GPU-ACCELERATED RETRIEVAL AND SCALABLE MODEL SERVING

Anonymous authors
Paper under double-blind review

ABSTRACT

This report addresses GPU-based information retrieval, focusing on distance metrics and top-K retrieval for large-scale data (Task 1), along with a queue-and-batch serving design for handling real-time inference requests (Task 2). By accelerating retrieval kernels on GPUs, we significantly reduce per-query latency. In parallel, we adopt systematic request queueing and batching to smooth bursty traffic, increase throughput, and maintain stable average and tail latencies. Our experiments show that the synergy between low-level GPU optimizations and high-level orchestration yields robust performance improvements for real-world ML pipelines. We highlight challenges, potential solutions, and future directions for scaling these methods in production environments.

1 INFORMATION RETRIEVAL ON GPU (TASK 1)

1.1 MOTIVATION

Information Retrieval (IR) systems aim to efficiently locate and deliver relevant data in response to user queries through three core stages: indexing, query processing, and ranking/retrieval Baeza-Yates & Ribeiro-Neto (2011). The indexing stage structures data for rapid access using techniques including inverted indices for text searches Zobel & Moffat (2006), spatial indices like k-d trees for multidimensional data Bentley (1975), and clustering methods such as K-Means for approximate searches. Large-scale IR in machine learning systems face significant computational challenges due to the need to process high-dimensional data (e.g., millions of vectors with thousands of dimensions), where traditional CPU-based approaches become infeasible because of the quadratic complexity of exhaustive distance calculations.

Clustering enhances IR systems by organizing data into semantically coherent groups, reducing computational complexity. This approach allows dynamic pruning of irrelevant clusters, valuable for real-time applications like recommendation systems. Modern GPU architectures address clustering bottlenecks through three key mechanisms: parallelism that enables thousands of concurrent threads for distance computations, optimized memory hierarchies featuring coalesced memory access and shared memory utilization, and scalable processing of high-dimensional data through batched operations. Studies demonstrate GPU-accelerated clustering achieves 20–50× speedups over CPU implementations for large-scale IR tasks, though memory management remains essential for optimal performance Johnson et al. (2021a).In large-scale recommendation systems like those used by e-commerce or streaming platforms, efficient retrieval of similar user profiles or products from millions of high-dimensional vectors is critical. By combining clustering with GPU-accelerated search, these systems can deliver real-time suggestions with significantly reduced latency and computational cost(Elkahky et al. (2015)).

1.2 OVERVIEW: SYSTEM ARCHITECTURE

The Fig 1 1 illustrates a GPU-accelerated clustering-based IR pipeline optimized for large-scale, high-dimensional data. The workflow begins with L2-normalization for consistent similarity measurement. GPU-accelerated K-Means then clusters the data through parallelized distance calculations and iterative centroid updates. During retrieval, queries first identify relevant clusters before performing either exact K-Nearest Neighbor(KNN) or Approximate Nearest Neighbor(ANN) search, with all computations optimized for GPU parallelism and memory efficiency.

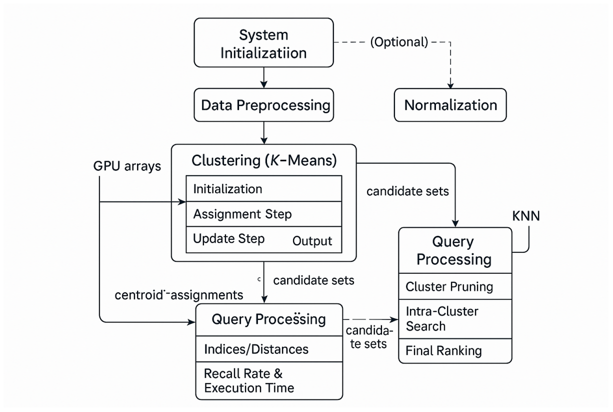

Figure 1: The system workflow.

Key Design Principles:

GPU Acceleration for Computational Efficiency CuPy for GPU-optimized operations including parallel distance calculations and centroid updates, achieving significant speedups over CPU implementations for large-scale datasets. (Johnson et al. (2021c)).

ANN-KMeans Hierarchical Retrieval: Integrates approximate nearest neighbor search with K-Means clustering by first pruning the search space through centroid comparisons, then performing exact distance calculations only within relevant clusters, optimizing the trade-off between retrieval accuracy and computational efficiency. (Zhang et al. (2019b)).

