MLS COURSEWORK

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper presents a comparative analysis of CPU and GPU performance for information retrieval tasks, focusing on query-to-vector search using exact k-nearest neighbours and approximate nearest neighbours, and model serving systems, testing the effectiveness of queueing and batching for efficient real-world employment. We evaluate both architectures across a range of retrieval workloads. Our findings highlight the GPU's advantages in parallelism and high-throughput computation, especially for dense retrieval and large-scale model serving, while CPUs demonstrate competitive performance for low-latency, memory-efficient tasks.

1 INFORMATION RETRIEVAL ON GPU (TASK 1)

1.1 MOTIVATION

Information retrieval is the process of extracting relevant information from a large collection of unstructured data, such as text documents, web pages or multimedia files, in an efficient and accurate way (Singhal et al., 2001). It involves the use of computational algorithms, data structures, and statistical models to efficiently match user queries with relevant documents.

The typical workflow in an information retrieval system comprises several steps (Kowalski, 2007). All the collected data is processed and stored in an index. Once a query is submitted, it is processed (expanded or refined) to improve search results, and the query is matched against indexed documents (e.g., neural embeddings (Zuccon et al., 2015) can be used for this) and ranked based on the assigned relevance scores. The most relevant documents are retrieved and presented to the user in a structured format. User interactions are used to evaluate the results and collect feedback to improve future search queries.

However, information retrieval struggles with efficiently organising and searching large, high-dimensional datasets. To address this, clustering is used to group similar documents, which helps reduce the search space and retrieval time, and makes scaling to large datasets more manageable. However, as the size of datasets increases, the time and memory consumption of clustering algorithms grow significantly (Whang et al., 2012)[1]. Common clustering algorithms like K-Means (Hartigan & Wong, 1979) may struggle to find a local minimum depending on the initialisation of the centroids, often yielding suboptimal results (Celebi et al., 2013). Moreover, it is suggested that the notion of distance becomes less meaningful in high-dimensional spaces, making it harder to identify meaningful clusters (Aggarwal et al., 2001).

Graphic Processing Unit (GPU) acceleration addresses such challenges by allowing parallel processing and enhancing the speed of computations such as distance calculations and centroid updates. GPUs can simultaneously process multiple data points, decreasing the time complexity associated with clustering algorithms (Asano et al., 2009). They also have optimised memory access patterns that enable high-bandwidth, low-latency data movement when threads access memory in a coalesced and structured manner, allowing for the effective management of extensive datasets and high-dimensional data. Additionally, GPUs excel at iterative optimisation tasks involved in clustering algorithms, such as the recalculation of centroids in K-Means (Li et al., 2013). Consequently, GPU-based clustering has various practical applications, such as recommendation engines of video streaming platforms like YouTube or Netflix and image and video processing (Roshdi & Roohparvar, 2015).

[1] for example, K-Means has a time complexity of $O(n^2)$, where n is the number of data points.

1.2 Overview

Our GPU-accelerated system is designed to efficiently handle large-scale information retrieval and clustering tasks by leveraging parallelism and optimised memory access patterns. It supports three core algorithms: K-Nearest Neighbours (KNN) for similarity search (Peterson, 2009), K-Means for unsupervised clustering, and Approximate Nearest Neighbours (ANN) methods for fast retrieval in high-dimensional spaces (Andoni et al., 2018). Our GPU-accelerated clustering system is divided into three layers, and visualised in Figure 1.

Data Generation Layer. This layer generates synthetic data for both the database and queries, providing customisations for size, dimensionality, and the number of clusters. Data can be generated randomly or using specified distributions. The data is converted to 32 precision to balance efficiency and accuracy.

GPU Processing Layer. This module consists of distance functions using CUDA kernels wrapped in CuPy, Triton, and Pytorch. It also consists of KNN, K-Means, and ANN algorithms for clustering and finding nearest neighbours. The generated data is first loaded onto the GPU, after which the GPU processing layer produces output, which is offloaded onto the CPU again.

Evaluation Layer. This layer computes the total and query time of all implemented algorithms. It also calculates the recall rates of the ANN implementations compared to the exact KNN results.

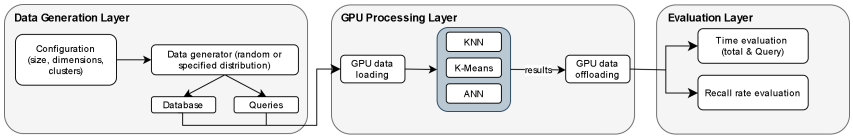

Figure 1: Workflow of the GPU-based clustering system.

We implement our system in different GPU-accelerated versions using PyTorch, CuPy, and Triton, and also include a Central Processing Unit (CPU) benchmark using Numpy. The core ideology behind our system is parallelisation, with each algorithm designed to work with 1D and 2D databases and queries.

