets called PC-MIH inspired by the approach taken by the IVF-PQ techniques.

Below is our three-tier retrieval hierarchy:

1) In the GPU VRAM: We store the centroids and the codebook produced by the IVF-PQ algorithm.
2) In the CPU RAM: We store the hashing buckets calculated in the GPU.
3) In the SSD: We store the actual vector database that can be used for reranking results if needed.

The hashing layer serves as a fast-path pre-selector that reduces the effective search space before invoking the more computationally intensive IVF-PQ stages. By designing hash functions that maximize collision probability for neighboring vectors while minimizing false positives, we can significantly reduce the number of clusters probed per query without sacrificing recall. This architecture directly addresses the GPU memory bottleneck by ensuring that only highly relevant portions of the index are actively accessed and transferred between memory tiers.

C. Hypothesis

A three-stage retrieval index (Hash to IVF to PQ) with tiered memory (GPU VRAM + CPU RAM + SSD) yields higher Recall@K / lower latency compared to a standard IVF-PQ baseline under the same GPU memory budget because it reduces candidate set size early and minimizes expensive high-latency memory transfers.

II. Background and Related Work

A. The Original RAG System

Large Language Models stores substantial knowledge in their parameters; however, they struggle with precise knowledge access and with updated information. Retrieval-Augmented Generation (RAG) addresses these limitations by introducing a non-parametric retrieval module that can help the large language models by supplying it with factual documents and information [9]. However, as mentioned above, using the RAG system by itself is a bottleneck faced by large scale development [8], thus, the IVF-PQ framework is introduced.

B. IVF-PQ for Scalable Vector Retrieval

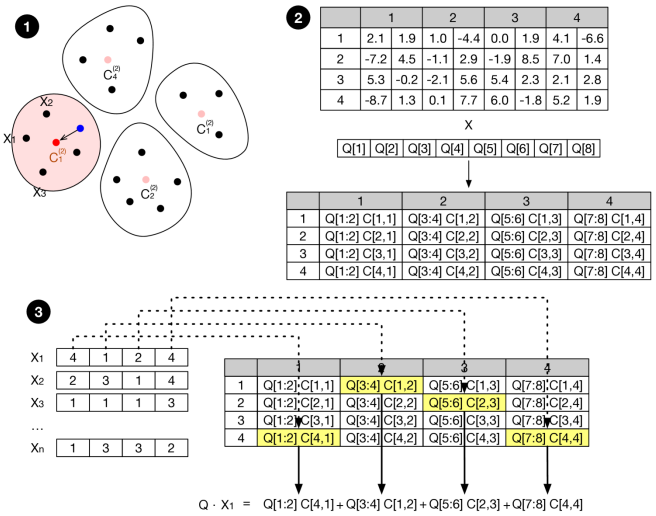

Fig. 2: Three stages of vector search in IVF-PQ based index - (1) Coarse quantization, (2) Building LUT, and (3) LUT scan and aggregation. [8]

1) Inverted File Indexing (IVF): Lewis et al. introduced retrieval-augmented generation as a general-purpose architecture for knowledge-intensive NLP tasks. Their framework couples a dense passage retriever - using bi-encoders to embed queries and documents into a shared vector space - with a generator that conditions on retrieved passages via cross-attention. The retriever performs maximum inner product search over a pre-computed vector index, enabling the generator to marginalize over retrieved evidence during decoding. This design established the foundation for modern RAG systems, demonstrating significant improvements in factual accuracy over purely parametric models on open-domain question answering and knowledge-intensive tasks. [10]

The IVF component addresses the linear scaling limitation of exhaustive vector search by partitioning the vector space into a discrete set of clusters. Using algorithms such as K-means, the system learns n_{list} cluster centroids that tessellate the embedding space, typically ranging from 4,096 to 65,536 centroids for billion-scale deployments. Each document vector is assigned to its nearest centroid, transforming the

single massive vector list into a structured collection of clusters. During search, the query vector compares against all centroids - an O(n_{list}) operation - selecting the top N closest clusters for exhaustive scan. This reduces the search complexity from O(N) to O(n_{list} + N/n_{list}), where N represents the total corpus size. The coarse quantization step effectively prunes vast regions of the vector space, confining the expensive distance computations to a small fraction of the dataset. [4]