Multi-Distance Metric Support: Implements four complementary distance functions (L2, cosine, dot product, Manhattan) to handle diverse retrieval scenarios, with GPU-accelerated computations ensuring scalability across high-dimensional data. (Mikolov et al. (2013)).

1.3 Distance Functions

In large-scale IR tasks, we implemented four core distance functions—Euclidean (L2), Manhattan (L1), cosine, and dot product—using CuPy to enable efficient, GPU-accelerated vectorized computations.

Problem Definition: Distance functions quantify similarity or dissimilarity between data points in information retrieval tasks. As dataset sizes and dimensionality grow, computing these measures becomes prohibitively slow on CPU-based systems, creating a scalability bottleneck. GPU acceleration addresses this by using massive parallelism to speed up distance calculations for efficient large-scale vector similarity computations.

Design Choice: Our approach addresses this challenge by using GPU acceleration to implement four fundamental distance metrics: dot product, cosine distance, squared Euclidean (L2) distance, and Manhattan (L1) distance. The vectorized GPU operations enable efficient parallel processing and eliminating the overhead of traditional CPU-based methods. All the functions are expressed below using two vectors x and y.

Dot Product:

$$\text{Dot Product}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{D} x_i \cdot y_i$$

The dot product quantifies the degree of alignment between two vectors. Its low computational cost makes it ideal for preliminary filtering in tasks like recommendation systems. In high-dimensional settings, it enables fast approximation of similarity before applying more refined distance metrics.

Cosine Distance:

$$\text{cosine\_sim}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

Cosine distance evaluates the angular difference between vectors by normalizing them and computing the inverse of cosine similarity. It is particularly effective in scenarios like semantic search or text embedding comparisons, where orientation is more informative than magnitude. This metric helps mitigate the curse of dimensionality by focusing on direction rather than scale.

Squared Euclidean (L2) Distance:

$$\text{L2\_distance}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{D} (x_i - y_i)^2$$

L2 distance measures the geometric similarity between vectors by summing squared component-wise differences. It is well-suited for clustering tasks aimed at minimizing intra-cluster variance due to its sensitivity to both magnitude and direction. Its efficiency and interpretability make it a common choice for image analysis and pixel-level comparisons.

Manhattan (L1) Distance:

$$\text{Manhattan Distance}(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{D} |x_i - y_i|$$

Manhattan distance calculates the total absolute difference across vector dimensions, capturing cumulative dissimilarity. It is more robust to outliers than L2 distance and performs well in sparse or anomaly-prone datasets. This metric is widely used in grid-based systems and fraud detection, where stability across features is essential.

Technical Implementation : The distance functions mentioned above perform element-wise operations entirely on the GPU, thereby reducing computational overhead compared to CPU-based methods. For low-dimensional data ($D = 2$), our GPU implementation achieves a speedup of approximately 2–3×, whereas for very high-dimensional data ($D = 2^{15}$), parallelism of GPUs yielded a speedup of 10–20× over CPU implementations. We addressed challenges such as managing GPU memory and ensuring numerical stability during operations like vector normalization by converting data to `float32` and employing efficient batch processing to maintain all computations on-device. Our approach using `CuPy` achieves scalable and efficient distance computations.

1.4 TOP-K RETRIEVAL

Problem Definition: In KNN search, we find K points closest to a query point X from dataset A using a distance metric. To improve efficiency, we implement ANN with clustering to partition data and restrict searches to relevant clusters, effectively leveraging GPU parallelism for handling millions of points. **Design Choice:** To address scalability, our design adopts a GPU-accelerated, clustering-based ANN approach that first partitions data using K-Means, then performs a two-stage search—selecting relevant clusters via centroids and searching within them. This strategy minimizes distance computations and CPU-GPU transfers, enabling high throughput with tunable speed-accuracy trade-offs. By narrowing search scope and leveraging parallelism, the system achieves significant speedups with minimal loss in recall.

Implementation:

KNN: We ran our GPU-based KNN function and measured its execution time. As a baseline, we ran an exact KNN on the CPU (using NumPy for distance computations) under the same conditions. We then compared the results. Tests were repeated across different vector dimensions and dataset sizes to evaluate scalability.

K-Means: The K-Means implementation is fully CuPy-based, performing all computations directly on the GPU for maximum efficiency. It handles edge cases like empty clusters through re-initialization and returns the final results to the CPU using `.get()`. This design ensures scalability while maintaining flexibility for extensions such as cosine-based clustering or performance benchmarking.