1.3 Distance Functions

Problem Definition. Distance functions play a vital role in information retrieval, particularly in ranking and comparing document embeddings (Kusner et al., 2015), or query-document similarity scores (Yang et al., 2009). They help measure how similar or different two points (e.g., query vs. document vector) are in a given feature space and are thus crucial in clustering algorithms to group similar documents together.

Design Choice. In this work, we consider four distance functions to calculate the similarity between two points in a feature space. For conciseness, we extensively discuss the implementations using CuPy.

First, we utilise the Euclidean distance, or L2 distance, which is defined as follows: $d_{\text{L2}}(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$, where x and y denote the document vector and query vector, respectively, and n represents the dimensionality of both vectors. It measures the straight-line distance between two points, derived from the Pythagorean theorem. As the differences are squared, it is sensitive to large feature difference, and works well when magnitude matters or when features are normalized or comparable in range.

The second distance function is the Manhattan distance, or L1 distance, which measures the sum of absolute differences across dimensions. It is defined as follows: $d_{\text{L1}}(x, y) = \sum_{i=1}^{n} |x_i - y_i|$. Compared to L2 distance, it is more robust to outliers as it does not exaggerate large differences. It is, therefore, more useful when changes in individual features are important and in sparse data or high-dimensional settings where many features are zero.

Third, we make use of the negative dot product. The dot product measures similarity, projecting one vector onto another, and as such, it would return a high value for vectors that are close to each other.

However, our sorting algorithms look for the lowest distances, and taking the negative transforms the dot product into a form of distance (lower is better). It is defined as follows: $d_{\text{dot}}(x, y) = -\sum_{i=1}^{n} x_i y_i$, which measures the directional similarity and magnitude of two vectors. It is often utilised for applications with embedding spaces.

Last, the cosine distance is used, which is formulated as follows: $d_{\cos}(x, y) = 1 - \frac{x \cdot y}{||x|| ||y||}$. Cosine distance measures the angle between two vectors, regardless of their magnitude. It focuses on orientation and similarity of direction rather than distance. Cosine itself is also a measure of similarity by default, and subtracting cosine from 1 results in a distance function. It can be used for comparing text embeddings, or anything where scale is not informative, with applications in text retrieval and natural language processing.

Implementation Details. For implementation, we wrap CUDA kernels in Python using CuPy's `RawKernel`. For certain applications, it is desirable to compare a database to a matrix of queries instead of a single vector, which may result in slow execution times as the system has to loop over each query. To address this, our distance functions are able to parallelise vector to vector, matrix to vector and matrix to matrix operations. For memory access, each CUDA thread is responsible for computing the distance between one pair of vectors, where threads are arranged in a 2D grid depending on the number of vectors in the database and query matrix, respectively. An overview of the algorithm is provided in Algorithm 1 in the Appendix. For a general CPU-based comparison, we implement the same distance functions using Numpy operations.

1.4 Top-K Retrieval

Problem Definition. Exact KNN search becomes computationally expensive for large-scale datasets, with complexity $O(N)$ per query. For a dataset with N points in D dimensions, finding exact nearest neighbours requires computing and sorting N distances for each query point. This becomes impractical for large N or high D. ANN approximates KNN, presenting a trade-off between perfect accuracy and dramatically improved speed, making ANN algorithms essential for real-world applications.

Design Choice. Two ANN algorithms are implemented in this work based on two different underlying principles: K-Means and Locality-Sensitive Hashing (LSH). For K-Means, the data is divided into K clusters ($K > K_1$) using K-Means clustering, then the algorithm finds the K_1 closest cluster centroids to the query. Next, for each elected cluster, it finds the K_2 nearest points, after which the K closest points from this candidate pool are returned. This reduces the search complexity to approximately $O(K + K_1 K_2)$ per query, making it particularly efficient for naturally clustered data.

The second implementation utilises LSH, which uses random projections to hash similar points into the same buckets with high probability. The key principle is that nearby points in the original space are likely to share similar hash values. This reduces complexity from $O(N)$ to approximately $O(L + nC)$, where L is the number of hash tables and nC is the average number of candidates.

Implementation Details. Both of our designs for ANN distinguish between total and query time: K-Means ANN requires initial clustering of the dataset, while LSH needs hash table generation and hash code computation. However, given a database, this only needs to be done once, and as such is not taken into account when comparing performance with KNN. For K-Means ANN, query time involves finding K_1 nearest centroids and examining K_2 points from their clusters. One challenge we encountered was balancing speed and recall with the hyperparameters K_1 and K_2, both determine the number of points the algorithm is searching through. We set $K_1 = \lceil 0.8K \rceil$ and $K_2 = \lceil 2K \rceil$, which was determined empirically. Similarly, for LSH, the hyperparameters include L, the number of hash tables determining the chance that similar points hash to the same bucket as the query, and the number of hash functions per table determining how specific each hash bucket is. We set $L = 6$ and the number of hashes to 2. An overview of ANN K-Means and ANN LSH can be found in Figures 2 and 3 in the Appendix, respectively.