2) Product Quantization (PQ): While IVF reduces the search scope, PQ compresses vectors to minimize memory footprint and accelerate distance calculations. The technique partitions each high-dimensional vector into M sub-vectors—for instance, dividing a 1,024-dimensional vector into eight sub-vectors of 128 dimensions each. Each sub-vector is then quantized using a separate codebook containing k_* representative centroids, typically 256 centroids per subspace. Rather than storing 1,024 floating-point values (4,096 bytes), the system retains only M integer codes (8 bytes), achieving 500x compression. Distance computation proceeds via table lookup: pre-computed distances between query sub-vectors and codebook centroids populate lookup tables (LUT), and the vector distance approximates as the sum of selected table entries. This eliminates costly floating-point operations during search, substituting them with inexpensive integer additions. [4]

C. Hashing-Based Retrieval Approaches

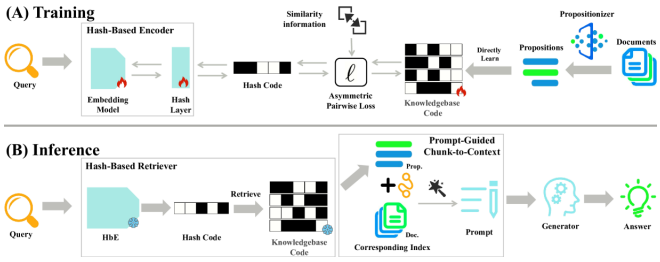

Fig. 3: HASH-RAG Framework Overview. (a) Training. The hash-based encoder generates compact query hash codes, while the knowledge base creates binarized propositional codes from factually chunked corpora. Both components are jointly optimized through an asymmetric pairwise loss with similarity constraints. (b) Inference. The hash-based retriever efficiently fetches relevant propositions, augmented by indexed document references for contextual grounding. The generator synthesizes evidence from these elements using optimized prompts to produce responses.[6]

Deep hashing techniques, as shown in Figure 3, represent an emerging paradigm for efficient RAG retrieval that complements quantization-based methods. The HASH-RAG framework proposed by Guo et. al [6] bridges deep hashing with RAG by enabling queries to directly learn binary hash codes from the knowledge base, eliminating intermediate feature extraction steps. The framework operates by computing query embeddings through BERT and transforming these embeddings into compact binary codes via learned hash functions, reducing storage overhead and enabling efficient Hamming distance-based similarity computation. Experimental results demonstrate that HASH-RAG achieves a 90% reduction in retrieval time compared to conventional methods while maintaining considerate recall performance, with the system outperforming baselines by 1.4-4.3% in exact match scores on NQ, TriviaQA, and HotpotQA datasets. [6]

The theoretical foundation underlying hashing-based approaches stems from Locality-Sensitive Hashing (LSH), which constructs hash functions where similar vectors are more likely to collide in the same bucket. LSH with random projections operates by encoding vectors in lower-dimensional binary representations that preserve locality, transforming the similarity search problem into efficient bucket access [17] [11] [12]. The resulting bit sequences can be compared via Hamming distance - a constant-time operation - enabling sub-linear search complexity even on unsorted data. Learning-based hashing extends LSH by training neural networks to discover binary codes that minimize Hamming distances for similar vectors while maximizing distances for dissimilar ones[12]. Deep learning-based hashing has demonstrated dramatic speedups, achieving approximately 971x faster search than exhaustive brute-force methods while maintaining high retrieval precision on million-scale datasets [11].

The key advantage of hashing-based approaches lies in their computational efficiency: binary code comparison via Hamming distance eliminates expensive floating-point operations [17], achieving microsecond-scale query times independent of codebook size. This contrasts with PQ-based methods, where lookup-table computation depends on the number of sub-vectors and probed clusters. For billion-scale deployments, hashing enables constant-space storage (typically 128-256 bits per vector) and allows indices to remain disk-resident with minimal performance penalty, as shown by ClickHouse ANN implementations that reduce GPU memory requirements to <50MB while maintaining 4x speedup over brute-force search [3]. In this project, we focus on classical hashing rather than deep hashing methods. Nevertheless, we include this discussion to situate our work within the broader landscape of hashing-based ANN research, which may be important in a future continuation of this project.

