Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining

Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin,
School of Computer Science, Peking University

https://www.usenix.org/conference/nsdi24/presentation/zhang-zili-pipelining

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Fast Vector Query Processing for Large Datasets Beyond GPU Memory with Reordered Pipelining

Zili Zhang Fangyue Liu Gang Huang Xuanzhe Liu Xin Jin

School of Computer Science, Peking University

Abstract

Vector query processing powers a wide range of AI applications. While GPUs are optimized for massive vector operations, today's practice relies on CPUs to process queries for large vector datasets, due to limited GPU memory.

We present RUMMY, the first GPU-accelerated vector query processing system that achieves high performance and supports large vector datasets beyond GPU memory. The core of RUMMY is a novel *reordered pipelining* technique that exploits the characteristics of vector query processing to efficiently pipeline data transmission from host memory to GPU memory and query processing in GPU. Specifically, it leverages three ideas: (i) *cluster-based retrofitting* to eliminate redundant data transmission across queries in a batch, (ii) *dynamic kernel padding* with *cluster balancing* to maximize spatial and temporal GPU utilization for GPU computation, and (iii) *query-aware reordering and grouping* to optimally overlap transmission and computation. We also tailor GPU memory management for vector queries to reduce GPU memory fragmentation and cache misses. We evaluate RUMMY with a variety of billion-scale benchmarking datasets. The experimental results show that RUMMY outperforms IVF-GPU with CUDA unified memory by up to 135×. Compared to the CPU-based solution (with 64 vCPUs), RUMMY (with one NVIDIA A100 GPU) achieves up to 23.1× better performance and is up to 37.7× more cost-effective.

1 Introduction

The breakthroughs in Deep Learning (DL) [1] enable unstructured data (e.g., images, videos, and audios) [2, 3] to be represented as high-dimensional feature vectors for serving a wide range of AI applications [4–12]. In particular, recent advancements in Large Language Models (LLMs) [13, 14] have catalyzed the emergence of a new generation of AI applications. However, LLMs only support a short-term memory (32k tokens limit for GPT-4 [15]). Vector databases [16–20] can provide persistence and long-term memory for LLMs. Consequently, Retrieval Augmented Generation (RAG) is applied to augment LLMs by dynamically retrieving relevant documents from a database during the generation process, thereby expanding LLMs' knowledge base and contextual understanding. Specifically, RAG first converts personal or organizational documents into vectors to build a vector database. When users post a question, it first issues a *vector query* to identify a set of documents that may contain the answer (i.e., find the most similar vectors from a vector database for a given vector). These documents, along with the original question, are then fed into an LLM, which analyzes the text information and returns the final answer. Additionally, vector query processing is also adopted in a wide range of applications, such as recommendation systems [4–6, 21], recognition [7–9, 22], and information retrieval [10–12].

With the explosive growth of the dataset scale, vector query becomes a performance bottleneck for AI applications. To reduce query overhead, GPUs, optimized for massive vector operations, are a natural choice to process vector queries. This is exemplified by early GPU-based vector query processing systems that load vector datasets into GPU memory [23]. A key problem is that they cannot support large datasets due to limited GPU memory. The explosion of unstructured data [3] and the increasing adoption of DL in production [4–9, 21, 22] make this problem particularly acute. Even high-end GPUs like NVIDIA H100 and A100, with tens of GB of memory, fall short of today's large vector datasets [24, 25] containing billions of items with a memory footprint of hundreds of GB. Thus, CPU-based solutions [16, 19, 23] are still the de facto choice for billion-scale datasets in production.

Conceivably, GPU memory can be augmented with host memory to support large datasets. A straightforward solution is to divide the dataset into multiple parts in host memory. Each part can be transmitted into GPU memory in rotation to process vector queries. This solution performs transmission and computation sequentially and cannot well utilize GPU resources. Another possible solution is to integrate a GPU-based solution with CUDA unified memory [26], which automatically handles GPU memory swapping and supports parallel transmission and computation. As CUDA unified memory is unaware of vector queries, this solution would incur massive GPU memory page faults, leading to low performance.

We present RUMMY, the first GPU-accelerated vector query processing system that (i) supports large vector datasets beyond GPU memory, and (ii) achieves higher performance and is more cost-effective than CPU-based solutions. The core of RUMMY is a novel *reordered pipelining* technique that

exploits the characteristics of vector query processing to efficiently pipeline data transmission from host memory to GPU memory and query processing in GPU.

Three primary technical challenges must be overcome to realize RUMMY, each pertaining to a different aspect: transmission, computation, and pipelining. First, redundant transmission occurs when processing vector queries in a batch. Due to limited GPU memory, the same data subset is transmitted and evicted repeatedly for different queries in the same batch. Leveraging the fact that each subset's processing is independent, we design *cluster-based retrofitting* to restructure the query plan for each vector query and *completely* eliminate redundant transmission.

Second, the Streaming Multiprocessors (SMs) in the GPU suffer from low utilization due to pipelining-caused load imbalance. Specifically, pipelining segregates the query processing into groups, each corresponding to a GPU kernel. A kernel comprises a grid of thread blocks, and each block is executed within an SM. A kernel for a small group is launched with a small grid of thread blocks, resulting in idle SMs (i.e., spatial underutilization). Besides, the discrepancy between the size of different data parts causes stragglers—some thread blocks execute longer than others (i.e., temporal underutilization). To maximize utilization, we introduce *dynamic kernel padding* with *cluster balancing*. Cluster balancing equalizes data part sizes offline. Dynamic kernel padding fills idle SMs with padding thread blocks online to enhance GPU occupancy.

Third and most importantly, directly applying pipelining suffers from limited overlapping between transmission and computation, and has high pipelining overhead due to frequent API invocations and synchronizations. The problem is exacerbated by that the pipelining plan can only be decided at runtime when queries arrive, as the quality of a particular plan is query-dependent. We design *query-aware reordering and grouping*, which dynamically reorders the query plan and divides the plan into groups with two algorithms. The reordering algorithm hides small transmission operations with large computation operations as many as possible to maximize the overlapping. The grouping algorithm finds the best tradeoff between pipelining efficiency and pipelining overhead. The two algorithms are lightweight with negligible runtime overhead. We prove that the two algorithms achieve optimal pipelining performance, respectively.

In addition, we also tailor the GPU memory management for vector query processing. RUMMY pre-allocates GPU memory and re-allocates the memory to each transmission task, without involving the general-purpose but costly GPU memory manager. We also exploit the vector query processing to reduce GPU memory fragmentation and cache misses.

We remark that the runtime *reordered pipelining* technique in RUMMY is distinct from canonical pipelining techniques adopted in microprocessor architectures [27, 28], DL systems [29, 30], and task schedulers [31, 32]. Those techniques rely on dependent and deterministic execution flows. For instance, DL models have a pre-defined layer-by-layer computation graph, which enables solutions like PipeDream [30] and PipeSwitch [29] to compute a pipelining plan *offline*. Besides, to ensure correctness, the deterministic computation graph cannot be *reordered*. In contrast, a vector query plan is *non-deterministic* and can be executed in any order since the processing of each subset is *independent*. The key novelty of RUMMY is that it leverages the independent and non-deterministic nature of vector query processing to dynamically decide the pipelining plan with *reordering* at *runtime*, and achieves optimal parallelism between transmission and computation.

In summary, we make the following contributions.

- We present RUMMY, to the best of our knowledge, the first GPU-accelerated vector query processing system that achieves high performance and supports large vector datasets beyond GPU memory.
- We introduce a new runtime *reordered pipelining* technique, which leverages *cluster-based retrofitting*, *dynamic kernel padding* with *cluster balancing*, and *query-aware reordering and grouping* to efficiently pipeline data transmission and query processing.
- We implement a RUMMY prototype. The evaluation shows that RUMMY outperforms IVF-GPU with CUDA unified memory by up to 135×. Compared to the state-of-the-art CPU solution (with 64 vCPUs), RUMMY (with one NVIDIA A100 GPU) achieves up to 23.1× better performance and is up to 37.7× more cost-effective.

2 Background and Motivation

In this section, we first introduce the background of vector query processing. We then describe GPU-based solutions to accelerate vector query processing. Finally, we summarize the challenges to support billion-scale datasets for GPU-based vector queries, which motivate the design of RUMMY.

2.1 Vector Query Processing

DL models convert unstructured data into high-dimensional feature vectors to serve applications [4–12, 21, 22]. These vector datasets [24, 25] are utilized to construct a vector database [16–20] as the persistence and long-term memory. Specifically, RAG leverages these datasets to enhance the quality of LLM-based AI applications. For instance, TEXT1B [24] contains one billion data entries from both textual and visual modalities, which can be used to enhance multimodal LLMs. Vector queries are referred to as the memory retrieval part in RAG and are widely used in emerging LLM-based AI applications [33–35].

Specifically, a vector query is to find the top-k nearest neighbors (KNN) in a vector dataset that are most similar to the given vector. KNN returns the exact top-k results and requires searching on the entire dataset. KNN becomes impractical for large datasets due to high query latency. Approximate top-k nearest neighbor (ANN) search trades query accuracy

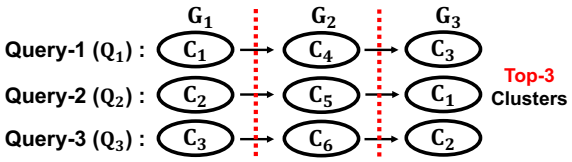


Figure 1: Query plan for a batch of three queries.

for query latency. Previous works [36] show that ANN can achieve 99% accuracy while only searching on 1% of the dataset. Even for accuracy-sensitive scenarios, ANN can meet the requirements of applications and outperform KNN by hundreds of times. Since DL models inherently introduce errors when generating vectors, ANN is widely used in existing vector databases for large datasets [16, 18, 19, 37–40].

The basic idea of ANN search is to build an index structure [41–44] to sample a subset of the dataset for answering a vector query. There are two representative ANN indexes, inverted file index (IVF) [41,45,46] and graph index [42,47,48]. While graph index is more accurate, it is more expensive to build and maintain due to its large memory footprint and building time. When confronting billion-scale datasets, IVF is proved to be more efficient and accurate than the state-of-the-art graph index [48] with the same memory consumption according to recent work [46]. Thus, IVF is more suitable for very large datasets. While IVF has many variants, their general workflow is similar. At offline, IVF trains a list of clusters ($\{C_1, C_2, ...C_l\}$) by k-means clustering [49], and each vector is assigned to the closest cluster. Online, it performs pairwise comparison between the query vector and each vector in the top-n closest clusters to produce the approximate top-k nearest neighbor vectors. The number of sampled clusters n is a configurable parameter to make a tradeoff between accuracy and latency.

For example, Figure 1 shows a query plan of the batch of three queries ($\{Q_1, Q_2, Q_3\}$). IVF selects top-3 nearest clusters (i.e., $n = 3$) for each query, e.g., $\{C_1, C_4, C_3\}$ for Q_1. Q_1 is compared with each vector in the three clusters, in which the top-k nearest vectors are returned as the result.