For conciseness, we restrict the implementation details to CuPy. Our initial implementation using standard CuPy utilities for distance calculations (`cp.linalg.norm`, `cp.dot`) and top-K selection (`cp.argpartition`) proved inefficient for large-scale nearest neighbour search. We optimised these calculations by developing custom CUDA kernels for all distance metrics (cosine, L2, dot, and L1) to maximise GPU utilisation. Furthermore, the LSH implementation leverages

vectorised operations through `cp.tensordot` for efficient hash computation, while both methods use a custom quickselect-based kernel for top-K selection, significantly improving performance over `cp.argpartition`. It performs an efficient block-wise top-K selection using a custom CUDA kernel: it splits the input into blocks, finds the K smallest distances within each block in parallel, and then does a final global top-K selection from all block-level candidates. This improves speed by massively reducing the number of values considered in the final top-K step.

Although no Out-Of-Memory (OOM) issues were encountered on the GPU, being able to handle datasets that do exceed GPU memory is essential for real-world problems. Therefore, we also implemented batched versions of each framework, since the large size of the database is a common cause OOM issues. We estimate the available GPU memory, after which the database is loaded onto the GPU in batches based on the available memory, after which batched distance calculations are performed. Similarly, for the ANN algorithms, everything that includes the database is done in batches. For testing, we simulate that the GPU has 4 GBs of VRAM.

Evaluation Each function was run 13 times, with the first 3 runs used as warm-up and discarded. Final results are averages over the remaining 10 runs. We compare three implementations: NumPy KNN, CuPy KNN, and CuPy LSH ANN. Metrics include total execution time, execution time excluding data transfer (to isolate algorithm performance), and for ANN, separate query time and recall (for correctness). Note that the batched implementation does not report data loading time due to its integrated loading and computation process. Tests were conducted on four datasets to evaluate performance across varying dimensionalities and sizes. The largest tested file was `400000×1024` (N, D), limited by disk space and available compute time. Below we present results for the cosine distance function and only cover ANN LSH, while detailed results for the other distance functions and the K-Means ANN can be found in the appendix.

Table 1: Results using the cosine distance.

			CuPy						CuPy batched		
	KNN CPU		KNN		ANN LSH				KNN	ANN LSH	
(N,D)	total (s)	w/o data (s)	total (s)	w/o data (s)	total (s)	w/o data (s)	query (s)	recall	total (s)	total (s)	query (s)
(4000,2)	0.0004 ±0.0000	0.0004 ±0.0000	0.0023 ±0.0006	0.0006 ±0.0020	0.0044 ±0.0021	0.0039 ±0.0021	0.0033 ±0.0020	1.00	0.0026 ±0.0005	0.0037 ±0.0010	0.0021 ±0.0006
(4 mil,2)	0.5653 ±0.0254	0.5609 ±0.0254	0.0116 ±0.0005	0.0050 ±0.0004	0.0162 ±0.0008	0.0118 ±0.0004	0.0113 ±0.0004	1.00	0.0293 ±0.0024	0.3882 ±0.0266	0.0877 ±0.0084
(4000,65536)	0.5454 ±0.0142	0.5403 ±0.0137	0.0119 ±0.0009	0.0049 ±0.0004	0.0157 ±0.0005	0.0116 ±0.0005	0.0111 ±0.0005	1.00	0.3499 ±0.0180	5.1337 ±0.1269	1.4016 ±0.0344
(400000,1024)	0.8271 ±0.0811	0.4978 ±0.0636	4.5820 ±0.4674	0.0126 ±0.0008	4.3943 ±0.3285	0.0293 ±0.0012	0.0288 ±0.0011	0.80	2.7219 ±0.5790	10.8444 ±0.8801	2.8530 ±0.6470

Notes: We report the total time the algorithm takes, the time without converting the data to float32 and onto the GPU if applicable (w/p is the abbreviation for without), and the query time and recall for the ANN algorithms. For the batched versions, no times without data loading are available.

Results and Analysis LSH performed better on high-dimensional, uniformly distributed data, while k-means ANN showed superior performance on naturally clustered datasets [see Section 1.7]. Memory usage was higher for LSH due to multiple hash tables, but it offered more consistent query times across different data distributions. The trade-off between recall rate and query time could be adjusted through the involved hyperparameters (K_1, K_2 for k-means; number of hash tables for LSH), allowing flexibility based on application requirements.

1.5 Alternative implementations (Pytorch and Triton)

For Pytorch, we mainly make use of built-in operations. For the Euclidean and Manhattan distance, `torch.cdist` is used. This function computes batched pairwise distances between two sets of vectors and leverages matrix multiplication for the Euclidean distance for efficiency. Similar to the CuPy implementation, Pytorch is implemented in such a way that the distance functions also handle matrix-to-matrix operations. `torch.topk(distances, k=K, largest=False, sorted=False)` is used, which retrieves the indices of the K smallest values along the last dimension of the distances tensor, which typically represents distances from a query point to all other points (e.g., in KNN search). By setting `largest=False`, it selects the smallest values (nearest neighbours), and `sorted=False` returns them in an arbitrary order to improve performance. The ANN algorithms for Pytorch are similar to the CuPy implementations but use the Pytorch distance function.