Approximate Nearest Neighbors (ANN): Clustering-Based ANN is implemented, offering trade-offs between speed and accuracy. In this approach, the dataset is first partitioned using GPU-accelerated K-Means, forming a coarse-grained index. During a query, the input vector is compared to all cluster centroids, and the top-K_1 nearest clusters are selected based on the distance metric selected. All data points within these selected clusters are then gathered as candidate neighbors. The system computes exact distances between the query and these candidates on the GPU. This localized search reduces the computational overhead compared to a brute-force search.In addition to the clustering-based approach, an additional implementation of **HNSW** (Hierarchical Navigable Small World) and **IVFPQ** (Inverted File with Product Quantization) is also included.

Evaluation: Referring the table 1, At low dimensionality ($D = 2$), the GPU showed no speed advantage and was significantly slower due to overhead, taking approximately 11.9 seconds compared to 1.06 seconds on the CPU. However, at higher dimensionality ($D = 1024$), the GPU completed an L2 ANN query in about 1.72 seconds versus 1.79 seconds on the CPU. This indicates that the crossover point for GPU efficiency occurs as vector dimensionality increases, with GPU performance becoming competitive or superior from around $D = 1024$.

2 Model Serving Systems (Task 2)

2.1 Motivation

Modern machine learning (ML) applications, including recommendation engines and retrieval-augmented generation (RAG) systems, often handle vast numbers of inference requests under stringent latency requirements (Hazelwood et al., 2018; Crankshaw et al., 2017). Even moderate increases in queuing or processing delay can degrade user experiences, while poor resource management risks leaving GPUs underutilized or causing CPU bottlenecks (Wu et al., 2022; Ali et al., 2022). These problems become especially severe when traffic arrives in bursts, making it difficult to maintain consistent performance (Yu et al., 2021). Such challenges are further amplified by the complexity of large models, which require substantial compute and memory resources.

In practice, frameworks like TensorFlow Serving (Developers, 2022), TorchServe (PyTorch Contributors, 2025), and NVIDIA Triton (Morales-Hernández et al., 2021) employ dynamic batching and request queueing to manage large-scale or bursty workloads more effectively. By gathering multiple requests into a single forward pass on the GPU, batching reduces overhead and harnesses the parallelism of modern accelerators. Meanwhile, systematic request queueing prevents momentary load spikes from overwhelming the system. Together, these strategies have been shown to mitigate both average and tail latencies, improving cost efficiency and scalability in production-grade ML services (Yu et al., 2021). The approach explored here similarly adopts queueing and batching to reduce resource waste, stabilize throughput, and meet the real-time needs of contemporary machine learning pipelines.

2.2 Overview

Our system is designed around a multi-stage workflow that integrates retrieval and large-scale inference, reflecting practical setups in modern ML-serving pipelines. At a high level, incoming requests, often user queries, are first placed into a centralized request queue. This queue decouples request arrival from processing, thereby preventing transient spikes in traffic from immediately overloading the system. Once requests accumulate, a background process forms a batch of requests, aiming to achieve favorable utilization by submitting them together to the GPU in a single forward pass.

In our implementation, each request triggers an initial retrieval step that identifies relevant documents or embeddings, a step often used in retrieval-augmented generation. The system then constructs a prompt or input suitable for the chosen large language model. After batched inference completes, responses are split and mapped back to their respective request IDs. This design minimizes overhead per request by reusing shared resources, including GPU kernels and memory, across multiple requests simultaneously. In addition, developers can configure the maximum batch size and the waiting period for building a batch, striking a balance between lower latency for small numbers of incoming requests and higher throughput during bursts.

2.2.1 Request Handling and Processing Workflow

Figure 2 illustrates the overall architecture (how the process inference requests are handled), with the core steps as follows:

1. **Request Arrival**: Clients send inference requests at varying RPS (requests per second) rates, simulating real-world traffic and load patterns.
2. **Request Queue**: Incoming requests are buffered in a queue to smooth out short-term spikes, avoid immediate overload and maintain order.
3. **Batching Module** : Requests from the queue are grouped into batches using a fixed batch size only. This allows for optimized utilization of GPU resources.
4. **RAG Inference Execution**: For each batch of requests, the system retrieves relevant documents via their precomputed embeddings, constructs a prompt, and uses a text-generation model to produce the final answer. These batched requests are processed in a single forward pass on the GPU, leveraging hardware parallelism.
5. **Output Mapping**: The responses are matched with the original request IDs and returned to the respective clients.