2.2 GPU Acceleration

Vector query processing is compute-intensive and involves massive vector operations. GPUs are a natural choice to accelerate vector processing [23]. The pairwise distance computation between high-dimensional vectors is a good fit for the GPU architecture. A detailed comparison between CPUs and GPUs on IVF and graph index in terms of performance and cost is listed in Appendix A.2, which demonstrates the benefits of GPU-based vector query processing systems.

Vector query processing in GPUs. A GPU consists of many SMs, each including numerous CUDA cores for extensive vector operations. Such hardware architecture involves a hierarchical parallel computation model [50]: each computation consists of multiple CUDA kernels; each CUDA kernel includes a grid of thread blocks; and each block comprises a number of threads. A block of threads can be executed simultaneously within one SM, where each thread is executed on a CUDA core. To process a vector query in a GPU, the computation of the query on one cluster (e.g., $Q_1 \rightarrow C_1$ in Figure 1) is a thread block. The computations of different queries on their corresponding clusters (e.g., $\{Q_1 \rightarrow C_1, Q_2 \rightarrow C_2, Q_3 \rightarrow C_3\}$) form a grid of thread blocks, i.e., a CUDA kernel.

Limitation: GPU memory capacity. Existing GPU-based vector query systems [23] load the entire dataset in the GPU. Today's large vector datasets [24, 25] contain billions of vectors, which have a memory footprint of hundreds of GB. GPU memory is too limited compared to large datasets. For example, NVIDIA A100 GPUs are equipped with up to 80GB of memory, while the size of the TEXT1B dataset [24] is 750GB. It requires at least ten NVIDIA A100 GPUs to store the TEXT1B dataset. Using multiple GPUs causes a mismatch between GPU compute resources and GPU memory resources. In particular, as we have explained in §2.1 and Figure 1, ANN search only performs vector operations on a small subset of the dataset, which exacerbates this mismatch and causes low utilization of expensive GPU resources.

The natural idea is to expand GPU memory with host memory. For example, the dataset is stored in host memory and is divided into multiple parts, each of which can fit GPU memory. To process a query, each part is transmitted from host memory to GPU memory *in rotation*. After iterating over all parts, the intermediate results of each part are aggregated to produce the final result. This solution has a long transmission time, and because it performs transmission and computation sequentially, it has low GPU utilization.

Another strawman solution is to integrate existing GPU-based query systems with CUDA unified memory [26]. CUDA unified memory automatically handles data transmission at runtime, and performs parallel transmission and computation to fully utilize the GPU copy engine and kernel engine. However, it is a general memory swapping technique that is unaware of vector queries. It incurs massive GPU memory page faults for data transmission. Besides, the overlapping between transmission and computation is also limited as it is agnostic of vector queries. Therefore, applying CUDA unified memory introduces a large gap from the ideal performance.

2.3 Challenges

Opportunity: pipelining transmission and computation. We can pipeline transmission and computation to increase GPU utilization and improve performance. Specifically, we divide the query processing into groups, e.g., $\{G_1, G_2, G_3\}$ in Figure 1. The transmission of one group can be pipelined with the computation of the preceding group. Figure 2(a) shows how to pipeline the three groups in Figure 1, where PCIe represents data transmission from host to GPU memory over PCIe, and GPU represents computation in GPU. The transmission of G_2 is pipelined with the computation of the

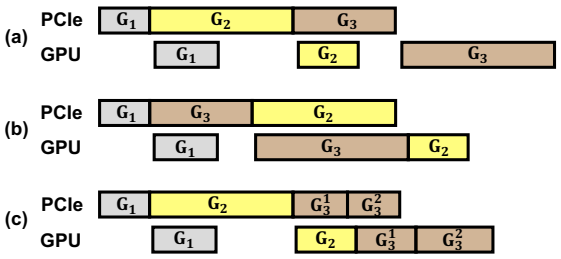


Figure 2: Examples for the pipelining plans (ignore data reuse).

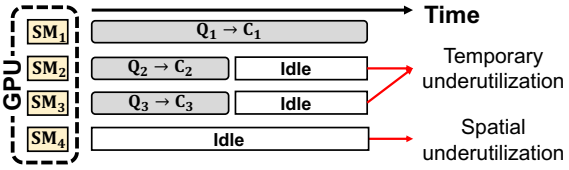


Figure 3: Spatial and temporal GPU underutilization.

preceding group G_1, partly hiding the transmission overhead. We summarize three challenges to exploit this opportunity and achieve optimal pipelining efficiency, which focus on three system aspects: transmission, computation, and pipelining.

Challenge 1: cross-query redundant transmission. The first challenge is that processing a batch of vector queries introduces redundant transmission. As Figure 1 shows, if the GPU memory capacity equals three clusters and the execution order is $[G_1, G_2, G_3]$. Initially, the GPU is empty. The system transmits $G_1 = \{C_1, C_2, C_3\}$ to the GPU, which fills up the GPU memory. Next, to transmit $\{C_4, C_5, C_6\}$ for G_2, it first evicts $\{C_1, C_2, C_3\}$ and then starts the transmission. As for G_3, it clears the GPU memory again and then transmits $\{C_1, C_2, C_3\}$ back to GPU memory. In total, the order $[G_1, G_2, G_3]$ transmits nine clusters, among which C_1, C_2 and C_3 are transmitted twice each. Alternatively, swapping the order between G_2 and G_3 only needs to transmit six clusters. In this case, the system reuses the data $\{C_1, C_2, C_3\}$ for G_3 after processing G_1. Reordering G_2 and G_3 eliminates the redundant transmission.

Challenge 2: spatial and temporal GPU underutilization. The second challenge is the SMs in the GPU have low utilization due to the load imbalance of thread blocks. The computation of G_1 in Figure 1 corresponds to a kernel that contains three thread blocks (i.e., $\{Q_1 \to C_1, Q_2 \to C_2, Q_3 \to C_3\}$). $Q_i \to C_j$ represents the thread block of the computation of query Q_i on cluster C_j. Each thread block is executed within one SM and cannot be migrated to other SMs [50]. As shown in Figure 3, when processing G_1, there are only three thread blocks to run, if the GPU has four SMs, then one SM (i.e., SM_4) is idle, causing *spatial underutilization*. Besides, because the clusters have different sizes, the thread blocks for larger clusters take a longer time to run, i.e., stragglers. As shown in Figure 3, SM_2 and SM_3 are idle after completing their thread blocks, leading to *temporal underutilization* as they wait for SM_1 to finish. In summary, spatial underutilization stems from a mismatch between the number of SMs and thread blocks, while temporal underutilization is caused by cluster heterogeneity.

Challenge 3: transmission and computation overlapping. The third challenge is to maximize the overlapping between transmission and computation in the pipeline. The overlapping is affected by both the ordering and granularity of the groups in the pipeline. We illustrate each of them in Figure 2, and we do not consider data reuse in this example. Figure 2(a) shows a primitive pipelining plan based on the execution order $[G_1, G_2, G_3]$. First, as shown in Figure 2(b), if we reorder G_2 and G_3, then the transmission of G_2 is completely hidden by the computation of G_3, which increases the overlapping. Second, as shown in Figure 2(c), dividing G_3 into two smaller groups, G_3^1 and G_3^2, also improves the overlapping, because finer-grained grouping can hide more transmission operations with computation operations. However, finer-grained grouping also introduces more system overhead due to frequent API invocations and synchronizations.

The challenge is exacerbated by the runtime nature of the problem. The pipelining plan depends on the input vector queries, as different queries process different clusters, which affects the ordering and grouping decisions of the pipeline. Let N be the number of thread blocks. The search space of exhaustive search contains $O(N! \times 2^N)$ choices, as there are $O(N!)$ ordering cases and $O(2^N)$ grouping cases. It is challenging to find the best plan among these choices. In addition, the system also needs an accurate *profiler* to precisely estimate the transmission and computation time for a given group at runtime.

3 RUMMY Overview

We present RUMMY, a GPU-accelerated vector query processing system to support large vector datasets beyond GPU memory. RUMMY exploits the characteristics of vector queries to achieve fast and cost-effective query processing. RUMMY achieves so by pipelining transmission and computation through a novel *reordered pipelining* technique. This technique eliminates redundant data transmission (§4.1), maximizes GPU utilization (§4.2), and finds out optimal pipelining plans with negligible runtime overhead (§4.3). Here we provide a brief overview of RUMMY as Figure 4 shows. RUMMY consists of an offline part and an online part.

Offline. RUMMY first builds a primitive IVF index for the dataset, which can be extended into many variants of IVF. IVF divides the vectors into a few clusters. RUMMY extends the index building with *cluster balancing* to alleviate the straggler problem (§4.2). The dataset is stored in host memory and is transmitted to the GPU memory over PCIe online under the control of the GPU *memory management system* (§4.4). RUMMY also builds a *profiler* offline to measure the computation and transmission time (§4.3.1) for online decision making.

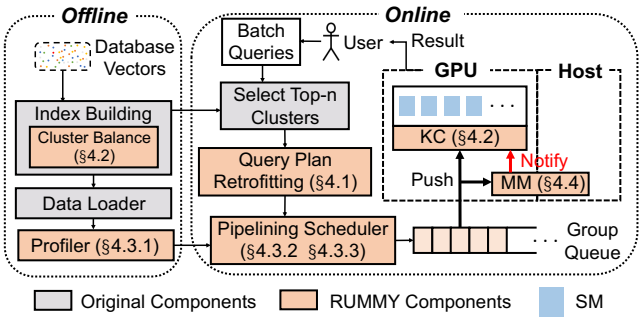


Figure 4: RUMMY overview.

Symbol	Description
C_i	The i_{th} cluster
Q_i	The i_{th} query vector
G_i	The i_{th} group
$\{C_1, C_2 ...\}$	A set of original clusters
$[C_1, C_2 ...]$	The execution order of original clusters
$\{B_1, B_2 ...\}$	A set of balanced clusters
$[B_1, B_2 ...]$	The execution order of balanced clusters
ρ	The fixed size of balanced clusters (the number of vectors)
$T(G)$	The transmission (time) of group G
$E(G)$	The computation (time) of group G
$Q_i \rightarrow B_j$	The computation (thread block) of Q_i on B_j

Table 1: Key notations in the design.

Online. At runtime, RUMMY processes vector queries that arrive at the system. RUMMY has the following three major components in the query runtime.

Query plan. After the top-n closest clusters of each query are determined through IVF, RUMMY retrofits the query plan (i.e., queries to clusters as shown in Figure 1) of each query, which eliminates the redundant data transmission (§4.1).