Similar to CuPy, Triton also utilises kernels to implement the distance functions. In our Triton kernels, each program instance (i.e., thread) is responsible for computing the distance between one row/vector in a database and one row/vector in a query matrix. In case when the query is a vector, the kernel launches N threads, one per row, using `tl.program_id(0)` to identify the row index. Each thread processes its row in chunks of columns, using `tl.make_block_ptr` to efficiently load data with boundary checks and padding, calculates the distance between the row and query for this chunk, and uses `tl.advance` to move the pointer to the next chunk. Furthermore, Triton leverages `triton.autotune` for efficiency. Triton's autotune feature automatically searches for the best kernel configuration, such as block sizes and tiling parameters, based on the input shapes and hardware characteristics. Triton benchmarks each one on the fly and caches the fastest for future use. This helps maximise performance without manual tuning, adapting the kernel to different GPUs or input sizes efficiently.

For the ANN implementations, they mimic the CuPy implementation with a block-wise top-K selection function using a kernel. Last, batched versions of Pytorch and Triton are also implemented in similar fashion to CuPy.

1.6 Testing framework

To facilitate benchmarking and iterative development, a reusable `Testing` class was implemented as part of a wider testing framework. It supports customisable configuration via `argparse` and allows dynamic specification of runtime parameters, such as trials to run (e.g., `N=8000 D=1024`), the distance function(s) to use, the number of warm-up runs, among others.

Internally, `Testing` leverages a `Benchmark` class responsible for collecting performance metrics during execution. `Benchmark` supports runtime checkpoints, such as query-time capture, which contribute to flexible timing breakdowns (e.g., isolating total runtime from core algorithm query time). `Testing` is iterable, and emits standardised test cases across runs and frameworks (Algorithm 4). It achieves this by embodying an iterator abstraction, which provides fine-grained control over the iterated elements (Algorithm 5).

For each trial, performance data is logged and a CSV summary is subsequently generated. When a specific framework file (e.g., PyTorch) is executed, a corresponding report is produced. When the central testing program is run, it sequentially executes all framework files, aggregating results into a unified report for cross-comparison.

1.7 Influence of different data distributions

The effectiveness of our k-means based ANN implementation varies significantly with data distribution patterns. Specifically, we focused on two distinct variations: well-clustered data (Figure 3) and more spread-out data (Figure 4). For well-clustered distributions (*cluster std.* $= 0.3$–0.5), a smaller K_1 (4–8) yields high recall due to clearly defined cluster boundaries. In contrast, spread-out data (*cluster std.* $= 0.8$–1.0) requires larger K_1 values (8–16) to achieve comparable recall, owing to increased cluster overlap. This direct correlation between cluster separation and the optimal K_1 facilitates more efficient hyperparameter tuning based on data characteristics.

2 Model Serving Systems (Task 2)

2.1 Motivation

Machine learning model serving is the process of deploying trained models to make predictions on new, incoming data, often in real time. However, challenges occur when handling online inference requests. Many online applications (e.g., search engines) require sub-second response times. The number of individual requests may fluctuate over time, making it difficult to maintain consistent latency. Furthermore, processing single requests at a time leads to the underutilisation of hardware accelerators, which are designed to perform optimally with parallel computations. In addition, real-time deployment can be expensive, as continuously spinning up infrastructure for individual requests increases operational costs. Last, high request volumes can overwhelm the system if not properly managed, leading to increased response times or dropped requests.

To address such shortcomings, efficient request queueing and batch processing are necessary. Queueing temporarily holds incoming requests in a buffer. As such, it allows the server to organise and group them for more efficient processing. Batching combines multiple requests into a single batch and processes them together. It takes advantage of vectorised computation on CPUs/GPUs, significantly improving throughput, and reduces overheads across multiple requests. Together, queueing ensures smooth intake of requests, while batching maximises computation per cycle, striking a balance between latency and throughput. For example, OpenAI APIs internally use intelligent batching to serve large-scale inference, ensuring responsiveness despite handling thousands of concurrent users.

2.2 Overview

The system implements a Retrieval-Augmented Generation (RAG) pipeline deployed via a FastAPI web server, supporting both standard request handling and batched processing for improved throughput. Requests are submitted through one endpoint, `/rag-batched` queues requests and processes them in mini-batches for efficiency. The core logic revolves around embedding the query using a transformer model, retrieving top-K similar documents based on embedding similarity, and generating answers using a text-generation pipeline. Figure 2 provides a high-level overview of the request queue and batch processing pipeline.