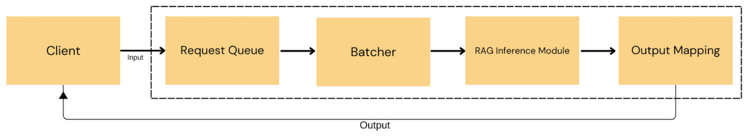

Figure 2: Workflow Diagram: Request Queue and Batch Processing Pipeline

The key advantage of this pipeline is that **queueing** helps handle bursty workloads and prevents overload, while **batching** maximizes GPU throughput by grouping multiple requests together.

2.3 System Measurement

We conducted performance tests by issuing concurrent HTTP requests at controlled rates, using a Python-based load generator that varied the arrival patterns (uniform, bursty/Poisson, random) in order to emulate realistic traffic. This allowed us to observe how the system behaves under steady conditions as well as sudden spikes. To ensure comprehensive evaluation, we measured four primary metrics: **average latency**, **tail latency** (95th or 99th percentile), **throughput** (requests served per second), and **error rate**. These metrics were chosen because they reflect the system's ability to handle high loads while remaining responsive, especially when large models and bursty workloads are involved.

Concretely, the load generator logged timestamps, response codes, and response times for every request, enabling us to compute average and percentile-based latency. We captured throughput by counting how many valid responses were returned per second, and we recorded error rates based on any failed or dropped requests. Analyzing these logs provided insight into how different configurations (baseline, queue-only, queue abd batcher) scaled with increasing request-per-second (RPS) loads. Our rationale for focusing on these metrics is that real-time services cannot simply maximize throughput without also guaranteeing low average and tail latencies, and error-free performance is central to system reliability.

To collect repeatable and detailed data, the load testing script ran trials at various combinations RPS levels and load patterns for each implementation, generating summary statistics after each run. These summaries highlighted how queueing prevented the system from becoming overloaded under bursty conditions, although it introduced minor delays by buffering requests. Meanwhile, batching leveraged the GPU's parallel capacity, boosting throughput at higher loads but occasionally increasing wait times for individual requests. We faced practical challenges arising from limited hardware resources and the overhead of running both the load generator and model inference on the same machine. Consequently, we prioritized moderate RPS scenarios and shorter test windows, acknowledging that more extensive stress-testing remains desirable future work.

2.4 Design: Request Queue and Batcher

The rationale for integrating a request queue and batcher is rooted in the need to manage high variability in request arrival while ensuring efficient use of GPU resources. Without a queue, sudden spikes can overwhelm the system, cause request failures, and leave some hardware idle under lighter traffic. By buffering requests in a queue, the system can dispatch them more systematically, smoothing out bursts. A batcher then aggregates several queued requests into a single GPU pass, reducing per-request overhead such as data transfers and kernel launches. This approach balances immediate responsiveness against the performance gains from larger batches; system administrators can tune parameters like maximum batch size and wait time to align with specific latency or throughput goals.

Implementation involves a central queue where each incoming request is time-stamped and enqueued. A background process continuously checks the queue, forming a batch when either a size threshold is reached or a short timeout elapses. The system then packs all items in the batch into a consolidated input for the model and runs one forward pass on the GPU. To evaluate these design choices, we conducted experiments measuring throughput, latency, and error rates under different traffic patterns. Our analysis showed that batching significantly boosts throughput, especially under moderate to high load, although early-arriving requests in each batch can wait slightly longer. Overall, queuing prevented overload by deferring arrivals rather than dropping requests, while batching minimized wasted GPU cycles by amortizing overhead across multiple queries. Nevertheless, limited GPU memory, unpredictable arrival bursts, and the overhead of run-time scheduling posed challenges. Solutions included setting a sensible batch size, implementing adaptive batch timeouts, and monitoring queue lengths to detect overload conditions. Full details of these experiments, along with additional metric breakdowns and extended discussion of challenges, can be found in Appendix B.4.

2.5 Open Discussions

This section reflects on the broader implications of our design and experiments, focusing on hardware constraints, traffic variability, and potential extensions. We tested multiple arrival patterns (uniform, Poisson, random) across different RPS levels, though not exhaustively, limiting definitive comparisons. Each configuration was run only once, so outliers could skew results, and running the load generator and inference on the same GTX 1060 (6 GB VRAM) likely inflated latencies. Although queueing and batching improved throughput in moderate-load scenarios, higher RPS saturated the GPU quickly, leaving absolute latencies higher than anticipated. In real-world applications, more advanced mechanisms may be necessary, particularly when traffic is highly bursty or model sizes exceed GPU memory constraints. Further details, including future directions such as dynamic batching, multi-threaded CPU preprocessing, and hybrid queue approaches, appear in Appendix B.5.