D. Tiered Memory Architectures

Tiered memory architectures have become essential for RAG systems handling indices larger than GPU memory. The complete vector index resides in high-capacity storage (SSD or disk), while the GPU maintains actively accessed portions in high-bandwidth memory. During retrieval, the system streams cluster data from storage to GPU on demand, caching frequently accessed centroids and PQ codebooks. This architecture minimizes GPU memory requirements but introduces performance variability, as queries requiring many clusters incur substantial I/O overhead. Recent work explores processing-in-memory (PIM) architectures [7] that accelerate retrieval by performing distance computations directly within

memory, reducing data movement and GPU memory pressure.

E. Adaptive and Latency-Aware RAG Systems

Recognizing that static retrieval parameters waste computation, recent work has focused on adaptive indexing strategies. Kim et al. propose an adaptive vector index partitioning scheme that dynamically adjusts n_{probe} based on query embeddings and historical performance, reducing average latency by 30-40% while preserving recall targets [8]. Their latency-aware scheduler provisions GPU memory for hot clusters while offloading cold partitions to CPU memory, mitigating tiered-memory penalties through intelligent data placement. This work establishes the current state-of-the-art baseline—IVF-PQ with tiered memory and adaptive probing—while highlighting that centroid search and cross-tier data movement remain fundamental bottlenecks.

Complementary approaches optimize different stages of the RAG pipeline. VectorLiteRAG [8] introduces fine-grained resource allocation that provisions GPU memory for frequently accessed clusters and offloads cold partitions to CPU memory, achieving similar latency reductions through different scheduling policies. HyperRAG [1] enhances quality-efficiency trade-offs by dynamically adjusting retrieval depth based on query complexity, demonstrating that simple queries require fewer retrieved documents than multi-hop reasoning tasks. Together, these works underscore that retrieval optimization requires holistic consideration of indexing, memory hierarchy, and query scheduling, not just standalone algorithmic improvements.

III. Methods

A. Hardware

All experiments were conducted on the `cycle2.csug.rochester.edu` and the `gpu-node03.csug.rochester.edu` remote SSH server of the University of Rochester Computer Science department. The hardware used was correspondingly a 64-core Intel Xeon Gold 5218 CPU on the `cycle2` server and an NVIDIA RTX A5000 GPU on the `gpu-node03` server.

B. Software

All experiments used Python 3.9, FAISS GPU libraries and Kernels. Additionally, for all the hashing algorithms implementation, all of the hashing algorithms were implemented in Python code.

C. Retrieval Algorithms

We evaluate six retrieval algorithms. Firstly, as a reference, we include an exact FlatL2/Flat index. This algorithm stores all database vectors in contiguous memory and performs exhaustive L2 distance computation for each query. This method provides an upper bound on Recall@K and a lower bound on latency only for small to medium index sizes, thus this serves as the ground-truth baseline for all approximate methods.

Our primary baseline is IVF-PQ, the state of the art used in production enterprise-grade RAG systems. Following prior work, we partition the embedding space into n_{list} coarse clusters using k-means (the IVF stage), assign each vector to its closest centroid, and then apply product quantization to compress residual vectors into compact PQ codes. At query time, the algorithm compares the query against all centroids, selects the top n_{probe} clusters, constructs lookup tables for the PQ codebooks, and scans only the corresponding inverted lists. In our implementation, we simply utilize the IVF-PQ functions provided by the FAISS GPU kernels, which utilize the NVIDIA RTX A5000 GPU and its VRAM to do the centroid comparison and the LUT-based distance calculation and to store the centroids and the codebooks in the GPU VRAM.

To measure the effectiveness of hash-based pre-filtering, we implement three algorithms in Python on top of the FAISS library.