Pipelining scheduler. The runtime *pipelining scheduler* receives the retrofitted query plan. Based on the prediction time provided by the *profiler*, the scheduler reorders the plan with a greedy algorithm, which finds the optimal order that minimizes the total time on per-cluster granularity (§4.3.2). After the order is determined, the scheduler groups the query plan with a dynamic programming algorithm, which finds the best tradeoff between pipelining efficiency and pipelining overhead (§4.3.3). Each group is pushed into a global *group queue* for transmission and computation.

GPU Runtime. The GPU runtime of RUMMY consists of two components: *kernel controller (KC)* and GPU *memory management system (MM)*. The two components maintain two local group queues respectively. They pull tasks from the global group queue. MM starts transmission immediately as long as its local group queue is not empty (§4.4). When finishing the transmission of a group G_i (i.e., T(G_i)), it notifies KC to set the computation of G_i (i.e., E(G_i)) executable and pops the next task for transmission. KC pops the task $E(G_i)$ and executes it if the kernel engine is idle and $E(G_i)$ is executable. Besides, when launching $E(G_i)$, KC will estimate whether $E(G_i)$ is able to saturate all SMs in the GPU. If not, KC dynamically pads the kernel with more thread blocks to utilize idle SMs (§4.2). After finishing the computation of all groups in a batch, KC returns the final result to the user.

4 RUMMY Design

In this section, we present the design of RUMMY, i.e., *reordered pipelining* that includes three main techniques. The first is *cluster-based retrofitting* to eliminate redundant transmission (§4.1). The second is *dynamic kernel padding* with *cluster balancing* to maximize GPU utilization (§4.2). The third is *query-aware reordering and grouping* to overlap transmission and computation (§4.3). Besides, we describe how to tailor GPU memory management for vector query processing (§4.4) based on IVF. The key notations are listed in Table 1.

4.1 Cluster-based Retrofitting

The key observation behind our idea, namely cluster-based retrofitting, is that the processing of each subset is independent of vector query processing. This observation allows us to retrofit the query plan of each query without changing its correctness. We retrofit the query plan from two aspects, *intra-batch* and *inter-batch*, to completely eliminate the cross-query redundant transmission.

Intra-batch. Figure 1 shows an original vector query plan. The order, $[G_2, G_1, G_3]$, eliminates the three redundant transmission (§2.3). A strawman approach might attempt to find the optimal order by enumerating all possibilities. It is impractical at runtime due to the problem's combinatorial nature.

RUMMY's solution, namely cluster-based retrofitting, is based on the key insight: the optimal number of cluster transmissions is no smaller than the number of involved clusters. In Figure 1, six clusters $\{C_1, C_2 ... C_6\}$ necessitate a minimum of six transmissions, where each cluster is transmitted at least once. As such, our goal is to transmit each involved cluster only once. Figure 5(a) depicts a retrofitted query plan based on Figure 1. We represent this process by a matrix M. $M[i, j] = 1$ means that C_j is involved in the i-th query Q_i, otherwise $M[i, j] = 0$. When transmitting a cluster C_j, the retrofitted plan instantly processes the corresponding queries $\{Q_i ...\}$ for $i \in \{1, 2 ... BS\}$ and $M[i, j] = 1$, with BS as the batch size. For instance, it processes Q_1 and Q_2 immediately after transmitting C_1. In this way, cluster-based retrofitting completely eliminates the redundant transmission. Besides, the retrofitting only traverses the original query plan *once*, resulting in a negligible linear time complexity.

Inter-batch. In intra-batch retrofitting, the cluster order is irrelevant, and every permutation of $[C_1, C_2 ... C_6]$ transmits six clusters. Conversely, inter-batch considers order. Let GPU memory accommodate two clusters C_5 and C_6 initially. Then it additionally eliminates two redundant transmissions

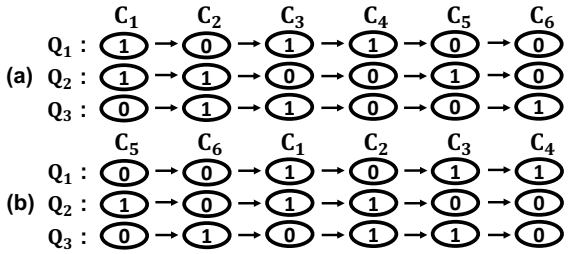


Figure 5: Retrofit the vector query plan.

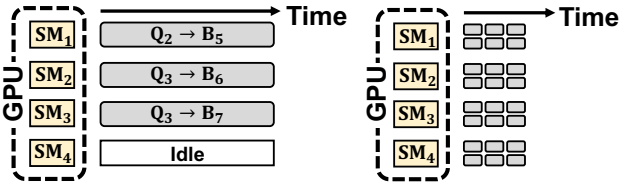
by moving C_5 and C_6 to the front, i.e., $[C_5, C_6, C_1 ..., C_4]$ as shown in Figure 5(b). To maximize data reuse from preceding batches, RUMMY prioritizes moving the clusters already present in the GPU to the forefront of the execution order. Moreover, this mechanism works together with the *reordering* algorithm, which will be described in §4.3.2.

4.2 Dynamic Kernel Padding with Cluster Balancing

Cluster balancing. The straggler causes temporal GPU underutilization by delaying the entire kernel. To address this issue, we design a cluster balancing technique that equilibrates the sizes of various clusters *offline*. Specifically, a cluster C_i is divided into a set of balanced clusters $\{B_{i_1}, B_{i_2} ...\}$. Each balanced cluster B_j is in equal size, denoted by ρ. The detailed balancing algorithm is summarized in Appendix A.3. We emphasize that this technique balances the clusters in *host memory* offline, and the balanced cluster is transmitted into the GPU at runtime to answer the queries. After eliminating the discrepancy between different clusters, the straggler thread block is split into multiple balanced blocks, which maximizes temporal GPU utilization. For instance, suppose that the group G consists of two clusters, C_5 and C_6, in Figure 5(b). C_5 is split into $\{B_5\}$, and C_6 is split into $\{B_6, B_7\}$. The computation of G comprises three balanced thread blocks: $\{Q_2 \to B_5, Q_3 \to B_6, Q_3 \to B_7\}$. Each thread block completes in the same time as illustrated by Figure 6(a). However, the spatial GPU underutilization still exists (SM_4 is always idle).

Dynamic kernel padding. Runtime kernel padding is a technique to address spatial GPU underutilization. An intuitive solution is to process different groups simultaneously through GPU space sharing and provision idle SMs to the subsequent group of computation. This principle resembles the kernel padding technique [51] in DNN schedulers, which shares the GPU with various jobs. However, it is infeasible in the context of pipelining, as the subsequent group remains unprepared until its transmission is finalized. Two groups cannot be executed simultaneously. We design a dynamic kernel padding technique to pad one individual group of kernel execution.

The thread block, $Q_i \to B_j$, receives Q_i and a pointer P_j to B_j as inputs and returns the top-k nearest vectors as the intermediate result. The inputs and outputs are both resident in GPU memory. Suppose the size of each balanced cluster is ρ and the dimension of each vector is d. The balanced cluster is divided into two parts with two pointers: P_j and $P_j + \rho \times d/2$. Thus, $Q_i \to B_j$ is divided into two smaller thread blocks, namely kernel padding. The split number is decided by *Kernel Controller* according to the number of SMs and the query plan to achieve 100% SM utilization and high GPU occupancy [52]. As shown in Figure 6(b), the split number is 8 and the number of padded blocks is 24. Every SM is utilized to achieve 100% spatial utilization. As for GPU occupancy [52], two blocks run on each SM concurrently to further improve the computation efficiency. Since dynamic kernel padding only operates the data pointers, its runtime overhead is negligible.



(a) Cluster balancing. (b) Dynamic kernel padding.

Figure 6: Dynamic kernel padding with cluster balancing.

A natural question is why not solve the straggler problem through kernel padding online only, given that straggler blocks can easily be split at runtime. The reason is that computing the split strategy incurs extra runtime overhead, as different clusters need different split numbers. Thus, our design shifts some work offline to simplify the online decision.

4.3 Pipelining Scheduler

As we describe in §2.3, it is impossible to enumerate every pipelining plan to find out the optimal solution at runtime. We break the entire problem into two individual tractable subproblems and design *query-aware reordering* and *grouping* algorithms that reorder the retrofitted query plan and divide the plan into groups. The algorithms compute optimal solutions for each sub-problem, and the combination of them achieves near-optimal results empirically (§6).

4.3.1 Profiler

Transmission profiler. The goal of the transmission profiler is to measure $T(G_i)$ for a given group G_i. The transmission time from host memory to GPU memory can be divided into two components: the propagation time and the overhead of API invocation. Let G_i consist of m balanced clusters. The propagation time is directly proportional to m. The overhead of API invocation can be estimated as a fixed value. Therefore, $T(G_i)$ is estimated by the following formula: $T(G_i) = a \times \rho \times m + b$, where a and b are parameters fitted offline, and ρ is the size of balanced clusters (§4.2). Querying the transmission profiler for $T(G_i)$ only costs constant time.

Computation profiler. The goal of the computation profiler is to measure $E(G_i)$. We define the computation quantity as $Com = \rho \times SIZE(G_i)$, where ρ is the size of balanced clusters and $SIZE(G_i)$ is the number of elements, whose value is 1, in the retrofitted and balanced query plan of G_i (i.e., the matrix

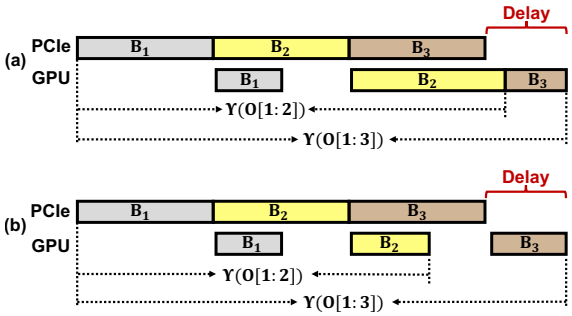


Figure 7: Two cases of the recurrence formula in formulation.

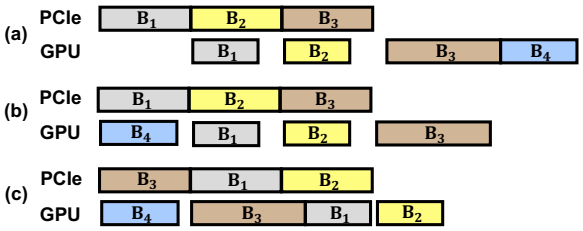


Figure 8: Examples for the reordering algorithm.

in Figure 5 with balanced clusters). Apparently, the degree of parallelism (i.e., the number of thread blocks) and *Com* dominate the computation time. Therefore, we map *Com* with different block numbers to computation time offline. For the intermediate point between the map keys, we use percentile-wise linear interpolation to estimate its value. Similarly to the transmission profiler, querying the unordered map of the computation profiler online also only costs constant time.