3 Discussion of Task Synergy

Although raw GPU acceleration (Task 1) lowers the cost of individual retrieval or distance-computation kernels (Taipalus, 2024; Zhang et al., 2019a), these gains can be lost if the system framework (Task 2) issues inefficient kernel launches due to poor batching or queuing. In real-world pipelines, a well-optimized kernel alone does not guarantee overall throughput if frequent, small-batch calls overwhelm the GPU with overhead (Crankshaw et al., 2017; Tillet et al., 2019). Conversely, faster kernel routines allow Task 2 to form larger batches without causing prohibitive latency. Hence, the two tasks are interdependent: Task 1 focuses on optimizing top-K searches and distance metrics for efficiency (Douze et al., 2024; Malkov & Yashunin, 2020; Johnson et al., 2021b), while Task 2 ensures that the parallel advantages of GPU kernels are fully realized through intelligent request orchestration. Aligning these layers yields a system that combines low-level performance with high-level scalability, necessary for robust, production-scale ML services (Li et al., 2020; Zhang et al., 2019a).

REFERENCES

Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Optimizing inference serving on serverless platforms. *Proc. VLDB Endow.*, 15(10):2071–2084, June 2022. ISSN 2150-8097. doi: 10.14778/3547305.3547313. URL `https://doi.org/10.14778/3547305.3547313`.

Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 2011. URL `https://www.pearson.com/en-us/subject-catalog/p/modern-information-retrieval/P200000000377/9780130353009`.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. URL `https://dl.acm.org/doi/10.1145/361002.361007`.

Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: a low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pp. 613–627, USA, 2017. USENIX Association. ISBN 9781931971379.

TensorFlow Developers. Tensorflow. *Zenodo*, 2022.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. A multi-view deep learning approach for cross domain user modeling. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, pp. 278–288. ACM, 2015. URL `https://dl.acm.org/doi/10.1145/2736277.2741667`.

Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 620–629, 2018. doi: 10.1109/HPCA.2018.00059.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2021a. doi: 10.1109/TBDATA.2021.3073628. URL `https://ieeexplore.ieee.org/document/9401509`.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021b. doi: 10.1109/TBDATA.2019.2921572.

Jeff Johnson et al. Gpu speedup benchmarks. *IEEE Transactions on Big Data*, 2021c. doi: 10.1109/TBDATA.2021.3073628. URL `https://ieeexplore.ieee.org/document/9401509`. Accessed 2024.

Yang Li, Zhenhua Han, Quanlu Zhang, Zhenhua Li, and Haisheng Tan. Automating cloud deployment for deep learning inference of real-time online services. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pp. 1668–1677, 2020. doi: 10.1109/INFOCOM41043.2020.9155267.

Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020. doi: 10.1109/TPAMI.2018.2889473.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. URL `https://arxiv.org/abs/1301.3781`.

Mario Morales-Hernández, Md B Sharif, A Kalyanapu, Sheikh K Ghafoor, Tigstu T Dullo, Sudershan Gangrade, S-C Kao, Matthew R Norman, and Katherine J Evans. Triton: A multi-gpu open source 2d hydrodynamic flood model. *Environmental Modelling & Software*, 141:105034, 2021.

PyTorch Contributors. TorchServe: Serve, optimize and scale PyTorch models in production. https://github.com/pytorch/serve, 2025. Accessed: 2025-04-13.

Toni Taipalus. Vector database management systems: Fundamental concepts, use-cases, and current challenges. *Cognitive Systems Research*, 85:101216, 2024. ISSN 1389-0417. doi: https://doi.org/10.1016/j.cogsys.2024.101216. URL https://www.sciencedirect.com/science/article/pii/S1389041724000093.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures. *Proc. VLDB Endow.*, 16(3):406–419, November 2022. ISSN 2150-8097. doi: 10.14778/3570690.3570692. URL https://doi.org/10.14778/3570690.3570692.

Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Xulong Tang, Chenchen Liu, and Xiang Chen. A survey of large-scale deep learning serving system optimization: Challenges and opportunities. *arXiv preprint arXiv:2111.14247*, 2021.

Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1049–1062, Renton, WA, July 2019a. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/zhang-chengliang.

Xiaoxuan Zhang et al. Ann hybrid architectures. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. USENIX Association, 2019b. URL https://www.usenix.org/conference/atc19/presentation/zhang. Accessed 2024.

Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):1–56, 2006. URL https://dl.acm.org/doi/10.1145/1132956.1132959.

A Appendix: Task 1

A.1 System Workflow

The system employs a multi-stage pipeline for similarity search, encompassing data preprocessing, clustering (K-Means), query processing, and evaluation. It's designed to handle high-dimensional data and is optimized for execution on NVIDIA GPUs using the CuPy library. The workflow includes optional normalization, different approximate nearest neighbor (ANN) search techniques, and exact K-Nearest Neighbors (KNN) search.

A.2 System Initialization and Data Preprocessing

A.2.1 System Initialization

The first stage, *System Initialization*, configures the CuPy environment and pre-allocates GPU memory. This step is crucial for avoiding runtime memory allocation overhead. The function estimate_max_batch_size determines the optimal batch size to prevent out-of-memory errors, given the dimensionality of the input vectors and available GPU memory.

```
def estimate_max_batch_size(D, memory_limit_bytes=2 * 1024**3):
    bytes_per_vector = D * 4  # float32
    return max(1, memory_limit_bytes // bytes_per_vector)
```

Here, `D` is the dimensionality of the data, and the memory limit is set to 2GB by default. The function computes how many vectors of dimension `D` can fit into the available GPU memory.

In the *Data Preprocessing* stage, input data (vectors A and X) are converted into CuPy arrays using `cp.asarray(data, dtype=cp.float32)`. This transfers the data to GPU memory. The dimensionality of the input is also checked:

```
if X_cp.ndim == 1:
    X_cp = X_cp[None, :]
```

This code ensures that the input vector X is always a 2D array, which is important for consistent matrix operations later on. Additionally, the diagram shows an optional *Normalization* step, typically L2 normalization, which is required when using cosine similarity:

Clustering (K-Means)

The *Clustering* stage employs the K-Means algorithm to group similar data points into clusters. This stage involves three key steps: *Initialization, **Assignment Step, and **Update Step*.

A.2.2 Initialization

Initial centroids are randomly selected from the input data. Randomly initializing `K` centroids from the dataset ensures a diverse starting point for the algorithm.Each data point is assigned to the nearest centroid based on a chosen distance metric.This step computes the distance between each data point and all centroids and assigns the point to the cluster with the nearest centroid.

Centroids are updated by computing the mean of all data points assigned to each cluster. The function also includes a fallback mechanism for empty clusters to prevent errors. If a cluster is empty (no data points assigned to it), the centroid remains unchanged to avoid introducing `NaN` values.

The algorithm iterates between the assignment and update steps until a convergence criterion is met.The *KNN* path finds the K nearest neighbors to a query vector by exhaustively computing distances between the query vector and all data points. The `cp.argsort` function efficiently identifies the indices of the K smallest distances, which are then returned as the nearest neighbors.

A.2.3 Approximate Search (ANN)

The *Approximate Search* path provides faster, albeit less precise, search results using techniques like cluster pruning and graph-based search. The code presents three ANN search methods: Cluster-based ANN (`our_ann`), HNSW-based ANN (`our_ann_hnsw`), and IVFPQ-based ANN (`our_ann_ivfpq`).

Cluster-Based ANN This method first identifies the nearest clusters to the query and then searches only within those clusters. It uses `K1` to specify the number of clusters to consider and `K2` to specify how many points to retrieve from each cluster. Finally, it performs a final ranking to select the top K results.

HNSW-Based ANN The HNSW (Hierarchical Navigable Small World) method constructs a multi-layered graph and uses it to efficiently find the nearest neighbors.

The construction involves building a graph where each node is connected to its M nearest neighbors. The search process traverses this graph, maintaining a candidate list of the best ef candidates.

A.3 Evaluation

The final stage of the workflow is *Evaluation. The primary metrics are **Recall Rate & Execution Time*. Recall rate measures the accuracy of the ANN methods compared to the exact KNN results.This function calculates the ratio of relevant items retrieved to the total number of relevant items. Execution time measures the indexing and query performance of the system.This function benchmarks the execution time of different ANN methods, allowing a comparison of their performance characteristics.