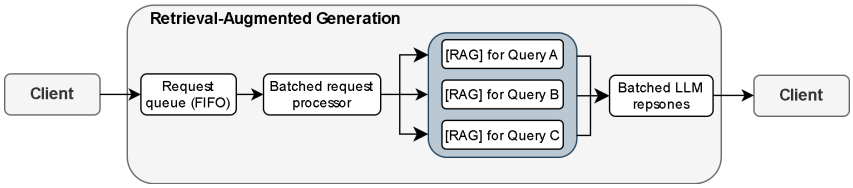

Figure 2: The implemented RAG system using queueing and batching.

2.3 System Measurement

We evaluated the RAG system's performance using a structured testing framework in `query_script.py`, employing three traffic patterns to simulate load conditions: (1) constant rate at 1 request/second, (2) random rate pattern with variable delays (e.g., 0.1–2.0s), and (3) Poisson-distributed inter-arrival times to approximate real-world traffic. Each test used the same query input and ran for 60 seconds, targeting both `/rag` and `/rag-batched` endpoints. We measured average and median response times, along with the 90th, 95th, and 99th percentiles to capture both central tendency and tail latency. These metrics were selected to provide a holistic view of system responsiveness while highlighting potential performance degradation under high load or outlier conditions. Metrics were collected by logging request timestamps and durations directly within the testing script, with results aggregated post-run. A challenge encountered was an inconsistent system load affecting baseline measurements; this was mitigated by isolating the test environment and repeating runs to average out anomalies.

2.4 Request Queue and Batcher

Rationale for Request Queue and Batcher. In RAG systems, batching multiple requests improves GPU acceleration by minimising per-request overhead and maximising parallelisation. A queue-based system further manages load by queuing requests during traffic spikes, preventing overload. Together, batching and queuing enhance throughput and optimise memory usage.

Implementation Details. Our implementation employs a thread-safe FIFO queue for managing requests, using Python's Future objects to handle results asynchronously. A dedicated background daemon thread [2] continuously monitors the queue, batching requests based on two globally set parameters: `MAX_BATCH_SIZE` (4) and `MAX_WAITING_TIME` (1.0s). This design ensures efficient

[2] ensures that the main program is not blocked from exiting

GPU utilisation through controlled batching while also maintaining responsive performance under varying load conditions.

Experiments and Metrics Analysis. We evaluated the RAG system's performance under various conditions, utilising a structured testing framework implemented in `query_script.py`. The tests were conducted over 60-second intervals with 60 total requests, comparing two service endpoints: the standard `/rag` and the batched `/rag-batched` implementations. We tested under both constant rate (1 request/second) and random rate patterns (with delays between 0.1-2.0 seconds), using the query "Which animals can hover in the air?" as our test input.

We collected all the important performance metrics such as total requests processed, error counts, average response times, median response times, and various percentiles (90th, 95th, 99th). The distribution of response times from the load testing is visualised as a histogram in Figure 5.

Challenges and Solutions. A significant challenge for us was setting up the system so it could do queuing and batch processing simultaneously while maintaining memory efficiency and system stability. We addressed this challenge by implementing a batching mechanism that aggregates requests based on either time intervals or batch size. Each batch is then executed on a separate thread, with the total number of active threads capped to ensure controlled resource usage.

3 Discussion of Task Synergy

GPU kernel optimisation (Task 1) and distributed ML system optimisation (Task 2) are interdependent components in modern machine learning pipelines. Kernel-level performance determines how efficiently compute resources (e.g., GPUs) are utilised, while distributed system logic governs data and task coordination across nodes. Optimising one without considering the other often leads to inefficiencies; e.g., fast kernels may idle wait for data due to poor scheduling, or alternatively, optimal data distribution may overload poorly tuned kernels.

Their synergy lies in co-design: predictable and well-optimised kernels enable effective overlapping of communication and computation; conversely, distributed systems must adapt to kernel characteristics to minimise bottlenecks, such as stragglers or bandwidth contention. Trade-offs arise between maximising GPU utilisation and ensuring system-level balance. For example, fused kernels may reduce launch overhead but introduce latency variability that disrupts distributed pipelines.

Alternative designs include hierarchical optimisation (kernel-to-cluster), operator-aware scheduling, and asynchronous execution models to decouple computation and communication. Overall, integrated optimisation across both tasks is essential for scalable and efficient ML system performance.

REFERENCES

Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pp. 420–434. Springer, 2001.

Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. Approximate nearest neighbor search in high dimensions. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, pp. 3287–3318. World Scientific, 2018.

Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 international conference on field programmable logic and applications*, pp. 126–131. IEEE, 2009.

M Emre Celebi, Hassan A Kingravi, and Patricio A Vela. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert systems with applications*, 40(1): 200–210, 2013.

John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

Gerald J Kowalski. *Information retrieval systems: theory and implementation*, volume 1. springer, 2007.

Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. From word embeddings to document distances. In *International conference on machine learning*, pp. 957–966. PMLR, 2015.