4.3.2 Reordering

Problem formulation. The first sub-problem is to reorder the clusters to maximize the overlapping between transmission and computation (i.e., minimize the total time). Let the set of the involved balanced clusters be $\{B_1, B_2 ... B_m\}$, and $[B_1, B_2 ... B_m]$ represents an execution order. $[o_1, o_2 ... o_m]$ is a permutation of $[1, 2 ... m]$. Due to cluster balancing, each B_i is in equal size and introduces the same $T(B_i) = \eta$, unless B_i is already in the GPU from the preceding batch (i.e., $T(B_i) = 0$). The order $[B_{o_1}, B_{o_2} ... B_{o_m}]$ is represented as O, and $O[i:j]$ is a slice of O and is also a new order. $\Upsilon(O)$ is the total time based on the order O, where $O = [B_{o_1}, B_{o_2} ... B_{o_m}]$.

Based on these definitions, we have the following recurrence formula to calculate $\Upsilon(O)$:

$$\Upsilon(O[1:i]) = max(\ \Upsilon(O[1:i-1]) + E(B_{o_i}),\ \sum_{k=1}^{i} T(B_{o_k}) + E(B_{o_i})\ ). \quad (1)$$

The entire computation of an arbitrary order is always later than its transmission, which introduces a *delay*. Figure 7 shows the delay of three clusters (i.e., $i = 3$). The first formula in the *max* function introduces that $T(B_{o_i})$ (i.e., the transmission of the last cluster in $O[1:i]$) is not able to compensate for the delay of $O[1:i-1]$ in Figure 7(a). Figure 7(b) demonstrates $T(B_{o_i})$ is able to compensate for the delay, causing the second formula in the *max* function. Consequently, the maximal case is the final result, i.e., $\Upsilon(O[1:i])$.

In summary, the reordering problem is formulated as the following optimization problem, where the objective is to minimize the total time $\Upsilon(O)$ based on Formula 1.

$$\begin{aligned} &Min. \quad \Upsilon(O). \\ &s.t. \quad O = [B_{o_1}, B_{o_2} ... B_{o_m}], \\ &\qquad [o_1, o_2 ... o_m] = permutation([1, 2 ... m]). \end{aligned} \quad (2)$$

To solve this optimization problem, one can use a search algorithm with pruning or leverage existing solvers for optimization problems. However, such solutions are complex and infeasible due to the vast search space ($m!$ possibilities), the recursive nature of the objective function, and the discrete constraint (a permutation array). Designing an algorithm capable of rapidly finding the optimal order during runtime poses a substantial challenge.

Insights. We leverage two insights to address the challenge. As described in the problem formulation, the transmission time of each balanced cluster is either a fixed value, η, or zero. However, the computation time of each balanced cluster $E(B_i)$ varies since each cluster relates to a different number of queries. Such characteristics provide an opportunity to simplify the problem with two *insights*. To demonstrate the insights, we give an example of four balanced clusters in Figure 8(a), and the execution order is $[B_1, B_2 ... B_4]$ where B_4 is already in the GPU (i.e., $T(B_4) = 0$).

The first insight is that moving a cluster with a transmission time of zero to the front increases the overlapping. In the example of Figure 8, $T(B_4) = 0$. When moving B_4 to the front, i.e., $[B_4, B_1, B_2, B_3]$, the overlapping increases as $T(B_1)$ is hidden by $E(B_4)$ as shown in Figure 8(b).

The second insight is that moving a cluster with a large computation time forward increases the overlapping. In the example of Figure 8, B_3 has the largest computation time. Moving B_3 to the second, i.e., $[B_4, B_3, B_1, B_2]$, further increases the overlapping as shown in Figure 8(c). The intuition is that $E(B_3)$ compensates the non-overlapping area between $E(B_i)$ and $T(B_i)$ for $i \in \{1, 2\}$.

Algorithm. Based on the two insights, we design a greedy algorithm that reorders the clusters to maximize the overlapping and minimize the total time, and we prove that the greedy algorithm is optimal. Algorithm 1 shows the pseudo-code. The function *FindOptOrder* finds the optimal execution order based on the greedy policy: moving the balanced clusters with zero transmission time and large computation time

Algorithm 1 Optimal greedy reordering algorithm

```
1:  function FINDOPTORDER({B_1, ..., B_m})
2:      opt_order ← ∅
3:      for i = 1 → m do
4:          if T[B_i] == 0 then
5:              B_i.priority ← +∞
6:          else
7:              B_i.priority ← E[B_i]
8:          opt_order.append(B_i)
9:      Sort opt_order in descending order based on B_i.priority
10:     return opt_order
```

to the front. It takes the m balanced clusters. It uses *opt_order* to store the execution order and initializes this variable with none (line 2). It iterates over the balanced clusters, and assigns a priority to each cluster based on the greedy policy (line 3-8). It then sorts *opt_order* in descending order based on the priority of each cluster (line 9). Given m balanced clusters, the critical path is the sort operation. Therefore, the time complexity of Algorithm 1 is $O(m \log m)$, which enables RUMMY to quickly find the optimal order at runtime. We have the following theorem for the reordering algorithm.

Theorem 4.1 *Algorithm 1 finds the optimal execution order of balanced clusters that minimizes the total time for the pipelining on per-cluster granularity.*

The main idea of the proof is to show that any transformation of the optimal execution order does not decrease the total time. The proof is in Appendix A.1. Since the reordering algorithm moves the clusters with zero transmission time to the front, it solves the inter-batch retrofitting problem described in §4.1.

4.3.3 Grouping

The second sub-problem is to group the balanced clusters after the order is determined. The basic way for pipelining is to pipeline on per-cluster granularity, i.e., transmitting and computes cluster by cluster. However, such fine-grain pipelining introduces two sources of pipelining overheads: the overhead of the frequent invocations and the synchronization overhead between each group of transmission and computation. Another way for pipelining is to group all clusters together. It eliminates the pipelining overhead, but there would be no overlapping between transmission and computation at all.

We employ a dynamic programming algorithm, similar to previous works [29, 30], to find the optimal grouping plan in the context of pipelining. To further reduce the search space, we use a heuristic pruning method, maintaining a global variable to record the best time in the current search space. During the search tree traversal, the lower bound of a node's subtree is calculated by completely overlapping leftover transmission and computation while ignoring the pipelining overhead. If such lower bound is greater than the best time, the subtree is pruned. The time complexity of the dynamic programming algorithm is polynomial, and the algorithm with the pruning heuristic enables RUMMY to quickly find the optimal grouping plan at runtime.

4.4 GPU Memory Management

Existing GPU-based vector query processing systems lack support for runtime GPU memory management and load the entire dataset into the GPU offline. RUMMY provides runtime GPU memory management tailored for vector query processing. It leverages GPU's native interfaces, like `cudaMalloc` and `cudaFree` in NVIDIA GPUs, to pre-allocate the entire GPU memory to RUMMY at system startup and manage the memory internally. This avoids frequent invocations to GPU's native interfaces. RUMMY handles data transmission tasks from its local task queue (§3), and notifies the kernel controller upon completion. Below, we describe the GPU memory layout and page replacement policy to reconcile vector query processing with the limited GPU memory capacity. RUMMY pins host memory to further reduce the transmission time.

GPU memory layout. The entire GPU memory is treated as a consecutive memory space in RUMMY. RUMMY's memory layout is able to eradicate both internal memory fragmentation and external memory fragmentation. RUMMY allocates GPU memory on page granularity for transmission tasks, enabling clusters to be stored in discontinuous space. RUMMY returns several discontinuous pages for the allocation of a transmission task. Thus, each free memory fragment (i.e., a free page) can be allocated for any task, which minimizes the external fragmentation issue. Besides, a large page size introduces internal memory fragmentation while a small page size introduces extra overhead of massive paging operations. Due to cluster balancing (§4.2), the cluster size is fixed. Setting the page size to the cluster size ensures that each page is fully utilized and the overhead of paging is minimized, thereby easily solving internal fragmentation issues.

GPU page replacement policy. When the GPU memory is full, RUMMY has to evict a page (balanced cluster) for the subsequent allocation. RUMMY's GPU page replacement policy accounts for both intra-batch and inter-batch to minimize the miss rate. For intra-batch, RUMMY traverses the local task queue of the current batch, and pins the intersection clusters between the future involvement in the current batch and those already in the GPU, without evicting pinned clusters. As for inter-batch, we observe that certain clusters (hot) are referred to frequently, while others (cold) are occasionally referred to. RUMMY records the referred count of each cluster. At runtime, RUMMY evicts the cluster with the smallest counter if it is not pinned, which is akin to LFU [53]. There are some canonical page replacement policies, e.g., LRU. However, they are not well-suited to vector query processing. Specifically, LRU evicts the cluster that is least recently used. A vector query with a large batch size uses every cluster. LRU may replace the hot clusters, those accessed early in the batch, with cold clusters, those accessed late in the batch. This leads to a high

miss rate in the subsequent batch that predominantly relies on hot clusters (i.e., small batch size).

Host memory pinning. The operating system swaps host memory pages to the disk if this page is inactive for a certain time. This mechanism causes severe page faults during GPU runtime data transmission, as GPU can't directly transfer pages from the disk. To mitigate the problem, RUMMY uses page-locked host memory (i.e., pin memory) to store the clusters through `cudaMallocHost` in NVIDIA GPUs. The pinned memory, accessible directly by GPU, allows read/write operations with higher bandwidth compared to pageable memory allocated by standard `malloc`.

5 Implementation

We implement a system prototype of RUMMY with 12K lines of codes in CUDA and C++, and integrate it with Faiss [23], a state-of-the-art vector query processing system adopted by many vector databases, like Milvus [16], Zilliz [17] and AnalyticDB-V [19]. RUMMY can be integrated with any vector query system. We choose Faiss as it is the most widely used GPU-based vector query processing system, and is adopted in production like Meta. RUMMY is built on top of IVF index rather than graph index, because the former is proved to be more efficient on billion-scale datasets [46] with limited memory capacity. Moreover, graph index (e.g., HNSW) cannot be easily integrated into GPU due to its random access pattern. The code of RUMMY is open-source and is publicly available at https://github.com/pkusys/Rummy.

Kernel controller. The kernel controller executes GPU kernels. It learns the GPU hardware information (e.g., the number of SMs) by reading CUDA macros. We extend CUDA kernels in Faiss to dynamically pad the thread blocks according to the split number. A larger split number means a larger block number to saturate the GPU (SMs).

Memory management. RUMMY divides GPU memory into two areas: auxiliary and primary memory. The auxiliary memory stores the query vectors, the query plan, and the intermediate results. The primary memory holds the clusters of vector datasets. The auxiliary memory is managed as a stack, which orchestrates memory for temporary data created and deleted in order during query processing. The primary memory is organized as a heap to cache the clusters. Each page in the heap is either free or allocated to a cluster. RUMMY uses an AVL tree to manage these pages to quickly retrieve an allocated page or allocate a free page.

System optimizations. Faiss spawns one CUDA stream to process a vector query and synchronizes all operations by default. In contrast, RUMMY uses three CUDA streams: one for launching CUDA kernel, another for GPU-to-host memory transmission, and the third for host-to-GPU memory transmission. This trio of streams allows RUMMY to parallelize computation and two types of transmission in the pipeline.