Table 1: Recall across 10 runs for different distance functions

Distance Function	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Mean Recall
L2 (Euclidean)	0.8	0.9	0.6	0.5	0.6	0.7	0.7	0.9	0.6	0.9	0.72
Cosine	0.5	0.6	0.5	0.6	0.4	0.6	0.4	0.8	0.3	0.5	0.52
Dot Product	0.2	0.6	0.6	0.8	0.3	0.6	0.5	0.5	0.3	0.8	0.52
Manhattan (L1)	0.6	0.3	0.5	0.8	0.5	0.6	0.5	0.4	0.6	0.6	0.54

A.4 Table

Table 2: Execution Times and Speedup Ratios (CPU vs GPU) Across Methods, Distances, and Dimensions

Method	Distance	Dimension	CPU Time (s)	GPU Time (s)	Speedup Ratio
KNN	L2	D=2	0.0005	0.0010	0.52
		D=1024	0.0328	0.0093	3.53
	Cosine	D=2	0.0003	0.0012	0.27
		D=1024	0.0051	0.0031	1.65
	Dot	D=2	0.0002	0.0009	0.26
		D=1024	0.0046	0.0025	1.81
	Manhattan	D=2	0.0002	0.0009	0.21
		D=1024	0.0072	0.0026	2.79
ANN	L2	D=2	0.5356	11.8861	0.05
		D=1024	0.7502	2.4130	0.31
	Cosine	D=2	0.7384	12.3839	0.07
		D=1024	0.4556	1.2920	0.35
	Dot	D=2	1.8328	31.3044	0.06
		D=1024	0.2447	0.6975	0.35
	Manhattan	D=2	0.4970	7.7278	0.05
		D=1024	0.4427	1.0271	0.43

B Appendix: Task 2

B.1 Motivation

Queueing and batching are not merely tactical optimizations; they serve as pillars for building robust, high-throughput model-serving pipelines. Recent studies illustrate how managed queues buffer incoming workloads to avoid sudden overload, thereby smoothing out utilization over time (Yu et al., 2021). Batching further boosts efficiency when infrequent or sporadic requests would otherwise leave GPUs idle. Systems like TensorFlow Serving (Developers, 2022) and TorchServe (PyTorch Contributors, 2025) incorporate these ideas by default, demonstrating real-world viability. NVIDIA Triton extends this paradigm with advanced multi-model support and adaptive scheduling (Morales-Hernández et al., 2021). The ability to handle abrupt surges or lulls in traffic while keeping large models fully utilized distinguishes state-of-the-art serving infrastructures from less flexible solutions. When combined with optimized retrieval and postprocessing stages, queueing and batching help maintain steady, predictable performance across an entire ML inference pipeline.

B.2 Overview

The system architecture draws inspiration from established serving frameworks such as TensorFlow Serving (Developers, 2022) and TorchServe (PyTorch Contributors, 2025), both of which employ internal queues and batch processing to handle inference tasks. NVIDIA Triton (Morales-Hernández et al., 2021) further generalizes this approach by supporting multiple backends and dynamic batching strategies that adapt to varying traffic patterns. Our focus remains on effectively dispatching requests to a single model at high rates, although the same concepts of request queueing and batch scheduling could be extended to multi-model serving scenarios. Separating arrival from processing not only helps prevent transient overloads but also allows for more precise control over how batches are formed. The trade-off, as explored in our experiments, is that batching can introduce additional wait time for early arrivals, a cost that is offset by substantially improved GPU utilization and throughput when request volume is high (Yu et al., 2021).

B.3 System Measurement

In each test, a Python script submitted requests at a target rate for a fixed duration (e.g., 60 seconds), while uniformly or randomly spacing out arrivals (Yu et al., 2021). To approximate real-world scenarios, we varied the RPS in increments and observed when the system approached saturation. Latency distributions (including 95th and 99th percentiles) were extracted from the request logs, and throughput was computed by dividing the total number of successful responses by the test duration. We also tracked any unsuccessful or timed-out responses to calculate error rates, which occasionally occurred when request spikes coincided with GPU-intensive operations. While repeating each test multiple times would have minimized variance, constraints such as GPU time quotas and hardware availability limited the number of repeated runs. Nonetheless, the recorded data confirm our central findings: batching substantially improves resource utilization at higher loads, and queueing helps prevent instantaneous overload, although these mechanisms can introduce minor latency overheads for users who arrive just as a batch is forming (Morales-Hernández et al., 2021; Developers, 2022; PyTorch Contributors, 2025).

B.4 Design: Request Queue and Batcher

This section expands on the five primary elements of the request queue and batcher subsystem: rationale, implementation details, experiments, overall findings, and challenges with corresponding solutions.