You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. Speeding up k-means algorithm by gpus. *Journal of Computer and System Sciences*, 79(2):216–229, 2013.

Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

Akram Roshdi and Akram Roohparvar. Information retrieval techniques and applications. *International Journal of Computer Networks and Communications Security*, 3(9):373–377, 2015.

Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4): 35–43, 2001.

Joyce Jiyoung Whang, Xin Sui, and Inderjit S Dhillon. Scalable and memory-efficient clustering of large-scale social networks. In *2012 IEEE 12th International Conference on Data Mining*, pp. 705–714. IEEE, 2012.

Yin Yang, Nilesh Bansal, Wisam Dakka, Panagiotis Ipeirotis, Nick Koudas, and Dimitris Papadias. Query by document. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pp. 34–43, 2009.

Guido Zuccon, Bevan Koopman, Peter Bruza, and Leif Azzopardi. Integrating and evaluating neural word embeddings in information retrieval. In *Proceedings of the 20th Australasian document computing symposium*, pp. 1–8, 2015.

A VISUALISATION OF DATA DISTRIBUTIONS

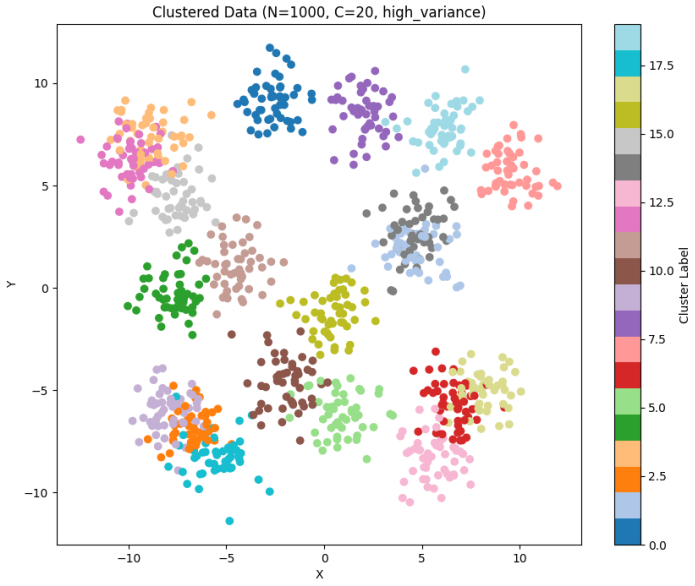

Figure 3: Clustered data with $N = 1000$, $D = 2$, and high variance

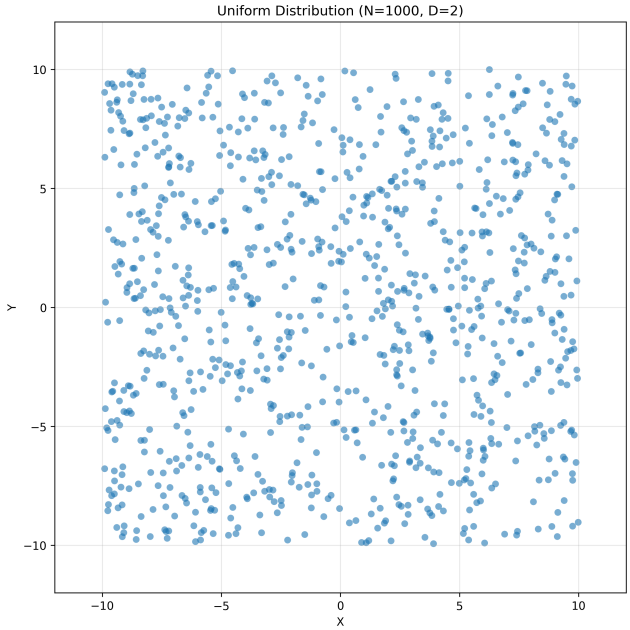

Figure 4: Uniformly distributed data with $N = 1000$, $D = 2$

B ALGORITHMS

Algorithm 1 Cosine Distance Computation

Require: Matrix $X \in \mathbb{R}^{N \times D}$, Matrix $Y \in \mathbb{R}^{K \times D}$
Ensure: Output matrix $out \in \mathbb{R}^{N \times K}$ containing pairwise cosine distances

for each thread (i, j) in parallel where $i < N$ and $j < K$ **do**
  $dot \leftarrow 0.0$ ▷ Initialize dot product
  $normX \leftarrow 0.0$ ▷ Initialize norm for vector X_i
  $normY \leftarrow 0.0$ ▷ Initialize norm for vector Y_j

  for $d = 0$ to $D - 1$ **do**
    $a \leftarrow X[i \cdot D + d]$ ▷ Get component d of vector X_i
    $b \leftarrow Y[j \cdot D + d]$ ▷ Get component d of vector Y_j
    $dot \leftarrow dot + a \cdot b$
    $normX \leftarrow normX + a \cdot a$
    $normY \leftarrow normY + b \cdot b$
  end for