Dataset	Dimensions	Database Vectors	Query Vectors	Distance	Memory Footprint
SIFT1B [25]	128	1B	10K	Euclidean	480 GB
DEEP1B [24]	96	1B	10K	Euclidean	361 GB
TEXT1B [24]	200	1B	100K	Angular	748 GB
SIFT40M [25]	128	40M	10K	Euclidean	31 GB
DEEP50M [24]	96	50M	10K	Euclidean	29 GB
TEXT30M [24]	200	30M	100K	Angular	32 GB
SIFT10M [25]	128	10M	10K	Euclidean	9 GB

Table 2: Datasets used in the evaluation.

RUMMY wraps up each stream with a host thread to simplify function invocations and metadata (e.g., AVL tree) changes. RUMMY groups concurrent requests into one batch. The concurrent requests (i.e., one batch) are primarily processed on the GPU. The host thread is responsible for generating the query plan and directing the GPU to process the batch.

6 Evaluation

In this section, we first use end-to-end experiments to demonstrate the overall performance improvements of RUMMY over existing GPU-based and CPU-based solutions on billion-scale datasets. Next, we use microbenchmarks to deep dive into RUMMY and show the effectiveness of each component in RUMMY under a variety of settings. As we discuss in §2.2, IVF is more suitable for billion-scale datasets than graph index schemes (e.g., HNSW). Thus, we focus on the IVF index in the evaluation. The baselines are also implemented based on Faiss.

6.1 End-to-End Experiments

Setup. All experiments are conducted on AWS. The end-to-end experiments use two types of AWS EC2 instances. One is p4d.24xlarge configured with eight NVIDIA A100 GPUs with 40 GB GPU memory each, 1152 GB host memory and PCIe 4.0×16. The GPU instance is used to compare RUMMY with existing GPU-based solutions (§2.2). While we only use *one* GPU, we use p4d.24xlarge because it is the only type of instance on AWS that has high-end GPU, A100. The other is x1.16xlarge configured with 64 vCPUs (Intel Xeon E7-8880) and 976 GB of host memory. The CPU instance is used to compare RUMMY with the existing CPU-based solution.

Workloads. Table 2 summarizes the benchmarking datasets, and the top three are used in end-to-end experiments. The datasets include SIFT1B [25], DEEP1B [24], and TEXT1B [24]. They consist of one billion database vectors and some query vectors. These datasets are standard benchmarks for vector database retrieval used by both academic and industry. They can also be integrated with LLMs to support retrieval augmented generation, e.g., TEXT1B (a cross-model dataset) can be used to enhance multimodal LLMs. The memory footprint of the 1B datasets is in the hundreds of GB range, which *oversubscribes* the 40 GB GPU memory, but these datasets can be held in the host memory. Query vectors are divided into two batch sizes, 2048 (large) and 8 (small), representing offline and online workloads, respectively.

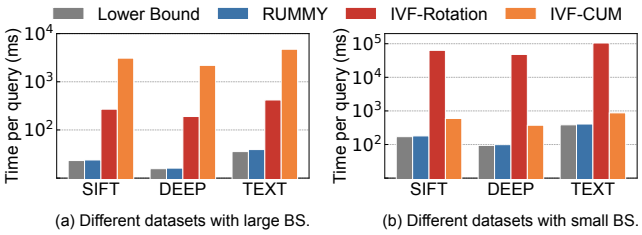


Figure 9: Overall performance on GPUs.

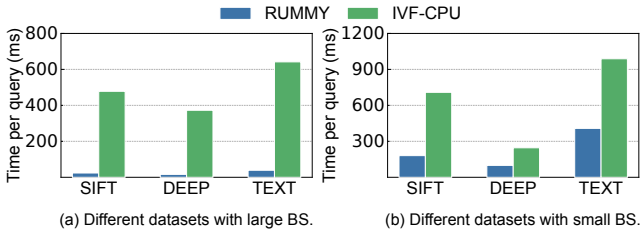


Figure 10: Comparison between RUMMY and IVF-CPU.

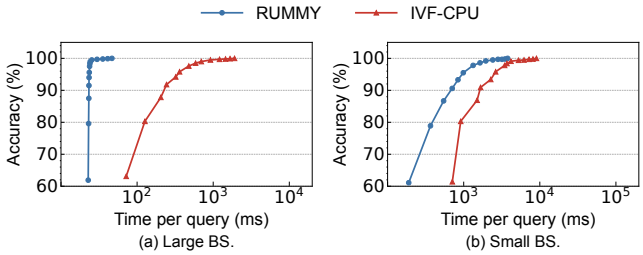


Figure 11: Comparison under different accuracy.

Metrics. We use the average end-to-end time per query as the main metric. In addition, when comparing GPU-based solutions to CPU-based solutions, we also use per-dollar performance, to evaluate the cost-effectiveness. Specifically, per-dollar performance is calculated as $\frac{Price}{Time\ per\ Query}$, where *Price* is the AWS EC2 On-Demand hourly rate [54].

Baselines. IVF is a state-of-the-art ANN index and has been widely deployed in production for billion-scale datasets. It is supported on both GPUs and CPUs in Faiss [23]. Its GPU version requires the dataset to be fully loaded into GPU memory. There are *no* existing GPU-based systems that support large datasets beyond GPU memory. We extend the GPU version of IVF to implement two baseline systems to compare with RUMMY. Specifically, we compare RUMMY to the following three baselines while keeping the same searching parameters under two different batch sizes.

- **IVF-Rotation.** It extends the GPU version of IVF with the strawman rotation method described in §2.2 to support large datasets beyond GPU memory.
- **IVF-CUM.** It extends the GPU version of IVF with CUDA unified memory [26] to expand GPU with host memory.
- **IVF-CPU.** It is the CPU version of IVF, which exploits Intel AVX [55] to speed up vector operations.

Overall performance. We first compare the time per query of RUMMY to the GPU-based baselines, IVF-Rotation and IVF-CUM, and keep the same searching parameters. The experiments are conducted on p4d.24xlarge with three billion-scale datasets. We only use one GPU in p4d.24xlarge since one GPU is enough for RUMMY and p4d.24xlarge is the only instance configured with A100 GPUs on AWS. RUMMY outperforms the baselines by hundreds of times and achieves near-optimal performance with *reordered pipelining*. The results are shown in Figure 9. We summarize as follows.

- The lower bound is calculated by the larger value of transmission and computation time per query. It represents an ideal case that has the maximum overlapping between transmission and computation without any pipelining overhead. Figure 9 shows that RUMMY processes a query within *a few milliseconds* and is close to the lower bound, i.e., it achieves near-optimal performance.
- RUMMY outperforms IVF-Rotation by 10.7-11.7× under large BS and hundreds of times under small BS. The performance gap is larger on small BS, as IVF-Rotation iterates over all data parts and introduces a fixed transmission time under any batch sizes. When confronting small BS, the query time is dominated by the fixed transmission time, which introduces a large performance gap between RUMMY and IVF-Rotation.
- RUMMY outperforms IVF-CUM by 121-135× under large BS and by 2.1-3.7× under small BS. RUMMY outperforms IVF-CUM by a large margin under large BS, since large BS introduces heavier computation and more interference between GPU SMs. This interference causes severe GPU memory page faults.

Comparison to CPU. We then compare RUMMY and IVF-CPU. The results are shown in Figure 10. RUMMY (with one A100 GPU) achieves 2.4-23.1× higher performance than IVF-CPU (with 64 vCPUs). Besides time per query, RUMMY also achieves better per-dollar performance (26.7-37.7× under large BS), i.e., RUMMY is more cost-effective. As the batch size grows, the performance gap becomes larger. This is because large BS introduces high computation parallelism for RUMMY with GPUs but high computation interference for CPU solutions. It is notable that the performance of two GPU baselines (IVF-CUM and IVF-Rotation) falls short of IVF-CPU. This is because the datasets exceed the GPU memory and existing GPU solutions are not optimized for such scenario (§2.2). This emphasizes the necessity for RUMMY on large datasets.

Moreover, Figure 11 demonstrates the performance improvement of RUMMY over IVF-CPU under different accuracy on SIFT1B. RUMMY constantly outperforms IVF-CPU under any configuration.

6.2 Deep Dive of RUMMY

Setup & workloads. We use two AWS EC2 GPU instances to deep dive into RUMMY and evaluate the effectiveness of each

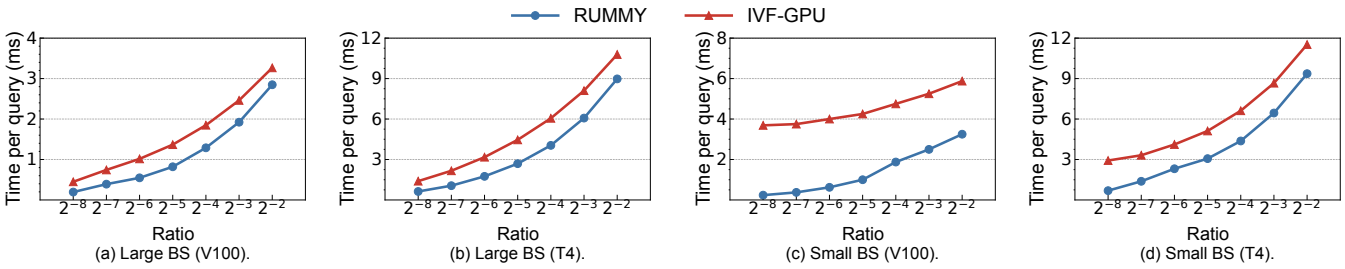


Figure 12: Effectiveness of RUMMY's dynamic kernel padding with cluster balancing.

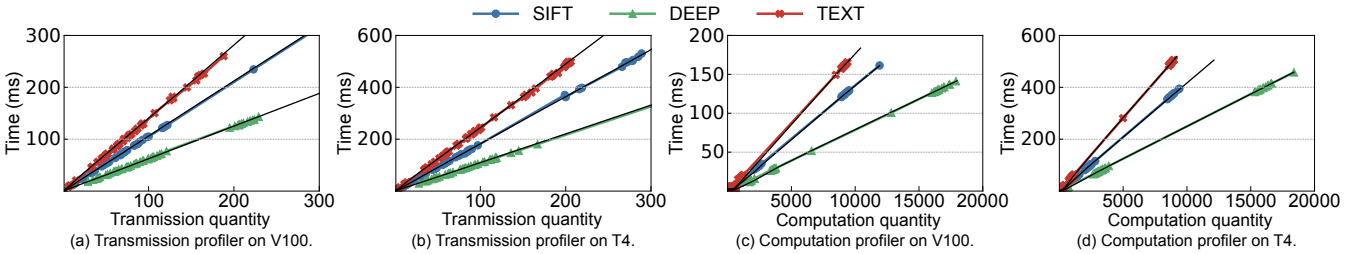


Figure 13: Effectiveness of RUMMY's profiler.

component in RUMMY. One is p3.2xlarge configured with one NVIDIA V100 GPU with 16 GB GPU memory. The other is g4dn.2xlarge configured with one NVIDIA T4 GPU with 16 GB GPU memory. In addition, we reduce large BS to 256 and the scale of the datasets since the computing power and memory of T4 and V100 is not a patch on that of A100. The four micro-datasets are listed at the bottom of Table 2, and the top three all oversubscribe the 16 GB GPU memory. The reason to use small GPUs and datasets is to reduce evaluation costs due to our *limited budget*. Besides, the experiments on million-scale datasets show that RUMMY maintains its effectiveness even with moderately sized datasets.