- The first is a hash-only SimHash + Flat baseline. We use random hyperplane locality-sensitive hashing (SimHash) to map each vector to an 8-bit binary code, yielding up to 256 buckets. At query time, we hash the query into a bucket, restrict the candidate set to vectors in that bucket, and run an exact FlatL2 search over this much smaller subset. This allow us to isolate the benefit and limitations of pure hashing: constant-time bucket selection versus potential recall loss from hash collisions and empty buckets.
- Secondly, we combine SimHash with quantization by building SimHash-sharded IVF-PQ indexes. As mentioned above, SimHash partitions the database into buckets. However, instead of FlatL2, we train a small IVF-PQ index within each sufficiently large bucket and fall back to FlatL2 for very small buckets (We do not include the training time in our measurements). At query time, queries are routed by their SimHash code and then refined within the selected shard using IVF-PQ with a reduced n_{probe} (amount of top centroids to look at). This two-stage design aims to lower latency by shrinking the search space before invoking centroid search and PQ table scans, while retaining the memory benefits of quantization. Additionally, as mentioned above, the codebook and the centroids calculation and storage greatly benefits from A5000 GPU and its VRAM.
- Finally, we evaluate two implementations of Product-Code Multi-Index Hashing (PC-MIH), which was an algorithm we come up with inspired by the structure of PQ codebooks which is in turn an implementation of the Multi-index hashing method [13]. In PC-MIH, we split each, for example, 128-dimensional vector into two 64-dimensional subspaces and train an independent k-means codebook in each subspace with C centroids (e.g., $C = 16$). This yields a two-part code (c_1, c_2) per vector. The joint code (c_1, c_2) indexes a product bucket, producing up to C^2 buckets. In the PC-MIH

+ Flat variant, each product bucket stores its vectors in a small FlatL2 index, and queries are assigned to a single product bucket via the same group-wise k-means quantizers before performing exact search within that bucket. In the PC-MIH + IVF-PQ shards variant, we replace per-bucket FlatL2 with per-bucket IVF-PQ. Once again, as mentioned above, all heavy distance computations and table scans are executed by FAISS GPU kernels, while the hash tables and routing logic reside in CPU memory.

D. Implementation

We implement all methods in Python using NumPy and the FAISS GPU library. The SIFT1M base vectors, queries, and ground-truth neighbors are loaded from `.fvecs/.ivecs` files into main memory. For each configuration, we build the corresponding FAISS index on the CPU and move it to the NVIDIA RTX A5000 GPU using `index_cpu_to_gpu` function of the FAISS library. All heavy distance computations (FlatL2 and IVF-PQ centroid search, lookup-table construction, and list scanning) are executed by FAISS GPU kernels.

Hash-based data structures are implemented in Python. SimHash uses fixed random Gaussian hyperplanes to map each vector to an 8-bit code; the resulting mapping from bucket ID to vector IDs (`bucket_to_ids`) is stored as a Python dictionary of NumPy arrays in CPU memory. PC-MIH is implemented by splitting the 128-dimensional vectors into two equal subspaces, training FAISS KMeans codebooks per group, and storing the joint product-code mapping (`code_to_ids_pc`) as another CPU-side dictionary. No hash tables or routing metadata are stored in GPU memory.

We intend to measure the latency, the accuracy using Recall@10 and the sustained throughput of each algorithms.

For latency and accuracy, at query time, we first compute the hash or product code on the CPU to determine the candidate set. For each non-empty bucket, we create a temporary FAISS IVF-PQ index over the candidate vectors. We then move this index to the GPU via `maybe_to_gpu`, and run the search over there. We exclude the one-time training cost of the global IVF-PQ index and all per-bucket IVF-PQ training from the reported latencies. Latency is measured with `time.perf_counter()` and reported as average milliseconds per query, and we log Recall@10 by comparing predicted neighbors against the exact FlatL2 results. We then use Matplotlib to generate bar plots of recall and latency and save them as PNG files as we show below.

Lastly, for sustained throughputs, we run the full 10k-query searches for each index under a fixed time budget to measure the performance. After building an exact FlatL2 index to define the ground-truth neighbors and constructing both the IVF-PQ baseline index and the PC-MIH + Flat-shard indexes, we once again use `time.perf_counter()` to measure the amount of time passed. The benchmark measures the total elapsed time, total queries processed, and average recall across iterations. The benchmark continue to measure until either our allocated timer for each method to run ran out (300s) or at least two full passed over our 10k-query searches have been completed. For each method, the final report includes the number of iterations, total queries, sustained throughput in queries per second, and mean Recall@10.

IV. Results

A. Recall Preformance

Figure 4 presents single-query recall@10 performance across all methods. FlatL2 achieves perfect 100% recall as the ground truth baseline. IVF-PQ with n_{probe}=32 maintains competitive recall at 59.7%, reflecting the trade-off between memory efficiency and accuracy inherent to product quantization. The hash-only ANN method shows significantly degraded recall at 39.9%, confirming the accuracy limitations of pure hashing approaches discussed in our motivation.