1) Rationale: In practice, ML serving infrastructure must handle diverse and at times unpredictable traffic levels. A request queue allows the system to buffer incoming requests when the GPU is busy, preventing immediate rejections. By decoupling arrival from processing, the system also gains time to consider how best to form batches.
2) Implementation Details: The queue is maintained in main memory, and each request is stored alongside its arrival timestamp. A batcher thread or process polls the queue for items. If the queue exceeds a size threshold, or a specified timeout occurs, the batcher builds a consolidated tensor input to feed the model. After inference, responses are paired back to the original requests and returned to the user.
3) Experiments and Metric Analysis: We ran controlled tests simulating multiple traffic patterns (e.g., uniform, bursty/Poisson, random) at increasing request-per-second rates. Metrics included average and tail latencies, throughput, and error rates. Repeated trials revealed how different batch sizes and timeouts influenced both performance and the responsiveness experienced by early-arriving requests.
4) Overall Experimental Findings: Batching consistently improved GPU utilization and throughput, especially at moderate to high load levels. Smaller batches reduced waiting time per request but led to modest throughput gains. Larger batches or extended timeouts provided stronger throughput improvements but increased tail latency. Queuing successfully prevented overloads when traffic spiked, allowing the system to gradually drain bursts instead of dropping requests.

5) Challenges and Solutions: Key challenges arose from limited GPU memory for extremely large batches, difficulties in selecting a default batch size across varying workloads, and potential overhead in forming batches on the fly. We addressed these by bounding the maximum batch size,

Implementation	Total Test Time (s)	Total Requests	Successful Requests	Error Rate (%)	Avg. Latency (s)	Throughput (req/s)
base	3281.73	53400.0	5627.0	63.023	146.0592	0.011
queue	3282.32	53400.0	6028.0	60.905	143.1135	0.011
rq+batcher	3221.68	53400.0	7644.0	53.829	140.4009	0.011

Table 3: Summary Statistics by Implementation. Each row reports metrics over the entire test duration for one system configuration.

adopting a short default timeout for low-traffic scenarios, and continuously monitoring queue length to detect overload conditions. Future work could refine these parameters adaptively, shifting thresholds based on recent demand and measured latency, thus improving both reliability and resource usage.

B.5 Extended Open Discussions

Limitations in Traffic Patterns and RPS Combinations: Although multiple arrival patterns were tested (uniform, Poisson, random), not every pattern was paired with every load level. As a result, it is difficult to say with certainty which pattern creates the greatest difficulty at high RPS. Comprehensive testing is time-intensive and was constrained by our hardware resources.
Single-Run Evaluations: Each configuration was run only once, so outliers might skew results. In a more rigorous setting, one would conduct multiple runs and compute confidence intervals to draw stronger conclusions about the system's consistency.
Resource Constraints and Measurement Artifacts: By co-locating the load generator and inference on the same machine, CPU contention likely inflated latencies. Our GTX 1060 GPU (6 GB VRAM) further restricted the batch sizes we could feasibly test. Ideally, we would separate load generation onto another host and adopt a more powerful GPU to validate the concurrency solutions at higher scales.
GPU Limitations on a GTX 1060 (6GB VRAM): Although our queue-and-batcher approach did improve throughput, the absolute gains remained modest due to memory constraints and model size. Larger batches risked out-of-memory errors, but using half-precision or INT8 inference and employing model distillation could enable higher concurrency without exhausting VRAM.
Real-World Considerations and Traffic Patterns: Bursty or Poisson traffic is common in production, and our tests showed that moderate Poisson loads saw efficiency gains from batching, but higher RPS saturated the GPU quickly. In practice, advanced scaling mechanisms might be required to manage such peaks effectively.
Expected vs. Surprising Outcomes: We anticipated batching to raise throughput, though the extent of these gains was smaller than hoped (e.g., from 0.01 req/s to 0.02 req/s in some low-load scenarios). The most surprising result was that absolute latencies stayed relatively high at moderate RPS, suggesting GPU overhead and model complexity were the dominant factors.
Statistical Validity and Further Steps: Comparisons across traffic patterns remain preliminary because we ran more Poisson tests than uniform or random. Systematic tests of each pattern at each RPS level, combined with multiple runs, would clarify these differences. Tracking standard deviations or confidence intervals would give additional perspective on performance variance.
Future Directions: Dynamic Batching could adapt the batch size in real time based on observed load. **Multi-Thread CPU Preprocessing** might accelerate tokenization or embedding steps. A **Hybrid or Tiered Queue** approach could separate low-latency from high-throughput workloads. Such refinements would allow queue-and-batcher systems to serve a broader range of ML inference demands with better latency control and resource management.