6.2.1 Dynamic Kernel Padding with Cluster Balancing

We measure the computation performance with SIFT10M, which can be held in GPU memory, and do not consider transmission in this experiment. We run the vector queries under various top-n. The baseline is IVF-GPU without the two techniques. The two both operate the memory pointers. Thus, the negligible overhead of index building and query is not included in the evaluation.

As shown in Figure 12, the x-axis represents the ratio of top-n to the number of all clusters. As the ratio grows, the performance decreases. Meanwhile, large BS introduces high computation density causing high time per query. As Figure 12(c) and Figure 12(d) show, RUMMY outperforms IVF-GPU by up to 15.5× under small BS. The performance improvement is mainly from the dynamic kernel padding since IVF-GPU cannot utilize the SMs under small BS. The performance gap becomes smaller under small BS as illustrated by Figure 12(a) and Figure 12(b). Both RUMMY and IVF-GPU fully utilize the SMs, but RUMMY still has a better performance (1.1-2.3×) due to the cluster balancing that mitigates the straggler block in the entire kernel.

6.2.2 Profiler

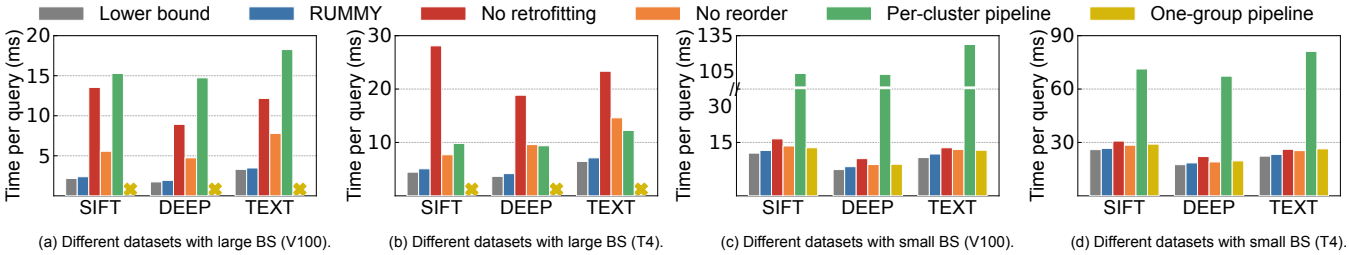
We use SIFT40M, DEEP50M, and TEXT30M to evaluate the effectiveness of RUMMY's profiler, and run the vector queries on V100 and T4 GPUs with two batch sizes. We record the profiler's estimated time of each group and measure the group's actual transmission or computation time. Figure 13 shows the results. The colored lines represent the actual time, and the black lines represent the estimated time. We plot the points of two batch sizes in one line. The x-axis represents the transmission quantity or the computation quantity (defined in §4.3.1). The results confirm that the profiler closely tracks the actual time, thereby demonstrating the effectiveness of the simple yet practical formulas in §4.3.1.

6.2.3 Pipelining Scheduler & Cluster-based Retrofitting

We use SIFT40M, DEEP50M, and TEXT30M in this experiment. We compare the following techniques discussed in §4.1 and §4.3, while keeping other components the same.

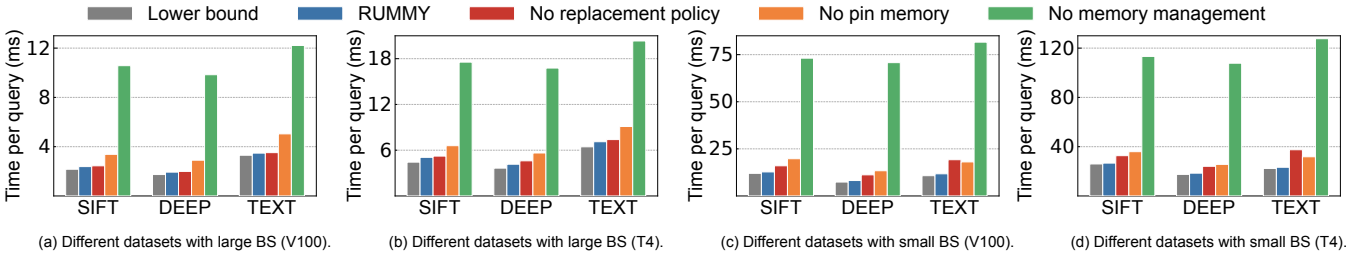
- **Lower bound.** It is the theoretical lower bound of time.
- **RUMMY.** It is RUMMY with all the techniques.
- **No retrofitting.** It does not use retrofitting.
- **No reordering.** It does not use Algorithm 1 for reordering.
- **Per-cluster pipeline.** It groups the clusters on per-cluster granularity and processes the query cluster by cluster.
- **One-group pipeline.** It groups all clusters into one group.

Figure 14 shows the performance of the above techniques under two batch sizes and two GPU instances. RUMMY's cluster-based retrofitting improves the performance by 1.1-5.7× through eliminating the redundant transmission. The reordering algorithm improves the performance by 1.5-2.4× under large BS while only having a slight improvement under small BS. This is because a larger batch size leads to a complex query plan (allowing the reordering algorithm to identify



(a) Different datasets with large BS (V100). (b) Different datasets with large BS (T4). (c) Different datasets with small BS (V100). (d) Different datasets with small BS (T4).

Figure 14: Effectiveness of RUMMY's reordering and grouping with cluster-based retrofitting.



(a) Different datasets with large BS (V100). (b) Different datasets with large BS (T4). (c) Different datasets with small BS (V100). (d) Different datasets with small BS (T4).

Figure 15: Effectiveness of RUMMY's GPU memory management.

Workload	Reorder Overhead	Group Overhead	Total Time
SIFT40M (BS=256, V100)	7.4 μs (0.31%)	10 μs (0.44%)	2.4 ms
SIFT40M (BS=256, T4)	6.3 μs (0.12%)	7.4 μs (0.15%)	5.1 ms
SIFT40M (BS=8, V100)	57 μs (0.45%)	47 μs (0.37%)	12 ms
SIFT40M (BS=8, T4)	55 μs (0.2%)	42 μs (0.16%)	27.3 ms

Table 3: Runtime overhead of reordering and grouping.

a greater scope for optimization), while a smaller batch size only has a few clusters for computation and the reordering algorithm has little room for optimization. The per-cluster and one-group pipelining are two extreme grouping plans, which lead to the maximum pipelining overhead and the minimum overlapping, respectively. RUMMY's grouping plan achieves the optimal tradeoff between them (1.1-13.8× performance improvement). The one-group pipelining fails to process vector queries under large BS since it requires a large amount of data transmission for one group, and thereby oversubscribing the GPU memory. The lower bound is the theoretical lowest time per query. RUMMY achieves the closest performance to the lower bound with the aforementioned techniques.

The runtime scheduling overheads of the reordering and grouping are listed in Table 3 on SIFT40M. The overhead per query is within tens of microseconds, which is relatively small compared to the total time per query ($\leq 0.5\%$).

6.2.4 Memory Management

The setup is the same as §6.2.3. We do not compare RUMMY's memory management to CUDA unified memory, since the comparison is already included in §6.1. We compare the following memory management mechanisms discussed in §4.3, while keeping other components the same.

- **Lower bound.** The same with the lower bound in §6.2.3.
- **RUMMY.** It is RUMMY with all the techniques.
- **No memory management.** It allocates and frees GPU memory at runtime through `cudaMalloc` and `cudaFree`.
- **No replacement policy.** It evicts a random page (cluster).
- **No pin memory.** It uses pageable host memory.

As shown in Figure 15, the replacement policy improves the performance by 1.2-1.6× under small BS while having little effect under large BS. This is because large BS transmits most of the clusters, including cold clusters. The differentiation between cold and hot clusters becomes meaningless. As for the pin memory and the memory layout with pre-allocated memory techniques, they improve the performance by 1.3-1.6× and 2.8-8.6×, respectively. This is because the pin memory improves the overall memory transmission bandwidth, and the memory layout reduces memory fragmentation and frequent invocations to CUDA native interfaces. This experiment demonstrates that all memory management techniques of RUMMY are effective and the combination of them achieves the closest performance to the ideal case.

7 Discussion

System, not index. We emphasize that RUMMY is not a new ANN index, but a new vector query processing system (with new techniques on query pipelining) to support billion-scale datasets beyond GPU memory. RUMMY is built on the vector query processing system, Faiss and ANN index, primitive IVF. Many works (e.g., SPANN [46] and Auncel [56]) propose variations of IVF to improve query efficiency. We do not compare these works to RUMMY, as the underlying hardware-backends and index algorithms differ significantly. However, RUMMY is orthogonal to these algorithmic works and can be integrated into these variations of IVF to support GPU acceleration. We do not directly compare RUMMY to vector

databases, since RUMMY focuses on the query processing part.

Vector quantization. Vector quantization [57] is proposed to reduce the memory footprint of large datasets. It compresses high-dimensional vectors into low-dimensional space, thereby reducing the memory usage. RUMMY supports vector queries beyond GPU memory through system techniques and is orthogonal to vector quantization. Besides, vector quantization can be integrated into RUMMY's primitive IVF index to further reduce memory usage and improve query efficiency.

Reordered pipelining. Pipelining, a common technique to enhance computer system performance, is uniquely implemented in RUMMY, which differs from traditional methods. RUMMY leverages the *independent* and *non-deterministic* nature of computational units in vector query processing on IVF. RUMMY's reordered pipelining is also applicable to other domains, such as batch request processing in LLM inference. Specifically, when using host memory to store the key-value tensors, it offers a strategy to decide the group (i.e., batch) and order of requests, which can efficiently parallelize the computation and key-value tensor transmission between host memory and GPU memory.