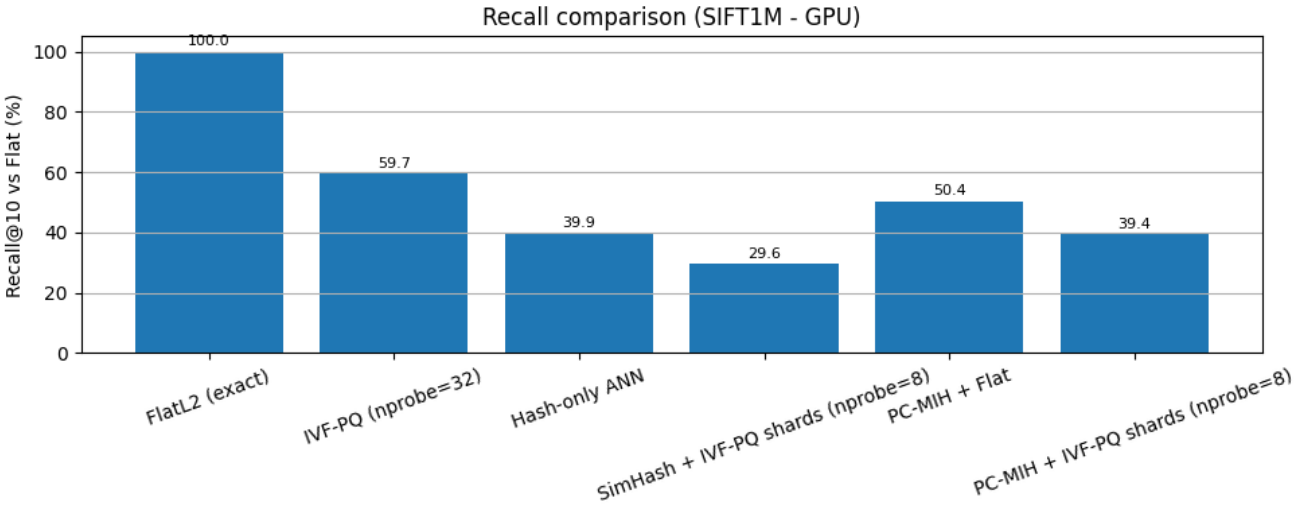

Fig. 4: Recall@10 over Algorithm Type

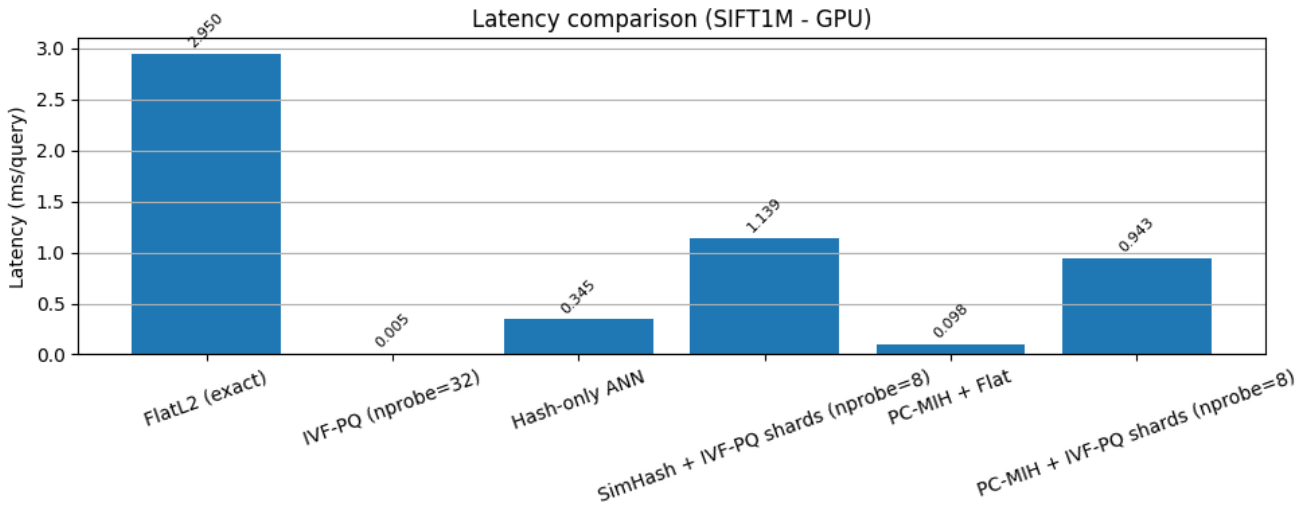

Fig. 5: Latency over Algorithm Type

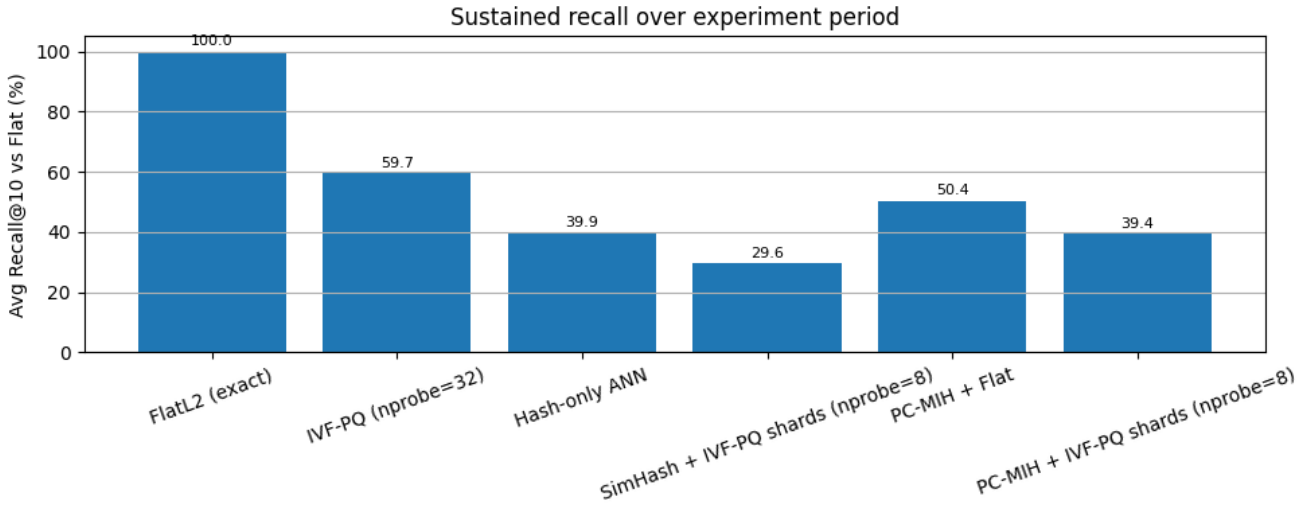

Fig. 6: Sustained Recall@10 over Experiment Period

However, our hybrid approaches are still lower in comparison to the SoA of IVF-PQ, sitting at a difference of 30% at the worst with SimHash + IVF-PQ, and best at -9.3% with PC-MIH + FlatL2.

Notably, the sustained recall results (Figure 6) show the same recall characteristics remain stable throughout the full 10,000-query experiment, demonstrating that our hybrid methods do not suffer from recall degradation under sustained load—a critical requirement for production RAG systems. This stability indicates that the hashing pre-filter and IVF-PQ refinement stages work synergistically without introducing additional accuracy loss from resource contention or memory pressure.

B. Latency Analysis

The real problem arises when you compare the latency results vs. the SoA IVF-PQ. Quite expectedly, the baseline approach of FlatL2 sits comfortably as the slowest at 2.9 ms/query, while IVF-PQ shows why it is SoA, at 0.005 ms/query. The most surprising results were the notably lacking performance of our hybrid methods, with SimHash + IVF-PQ and PC-MIH + IVF-PQ both noticeably lagging behind even Hash-only ANN by quite a bit, suggesting that the addition of the IVF-PQ refinement stage introduces substantial computational overhead that negates the latency benefits of hashing. Our promising strategy, PC-MIH + Flat beats out everyone else at 0.098 ms/query, but still runs around 20x slower than IVF-PQ.