  $normX \leftarrow \sqrt{normX}$
  $normY \leftarrow \sqrt{normY}$

  if $normX < 10^{-6}$ **then**
    $normX \leftarrow 10^{-6}$ ▷ Handle small norm values
  end if

  if $normY < 10^{-6}$ **then**
    $normY \leftarrow 10^{-6}$ ▷ Handle small norm values
  end if

  $cos\_sim \leftarrow dot/(normX \cdot normY)$
  $out[i \cdot K + j] \leftarrow 1.0 - cos\_sim$ ▷ Compute cosine distance
end for

return out

Algorithm 2 Approximate Nearest Neighbors using K-Means

Require: Data points matrix $A \in \mathbb{R}^{N \times D}$, Query vector $X \in \mathbb{R}^{D}$, Number of neighbors K
Ensure: Array of indices of the top-K nearest neighbors

$A\_gpu \leftarrow$ Convert A to GPU array (float32)
$X\_gpu \leftarrow$ Convert X to GPU array (float32)
Reshape X_gpu to $(1, D)$ if necessary

$K1 \leftarrow \lceil 0.8 \cdot K \rceil$ ▷ Number of closest centroids to consider
$K2 \leftarrow \lceil 2 \cdot K \rceil$ ▷ Number of points to retrieve per cluster

$labels \leftarrow$ Run k-means on A_gpu with K clusters ▷ Cluster assignment for each point
$centroids \leftarrow$ Empty matrix of shape (K, D)

for $k = 0$ to $K - 1$ **do**
  $cluster\_mask \leftarrow (labels == k)$

  if $cluster\_mask$ has any True values **then**
    $centroids[k] \leftarrow$ Mean of points where $cluster\_mask$ is True

  else
    $centroids[k] \leftarrow 0$ ▷ Handle empty clusters
  end if
end for

$nearest\_cluster\_indices \leftarrow$ Find $\min(K1, K)$ nearest centroids to X_gpu

$candidate\_indices\_list \leftarrow$ Empty list
$candidate\_vectors\_list \leftarrow$ Empty list

for each index i in $nearest\_cluster\_indices$ **do**
  $indices\_in\_cluster \leftarrow$ Indices of all points where $labels == i$

  if $indices\_in\_cluster$ is not empty **then**
    $candidate\_vectors \leftarrow A\_gpu[indices\_in\_cluster]$
    $k2 \leftarrow \min(|candidate\_vectors|, K2)$ ▷ Number of points to select
    $candidate\_local\_indices \leftarrow$ Find $k2$ nearest points in $candidate\_vectors$ to X_gpu

    Append $indices\_in\_cluster[candidate\_local\_indices]$ to $candidate\_indices\_list$
    Append $candidate\_vectors[candidate\_local\_indices]$ to $candidate\_vectors\_list$
  end if
end for

if $candidate\_indices\_list$ is empty **then**
  return Empty array
end if

$candidate\_indices \leftarrow$ Concatenate all arrays in $candidate\_indices\_list$
$candidate\_vectors \leftarrow$ Concatenate all arrays in $candidate\_vectors\_list$

$final\_k \leftarrow \min(|candidate\_vectors|, K)$
$top\_k\_local \leftarrow$ Find $final\_k$ nearest points in $candidate\_vectors$ to X_gpu
$top\_k\_indices \leftarrow candidate\_indices[top\_k\_local]$ ▷ Map to original indices

return $top\_k\_indices$

Algorithm 3 Approximate Nearest Neighbors using Locality-Sensitive Hashing

Require: Data points matrix $A \in \mathbb{R}^{N \times D}$, Query vector $X \in \mathbb{R}^{D}$, Number of neighbors K, Number of hash tables L, Number of hashes per table num_hashes
Ensure: Array of indices of the top-K nearest neighbors

$A\_gpu \leftarrow$ Convert A to GPU array (float32)
$X\_gpu \leftarrow$ Convert X to GPU array (float32)
Reshape X_gpu to $(1, D)$ if necessary

$bit\_values \leftarrow [2^0, 2^1, ..., 2^{num\_hashes-1}]$ ▷ For binary-to-integer conversion
$hyperplanes \leftarrow$ Random normal matrix of shape (L, num_hashes, D)

$dots\_all \leftarrow A\_gpu \cdot hyperplanes$ ▷ Matrix-tensor product
$codes \leftarrow (dots\_all > 0)$ ▷ Binary hash codes
$codes\_int \leftarrow \sum_{j=0}^{num\_hashes-1} codes[:, :, j] \cdot bit\_values[j]$ ▷ Integer hash values

$query\_dots\_all \leftarrow X\_gpu \cdot hyperplanes$
$query\_codes \leftarrow \sum_{j=0}^{num\_hashes-1} (query\_dots\_all > 0)[:, :, j] \cdot bit\_values[j]$
$query\_codes \leftarrow query\_codes[0]$