8 Related Work

A variety of ANN algorithms (e.g., inverted file [41, 45, 46], locality-sensitive hashing [44, 58–60], and graph algorithms [42, 47, 48]) are proposed for vector query processing. IVF and graph index are among the most popular ANN algorithms. They have different system characteristics. There are several reasons why we choose IVF to build RUMMY to support billion-scale datasets beyond GPU memory. First, IVF is proved to be more efficient than graph index on billion-scale datasets with the same memory footprint according to the recent work [46]. This work demonstrates the benefits of IVF and the limitations of graph index, which applies IVF and significantly outperforms the state-of-the-art graph index, DiskANN [48]. Second, graph index maintains a huge graph which introduces 4× memory usage [61] than IVF. As a result, graph index is not suitable for billion-scale datasets with limited GPU memory. Last, graph index (e.g., HNSW) cannot be easily integrated into GPU due to its random access pattern. This paper focuses on IVF-GPU, and we leave the support of different ANN algorithms in RUMMY as future work.

A set of works [56, 62, 63] focuses on parameter tuning to improve the accuracy and query latency. For example, LAET [63] leverages a decision tree model to early terminate a query when it is hard to improve query accuracy. These optimizations can be integrated into RUMMY since RUMMY focuses on system techniques and does not change any index characteristics. ANN algorithms can be accelerated by hardware accelerators, such as GPUs [23] and FPGAs [6]. However, prior solutions are not suitable for large vector datasets and require the datasets to be preloaded into the global memory of the accelerators. We can exploit RUMMY's core idea (i.e., reordered pipelining) to expand the accelerator memory with host memory as well.

With the proliferation of unstructured data and deep learning, ANN algorithms on embedding vectors become a key component (long-term memory retrieval) in many AI applications, such as recommendation systems [4, 5, 21], recognition [7–9, 22], information retrieval [10–12] and LLM-based AI applications [33–35]. RUMMY benefits these applications by accelerating vector query processing and reducing cost. Recent industrial vector databases [16, 17, 19, 64] adopt Faiss [23] as their query engine. RUMMY, based on Faiss, can be integrated into these systems. Big data processing [65–67] is prevalent in cloud services. RUMMY can be integrated into these services with GPUs to manage large unstructured data and accelerate vector query processing.

9 Conclusion

We present RUMMY, the first GPU-based system for billion-scale vector query processing beyond GPU memory. RUMMY expands GPU memory with host memory to achieve high performance and low cost by leveraging a novel *reordered pipelining* technique. We evaluate RUMMY on various benchmarking datasets with billions of items and show that it outperforms GPU-based baselines by up to 135× with near-optimal performance. Our experiments also show that RUMMY is capable of achieving better performance and cost than the state-of-the-art CPU-based solution.

Acknowledgments. We sincerely thank our shepherd Yu Hua and the anonymous reviewers for their valuable feedback on this paper. This work was supported by the National Natural Science Foundation of China under the grant numbers 62325201, 62172008, and the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Xin Jin and Xuanzhe Liu are the corresponding authors. Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.

[2] R. Blumberg and S. Atre, "The problem with unstructured data," *Dm Review*, 2003.

[3] "Eighty Percent of Your Data Will Be Unstructured in Five Years," 2023. https://solutionsreview.com/data-management/.

[4] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *WWW*, 2007.

[5] M. Grbovic and H. Cheng, "Real-time personalization using embeddings for search ranking at airbnb," in *ACM SIGKDD*, 2018.

[6] C. Zeng, L. Luo, Q. Ning, Y. Han, Y. Jiang, D. Tang, Z. Wang, K. Chen, and C. Guo, "*FAERY*: An fpga-accelerated embedding-based retrieval system," in *USENIX OSDI*, 2022.

[7] R. He, Y. Cai, T. Tan, and L. Davis, "Learning predictable binary codes for face indexing," *Pattern recognition*, 2015.

[8] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[9] L. Zheng, L. Shen, L. Tian, S. Wang, J. Wang, and Q. Tian, "Scalable person re-identification: A benchmark," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[10] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, "The rise of deep learning in drug discovery," *Drug discovery today*, 2018.

[11] A. C. Mater and M. L. Coote, "Deep learning in chemistry," *Journal of chemical information and modeling*, 2019.

[12] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, 2015.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017.

[14] "Introducing chatgpt," 2023. https://openai.com/blog/chatgpt.

[15] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[16] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, *et al.*, "Milvus: A purpose-built vector data management system," in *ACM SIGMOD*, 2021.

[17] "Introducing zilliz," 2023. https://zilliz.com/.

[18] "Introducing pinecone," 2023. https://www.pinecone.io/.

[19] C. Wei, B. Wu, S. Wang, R. Lou, C. Zhan, F. Li, and Y. Cai, "Analyticdb-v: A hybrid analytical engine towards query fusion for structured and unstructured data," in *Proceedings of the VLDB Endowment*, 2020.

[20] "Introducing qdrant," 2023. https://qdrant.tech/.

[21] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *ACM SIGKDD*, 2018.

[22] X. Liu, W. Liu, H. Ma, and H. Fu, "Large-scale vehicle re-identification in urban surveillance videos," in *ICME*, 2016.

[23] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, 2019.

[24] "yandex billion-scale datasets," 2023. https://research.yandex.com/datasets/biganns.

[25] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Searching in one billion vectors: re-rank with source coding," in *ICASSP*, 2011.

[26] "CUDA Unified Memory," 2023. https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[27] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "Mips: A microprocessor architecture," *ACM SIGMICRO Newsletter*, 1982.

[28] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," *ACM Sigarch Computer Architecture News*, 2002.

[29] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "Pipeswitch: Fast pipelined context switching for deep learning applications," in *USENIX OSDI*, 2020.

[30] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: generalized pipeline parallelism for DNN training," in *ACM SOSP*, 2019.

[31] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: Nimble task scheduling for serverless analytics," in *USENIX NSDI*, 2021.

[32] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in *USENIX OSDI*, 2020.

[33] "Chatgpt plugin: Retrieval," 2023. https://openai.com/blog/chatgpt-plugins#retrieval.

[34] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[35] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, 2020.

[36] "Benchmarking nearest neighbors," 2023. https://github.com/erikbern/ann-benchmarks/.

[37] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Advances in Neural Information Processing Systems*, 2015.

[38] M. Muja and D. Lowe, "Flann-fast library for approximate nearest neighbors user manual," *VISAPP*, 2009.

[39] "Annoy," 2023. https://github.com/spotify/annoy.

[40] L. Boytsov and B. Naidan, "Engineering efficient and effective non-metric space library," in *SISAP*, 2013.

[41] A. Babenko and V. Lempitsky, "The inverted multi-index," *TPAMI*, 2014.

[42] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *TPAMI*, 2018.

[43] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM TOG*, 2008.

[44] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, 2004.

[45] D. Baranchuk, A. Babenko, and Y. Malkov, "Revisiting the inverted indices for billion-scale approximate nearest neighbors," in *ECCV*, 2018.

[46] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "SPANN: Highly-efficient billion-scale approximate nearest neighbor search," in *Advances in Neural Information Processing Systems*, 2021.

[47] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," in *Proceedings of the VLDB Endowment*, 2019.

[48] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," in *Advances in Neural Information Processing Systems*, 2019.

[49] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the royal statistical society. series c (applied statistics)*, 1979.

[50] "CUDA Refresher: The CUDA Programming Model," 2023. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[51] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent gpu-accelerated dnn inferences," in *USENIX OSDI*, 2022.

[52] "Nvidia Achieved Occupancy," 2023. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[53] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computers*, 2001.

[54] "Amazon EC2 On-Demand Pricing," 2023. https://aws.amazon.com/ec2/pricing/on-demand/.

[55] "Intel Advanced Vector Extensions," 2023. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[56] Z. Zhang, C. jin, L. Tang, X. Liu, and X. Jin, "Fast, approximate vector queries on very large unstructured datasets," in *USENIX NSDI*, 2023.

[57] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2013.

[58] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the VLDB Endowment*, 2007.

[59] Y. Zheng, Q. Guo, A. K. Tung, and S. Wu, "Lazylsh: Approximate nearest neighbor search for multiple distance functions with a single index," in *ACM SIGMOD*, 2016.

[60] L. Gong, H. Wang, M. Ogihara, and J. Xu, "idec: indexable distance estimating codes for approximate nearest

neighbor search," in *Proceedings of the VLDB Endowment*, 2020.

[61] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[62] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos, "Progressive similarity search on time series data," in *BigVis*, 2019.

[63] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving approximate nearest neighbor search through learned adaptive early termination," in *ACM SIGMOD*, 2020.

[64] "Pinecone: Introduction to Facebook AI Similarity Search (Faiss)," 2023. https://www.pinecone.io/learn/series/faiss/faiss-tutorial/.

[65] "Google BigQuery," 2023. https://cloud.google.com/bigquery/.

[66] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, *et al.*, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[67] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *ACM SIGMOD*, 2015.

A Appendix

A.1 Proof of Theorem§ 4.1

Proof. To further elaborate the problem formulation in §4.3.2, let the final result of Algorithm 1 be $O = [B_{o_1}, B_{o_2} ... B_{o_m}]$, and there are m^* balanced clusters that are already in the GPU (i.e., $T(B_{o_i}) = 0,\ 0 < i \leq m^*$). η is the constant transmission time of the leftover balanced clusters since these clusters have the same size (i.e., $T(B_{o_i}) = \eta,\ m^* < i \leq m$).

Based on Formula 1 and Formula 2, we employ greedy exchange to show that *FindOptOrder* outputs the optimal order. The original order is denoted by O. The exchanged order is denoted by O^*. The general exchange can be transferred into the adjacent exchange. Our target is to prove that $\Upsilon(O) \leq \Upsilon(O^*)$ is always true if we exchange B_{o_i} and $B_{o_{i+1}}$ in O, $1 \leq i \leq m-1$. B_{o_i} has a higher priority than $B_{o_{i+1}}$, and the priority is shown in Algorithm 1. We classify the exchange paradigm into the following three cases.

Case 1: $T(B_{o_i}) = T(B_{o_{i+1}}) = 0$**.** The balanced clusters, B_{o_i} and $B_{o_{i+1}}$, are already in the GPU and their transmission time is both zero. Because of the delay, we have $\Upsilon(O[1:j]) \geq \sum_{k=1}^{j} T(B_{o_k}) \geq \sum_{k=1}^{j-1} T(B_{o_k})$ where $1 \leq j \leq m$. As such,

$$\begin{aligned}
&\Upsilon(O[1:i+1]) = max(\Upsilon(O[1:i]), \sum_{k=1}^{i-1} T(B_{o_k})) + E(B_{o_{i+1}}) \\
&= \Upsilon(O[1:i]) + E(B_{o_{i+1}}) \\
&= max(\Upsilon(O[1:i-1]), \sum_{k=1}^{i-1} T(B_{o_k})) + E(B_{o_i}) + E(B_{o_{i+1}}) \\
&= \Upsilon(O[1:i-1]) + E(B_{o_i}) + E(B_{o_{i+1}}) \qquad (3)
\end{aligned}$$

Similarly, we have:

$$\Upsilon(O^*[1:i+1]) = \Upsilon(O^*[1:i-1]) + E(B_{o_i}) + E(B_{o_{i+1}}) \quad (4)$$

Since O^* is the exchanged order of O which only swaps B_{o_i} and $B_{o_{i+1}}$, we get $O^*[1:i-1] = O[1:i-1]$ and $O^*[i+1:m] = O[i+1:m]$ (m is the number of input balance clusters). Based on Formula 3 and Formula 4, we conclude that $\Upsilon(O[1:i+1]) = \Upsilon(O^*[1:i+1])$. Besides, $O^*[i+1:m] = O[i+1:m]$, and $\sum_{k=1}^{j} T(B_{o_k})$ is the same in the recurrence formula of O and O^* when $j \geq i+1$. Therefore, $\Upsilon(O) = \Upsilon(O^*)$.