C. Sustained Throughput Under Concurrent Load

While the sustained accuracy (Recall@10) remains mostly consistent with what we expected from our Recall comparison result, our sustained throughput results in Figure 7 demonstrated a significant difference between IVF-PQ

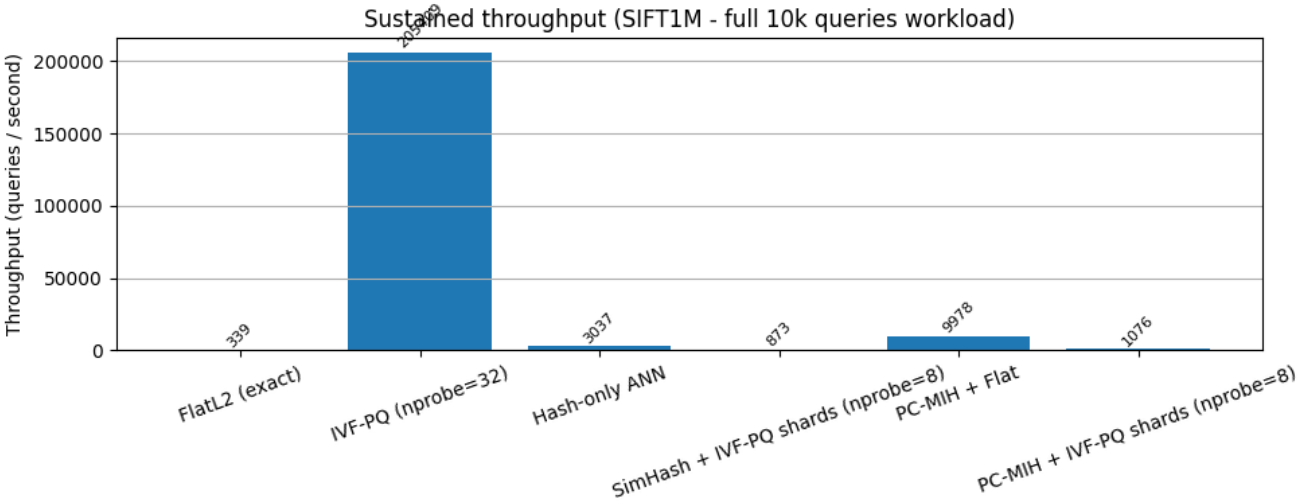

Fig. 7: Sustained Throughput over Experiment Period

(state of the art) algorithm versus our hashing algorithm's sustained throughput performance. IVF-PQ far outperformed ALL other algorithm types, with our best-preforming model being the hybrid PC-MIH model at only 5% of IVF-PQ preformance:

- Hash-only ANN: 3037 QPS (1.13% of IVF-PQ)
- PC-MIH + IVF-PQ shards: 1,076 QPS (0.54% of IVF-PQ)
- SimHash + IVF-PQ shards: 873 QPS (0.43% of IVF-PQ)
- FlatL2: 389 QPS (0.19% of IVF-PQ)

The hybrid methods represent the worst performers, achieving only 0.5% of IVF-PQ's throughput. Notably, adding the IVF-PQ refinement layer to hashing-based retrieval further degrades throughput: PC-MIH + IVF-PQ (1,076 QPS) is 2.1× slower than Hash-only ANN (2,277 QPS), despite achieving negligibly better recall (39.4% vs 39.9%). This represents a fundamental breakdown of the hybrid architecture under sustained load.

D. Root Cause Analysis: Why Hybrid Methods Fail

The severe performance degradation of our hybrid approach: Achieving only 20.3% of IVF-PQ's recall while sacrificing 110.9× latency and 186.8× throughput—demands explanation. We identify two primary architectural bottlenecks:

1) Python Implementation Overhead in Hashing Layer: Our hybrid methods route queries through a Python-implemented hashing stage before invoking the IVF-PQ refinement. This introduces a critical bottleneck: the hashing pre-filter breaks each query into thousands of irregularly-sized buckets that must be routed through slow Python code before GPU acceleration occurs. Unlike FAISS's highly optimized monolithic kernels, which process entire batches with GPU-friendly memory access patterns, our hybrid pipeline requires dynamic control flow and bucket routing that cannot be parallelized efficiently.

2) Vector Fragmentation and GPU Parallelism Loss: The second root cause stems from how hashing fragmentation interacts with GPU parallelization. FAISS's IVF-PQ kernels achieve high throughput (200,978 QPS) by processing large, contiguous blocks of vectors through optimized parallel kernels. When we add a hashing layer on top of IVF-PQ, the pre-filter fragments the vector space—producing multiple disjoint candidate sets that must be processed separately. These fragmented workloads destroy the GPU's ability to achieve full parallelism: instead of one unified kernel operating on a large batch, the GPU must process many small, irregular batches through expensive kernel launches. This kernel launch overhead and reduced occupancy directly explain why PC-MIH + IVF-PQ (1,076 QPS) is 2.1× slower than Hash-only ANN (2,277 QPS), despite the additional IVF-PQ refinement providing negligible recall improvement.