$candidate\_mask \leftarrow \exists l \in \{0, ..., L-1\} : codes\_int[:, l] = query\_codes[l]$ ▷ Match in any table
$candidates \leftarrow$ Indices where $candidate\_mask$ is True

if $candidates$ is empty **then**
  $candidates \leftarrow [0, 1, ..., N-1]$ ▷ Fallback to full search
end if

$A\_candidates \leftarrow A\_gpu[candidates]$
$dists \leftarrow distance\_func(A\_candidates, X\_gpu)$ ▷ Compute distances
$dists \leftarrow$ Flatten $dists$

$topk\_indices \leftarrow$ Find indices of K smallest values in $dists$
$final\_indices \leftarrow candidates[topk\_indices]$ ▷ Map to original indices

return $final\_indices$

Algorithm 4 Execution flow using the `Testing` class abstraction

$Testing \leftarrow$ Set configuration via `argparse`
$args \leftarrow$ Parse runtime args
$t \leftarrow$ Instantiate new $Testing$ instance

if $program\_entry$ **then**
  for each $(dist\_fn, [N, D, A, X, K])$ in t **do**
    $knn\_result \leftarrow t.run(our\_knn, [N, D, A, X, K])$
    $t.run(our\_kmeans, [N, D, A, X])$
    $t.run(our\_ann, [N, D, A, X, K], knn\_result)$ Recall rate comparator knn_result
    $t.run(our\_ann\_lsh, [N, D, A, X, K], knn\_result)$ Recall rate comparator knn_result
  end for
end if

Algorithm 5 Iterator logic defined by `Testing` class

$iter\_trial\_idx \leftarrow$ The current trials index
$iter\_dist\_idx \leftarrow$ The current distance function index
$iter\_run\_idx \leftarrow$ The current run for the given trial and distance function

if the set of runs for the trial and distance function have completed **then**
    $results \leftarrow benchmark.construct\_results()$
    $benchmark.reset()$ Reset internal collections
end if
if $iter\_trial\_idx$ is equivalent to length of trial collection **then**
    $disk \leftarrow$ Write results to a CSV file
    Yield $stop$ to callee
end if
$dist\_fn \leftarrow dist\_fn\_collection.get(iter\_dist\_idx)$
Assign $dist\_fn$ to global namespace
$[N, D] \leftarrow trials\_collection.get(iter\_trial\_idx)$
$K \leftarrow$10 nearest neighbours
$A \leftarrow$ load matrix file for $[D, N]$
$X \leftarrow$ load vector file for D
if there are still runs to execute per $iter\_run\_idx$ **then**
    $iter\_run\_idx \leftarrow iter\_run\_idx + 1$
else if there are still distance functions to trial per $iter\_dist\_idx$ **then**
    $iter\_dist\_idx \leftarrow iter\_dist\_idx + 1$
    $iter\_run\_idx \leftarrow 0$
else
    $iter\_dist\_idx \leftarrow 0$
    $iter\_run\_idx \leftarrow 0$
    $iter\_trial\_idx \leftarrow iter\_trial\_idx + 1$
end if
Yield $dist\_fn$, $[N, D, A, X, K]$ to callee

C VISUALISATION OF RESULTS FROM TASK 2

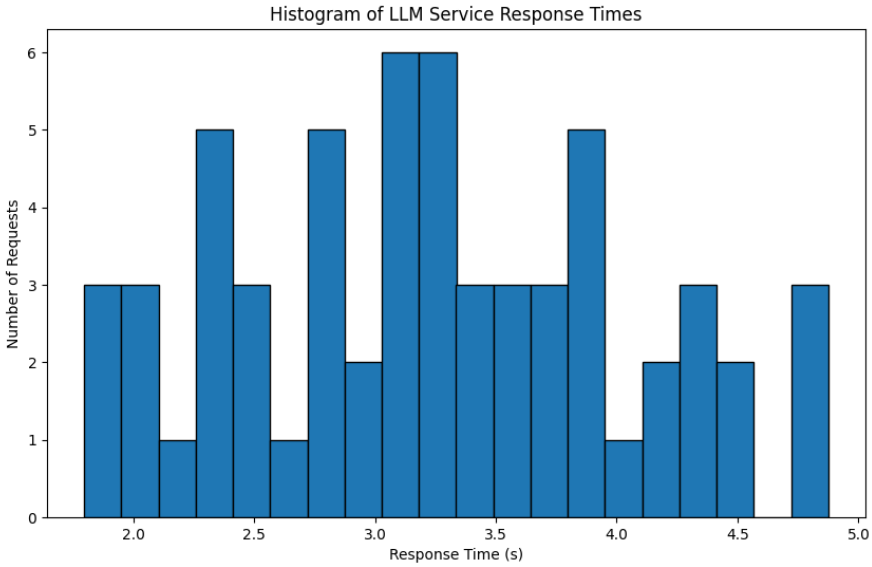

Figure 5: The distribution of response times from the load testing