Case 2: $T(B_{o_i}) = 0$ **and** $T(B_{o_{i+1}}) = \eta$**.** First, B_{o_i} is already in the GPU and its transmission time is zero. η represents the constant time to transmit a balanced cluster. According to Formula 1,

$$\begin{aligned}
\Upsilon(O[1:i+1]) &= max(\Upsilon(O[1:i]), n \times \eta) + E(B_{o_{i+1}}) \\
\Upsilon(O^*[1:i+1]) &= max(\Upsilon(O^*[1:i]), n \times \eta) + E(B_{o_i}) \qquad (5)
\end{aligned}$$

where $n \times \eta = \sum_{k=1}^{i+1} T(B_{o_k})$. Also, we have $\Upsilon(O^*[1:i]) = max(\Upsilon(O^*[1:i-1]), n \times \eta) + E(B_{o_{i+1}})$ and $\Upsilon(O[1:i]) = max(\Upsilon(O[1:i-1]), (n-1) \times \eta) + E(B_{o_i}) = \Upsilon(O[1:i-1]) + E(B_{o_i})$ because $\Upsilon(O[1:i-1]) \geq (n-1) \times \eta$. As $\Upsilon(O^*[1:i]) = max(\Upsilon(O^*[1:i-1]), n \times \eta) + E(B_{o_{i+1}})$, we derive that

$$\Upsilon(O^*[1:i]) \geq n \times \eta + E(B_{o_{i+1}}) \qquad (6)$$

As $O[1:i-1] = O^*[1:i-1]$, we get $\Upsilon(O[1:i-1]) = \Upsilon(O^*[1:i-1]) \leq max(\Upsilon(O^*[1:i-1]), n \times \eta)$. Therefore, we conclude that

$$\begin{aligned}
&\Upsilon(O[1:i-1]) \leq max(\Upsilon(O^*[1:i-1]), n \times \eta) \\
\Longrightarrow\ &\Upsilon(O[1:i]) - E(B_{o_i}) \leq \Upsilon(O^*[1:i]) - E(B_{o_{i+1}}) \\
\Longrightarrow\ &\Upsilon(O[1:i]) + E(B_{o_{i+1}}) \leq \Upsilon(O^*[1:i]) + E(B_{o_i}) \qquad (7)
\end{aligned}$$

Besides, we have the following formula based on Formula 5.

$$\Upsilon(O[1:i+1]) = \begin{cases} n \times \eta + E(B_{o_{i+1}}), & n \times \eta \geq \Upsilon(O[1:i]) \\ \Upsilon(O[1:i]) + E(B_{o_{i+1}}), & n \times \eta < \Upsilon(O[1:i]) \end{cases}$$

No matter which case is true, we have $\Upsilon(O[1:i+1]) \leq \Upsilon(O^*[1:i]) + E(B_{o_i}) \leq max(\Upsilon(O^*[1:i]), n \times \eta) + E(B_{o_i}) = \Upsilon(O^*[1:i+1])$ according to Formula 6 and Formula 7. $\sum_{k=1}^{j} T(B_{o_k})$ is the same in the recurrence formula of O and O^* when $j \geq i+1$. Therefore, $\Upsilon(O) \leq \Upsilon(O^*)$.

Case 3: $T(B_{o_i}) = T(B_{o_{i+1}}) = \eta$ **and** $E(B_{o_i}) \geq E(B_{o_{i+1}})$**.** In this case, we have a similar formula to Formula 5:

$$\begin{aligned}
\Upsilon(O[1:i+1]) &= max(\Upsilon(O[1:i]) + E(B_{o_{i+1}}), \\
&\qquad \sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_{i+1}})) \\
\Upsilon(O^*[1:i+1]) &= max(\Upsilon(O^*[1:i]) + E(B_{o_i}), \\
&\qquad \sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_i})) \qquad (8)
\end{aligned}$$

Also, we have $\Upsilon(O^*[1:i]) = max(\Upsilon(O^*[1:i-1]), \sum_{k=1}^{i-1} T(B_{o_k}) + \eta) + E(B_{o_{i+1}})$ and $\Upsilon(O[1:i]) = max(\Upsilon(O[1:i-1]), \sum_{k=1}^{i-1} T(B_{o_k}) + \eta) + E(B_{o_i})$. As $O[1:i-1] = O^*[1:i-1]$, we have

$$\begin{aligned}
&\Upsilon(O[1:i]) - E(B_{o_i}) = \Upsilon(O^*[1:i]) - E(B_{o_{i+1}}) \\
\Longrightarrow\ &\Upsilon(O[1:i]) + E(B_{o_{i+1}}) = \Upsilon(O^*[1:i]) + E(B_{o_i}) \qquad (9)
\end{aligned}$$

Since B_{o_i} has a higher priority than $B_{o_{i+1}}$, we get

$$\begin{aligned}
&E(B_{o_i}) \geq E(B_{o_{i+1}}) \\
\Longrightarrow\ &\sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_{i+1}}) \leq \sum_{k=1}^{i+1} T(B_{o_k}) + E(B_{o_i}) \qquad (10)
\end{aligned}$$

Combining Formula 8, Formula 9 and Formula 10, we conclude

$$\Upsilon(O[1:i+1]) \leq \Upsilon(O^*[1:i+1])$$

Because $\sum_{k=1}^{j} T(B_{o_k})$ is the same in the recurrence formula of O and O^* when $j \geq i+1$, we derive $\Upsilon(O) \leq \Upsilon(O^*)$.

Batch Size	IVF-GPU (T4) Time per Query (ms)	IVF-GPU (T4) Per-Dollar Performance	IVF-CPU (r5.2xlarge) Time per Query (ms)	IVF-CPU (r5.2xlarge) Per-Dollar Performance	HNSW-CPU (r5.2xlarge) Time per Query (ms)	HNSW-CPU (r5.2xlarge) Per-Dollar Performance
8	0.394	3370.81	20.62	96.2	11.37	174.42
32	0.366	3631.12	18.34	108.163	10.5	188.96
128	0.356	3729.54	17.07	116.23	9.79	202.52
512	0.354	3755.30	16.29	121.77	9.47	209.37
2048	0.355	3740.74	16.25	122.05	9.51	208.55

Table 4: Comparison between GPU and CPU on different ANN indexes.

It's easy to extend the three adjacent exchange cases to a general exchange paradigm, i.e., exchanging two arbitrary balanced clusters rather than an adjacent pair. We prove that exchanging will reduce the overlapping and Algorithm 1 is able to find the optimal order on per-cluster granularity.

A.2 Comparison between GPU and CPU

We perform a measurement to compare the performance and the price of GPU-based and CPU-based (IVF and the state-of-the-art graph index, HNSW) vector query processing systems. The current GPU vector query processing systems are based on IVF since the graph index requires random vector access and is not suitable for the continuous memory access pattern on GPUs. The setup is described as follows.

Setup. This measurement is conducted on AWS. We use three EC2 instance types, including two GPU instances (p3.2xlarge and g4dn.2xlarge) and one CPU instance (r5.2xlarge). p3.2xlarge is configured with one NVIDIA V100 GPU while g4dn.2xlarge is configured with one NVIDIA T4 GPU. Both of the GPUs are equipped with 16 GB GPU memory. r5.2xlarge is configured with 8 vCPUs (Intel Platinum 8259CL) with 64 GB host memory. We use SIFT10M [25] (with around 9GB memory footprint on GPU) in this measurement and fix the parameters in index building and query processing (e.g., similar memory footprint and accuracy, recall@10) for fair comparison. The per-dollar performance is calculated by $Price/Time\ per\ Query$. *Price* is AWS EC2 On-Demand hourly rate [54].

Table 4 shows the time per query and per-dollar performance of different devices under different batch sizes. The GPU-based system [23] has lower cost per query than the CPU-based system (IVF-CPU and HNSW-CPU). The larger batch size introduces higher computation parallelism, which fully utilizes the GPU resources. In summary, leveraging GPUs to accelerate vector query processing provides higher performance and is more cost-effective. It is notable that the dataset can be held in the GPU memory, but when the scale of datasets grows, existing GPU-based vector query processing systems cannot support large datasets beyond GPU memory.

A.3 Cluster Balancing Algorithm

The pseudo-code is in Algorithm 2. The function *BALANCE* balances the original clusters in *host memory* and returns the balanced clusters with equal size. It takes two inputs: the l original clusters and a configurable standard deviation. It uses *cluster_sizes* to store the original clusters' sizes, and uses *map_sizes* to store the balanced clusters' sizes and their related original clusters. ρ represents the fixed size of balanced clusters while χ is the minimal threshold specified by the user. The function first initializes array *cluster_sizes* through a for loop and initializes ρ as the minimum value in *cluster_sizes* (line 3-7). It then starts a while loop that splits the original clusters until the variance of the generated clusters is less than the given standard deviation, *dev* (line 9-25). Finally, it generates the balanced clusters with the appropriate sizes and returns them (line 26-32). The algorithm runs offline, and the overhead is negligible compared to the index building time.

Algorithm 2 Cluster balancing algorithm

```
1:  function BALANCE({C_1,...,C_l}, dev)
2:      cluster_sizes ← ∅, map_sizes ← ∅
3:      for i = 1 → l do
4:          // SIZE function returns the size of the cluster
5:          cluster_sizes.append(SIZE(C_i))
6:          map_sizes.append(∅)
7:      ρ ← Min(cluster_sizes)
8:      // χ is a configured value
9:      while ρ ≥ χ do
10:         for i = 1 → l do
11:             size ← cluster_sizes[i]
12:             while size ≥ 0 do
13:                 if size ≥ ρ then
14:                     map_sizes[i].append(ρ)
15:                 else
16:                     map_sizes[i].append(size)
17:                 size ← size − ρ
18:         // DEV function returns the standard deviation
19:         cur_dev ← DEV(map_sizes)
20:         if cur_dev ≤ dev then
21:             break
22:         else
23:             ρ ← ρ/2
24:             for i = 1 → l do
25:                 map_sizes[i] ← ∅
26:     BC ← ∅
27:     for i = 1 → l do
28:         start ← 0
29:         for j = 0 → SIZE(map_sizes[i]) do
30:             BC.append(C_i[start : start + map_sizes[i][j]])
31:             start ← start + map_sizes[i][j]
32:     return BC
```