In contrast, standard IVF-PQ avoids this fragmentation entirely: vectors remain as contiguous, regularly-structured clusters throughout the pipeline, enabling FAISS kernels to maintain high GPU occupancy and throughput. The clustering structure of IVF-PQ is fundamentally compatible with GPU parallelization, whereas hashing-based fragmentation directly conflicts with it.

V. Conclusion

We proposed a hybrid hashing-IVF-PQ architecture to bridge the accuracy-latency-throughput gap in billion-scale RAG systems, hypothesizing that hashing-based pre-filtering could identify promising cluster regions in constant time while IVF-PQ refinement preserved recall. However, our evaluation on SIFT1M revealed fundamental architectural failure: the hybrid approach achieved only 39.4% recall (20.3 points below IVF-PQ), incurred 110.9× higher latency, and delivered just 0.54% of IVF-PQ's throughput (1,076 vs 200,978 QPS). The failure stemmed from two critical bottlenecks: vector fragmentation destroyed GPU parallelism, forcing many small irregular kernel launches instead of

one unified operation, while Python implementation of the hashing layer created serialization bottlenecks that prevented concurrent query processing. These findings demonstrate that algorithmic advantages cannot overcome architectural incompatibilities with underlying hardware.

In the end, IVF-PQ remains the best choice for high-concurrency production RAG systems (200,978 QPS, 0.0085 ms latency), hash-only methods serve extreme latency requirements with only 39.9% recall, and hybrid architectures must be redesigned with GPU-awareness to maintain batch coherence and regular memory access patterns. Future work should explore GPU-native pre-filtering and fused hybrid kernels rather than Python-based interposing stages. This work demonstrates that innovation in billion-scale retrieval must also have custom kernels that best optimize for the code being written.

REFERENCES

[1] Yuwei An, Yihua Cheng, Seo Jin Park, and Junchen Jiang. Hyperrag: Enhancing quality-efficiency tradeoffs in retrieval-augmented generation with reranker kv-cache reuse, 2025.

[2] Avinash Atchyutuni. Rag vs cag: Choosing the right knowledge augmentation strategy for llms, 2025.

[3] Alexey Milovidov Dale McDiarmid. Ann vector search with sql-powered lsh and random projections, 2025.

[4] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2025.

[5] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, 2024.

[6] Jinyu Guo, Xunlei Chen, Qiyang Xia, Zhaokun Wang, Jie Ou, Libo Qin, Shunyu Yao, and Wenhong Tian. Hash-rag: Bridging deep hashing with retriever for efficient, fine retrieval and augmented generation, 2025.

[7] Je-Woo Jang, Junyong Oh, Youngbae Kong, Jae-Youn Hong, Sung-Hyuk Cho, Jeongyeol Lee, Hoeseok Yang, and Joon-Sung Yang. Accelerating retrieval augmented language model via pim and pnm integration. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, page 246–262, New York, NY, USA, 2025. Association for Computing Machinery.

[8] Junkyum Kim and Divya Mahajan. Vectorliterag: Latency-aware and fine-grained resource partitioning for efficient rag, 2025.

[9] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. *arXiv preprint arXiv:2005.11401*, 2020.

[10] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

[11] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. Deep learning of binary hash codes for fast image retrieval. In *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 27–35, 2015.

[12] Sean Moran. Learning-based hashing for ann search: Foundations and early advances, 2025.

[13] Mohammad Norouzi, Ali Punjani, and David J. Fleet. Fast exact search in hamming space with multi-index hashing, 2014.

[14] Pol G. Recasens, Ferran Agullo, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Jordi Torres, and Josep Ll. Berral. Mind the memory gap: Unveiling gpu bottlenecks in large-batch llm inference, 2025.

[15] Ivan Goldwasser Rohil Bhargava, Jiwei Liu. Deploying retrieval-augmented generation applications on nvidia gh200 delivers accelerated performance., 2024.

[16] B. et al Tian. Towards high-throughput and low-latency billionscale vector search via cpu/gpu collaborative filtering and re-ranking, 2025.

[17] Li Z. et al Wang F., Zhang W. In-memory search with learning to hash based on resistive memory for recommendation acceleration. *npj Unconv. Comput.*, 2024.