GIGP$^+$: A CPU-GPU Co-Processing Engine for Multi-Vector Retrieval

Zheng Bian
Southern University of Science and Technology
Shenzhen, China
The Hong Kong Polytechnic University
Hung Hom, Hong Kong
cszbian@comp.polyu.edu.hk

Man Lung Yiu
The Hong Kong Polytechnic University
Hung Hom, Hong Kong
csmlyiu@comp.polyu.edu.hk

Bo Tang*
Southern University of Science and Technology
Shenzhen, China
tangb3@sustech.edu.cn

Abstract

Multi-vector retrieval models (e.g., ColBERTv2) offer high retrieval accuracy but suffer from efficiency problems at scale. Recently, several methods have been developed to enhance the efficiency of multi-vector retrieval. On one hand, the state-of-the-art GPU-based method PLAID-GPU exploits the massive parallelism of the GPU to accelerate computation, but it needs to process a considerable amount (e.g., ten thousand) of document candidates. On the other hand, the state-of-the-art (SOTA) CPU-based method IGP employs a more effective strategy to reduce the number of candidates, but fails to utilize the massive parallelism of the GPU. To get the best of both worlds, we propose GIGP$^+$, a CPU-GPU co-processing engine designed to achieve high parallelism and low computational overhead. Our contributions are: (1) an efficient candidate generation kernel that enjoys parallelism while retaining the effectiveness of IGP, (2) a score reordering mechanism that reduces the synchronization overhead and (3) a scheduling strategy for efficient batch processing. Our experiments demonstrate that GIGP$^+$ achieves a 11.0× improvement in query per second (QPS) and reduces latency by 7.6× compared to PLAID-GPU, while maintaining equivalent retrieval accuracy. As for cloud pricing, GIGP$^+$ delivers a 2.3× improvement in queries per dollar over SOTA CPU-based solutions.

CCS Concepts

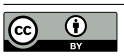
• **Information systems → Top-k retrieval in databases**.

Keywords

Multi-Vector Retrieval, Neural Information Retrieval, Efficient Search

ACM Reference Format:
Zheng Bian, Man Lung Yiu, and Bo Tang. 2026. GIGP$^+$: A CPU-GPU Co-Processing Engine for Multi-Vector Retrieval. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '26), July 20–24, 2026, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3805712.3809743

*Corresponding author.



SIGIR '26, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-2599-9/2026/07
https://doi.org/10.1145/3805712.3809743

1 Introduction

Neural embedding models are widely employed in information retrieval. Notably, multi-vector models (e.g., ColBERTv2 [29]) have achieved state-of-the-art quality across various retrieval tasks [29, 35, 36, 41]. These models represent both documents and queries as sets of embedding vectors, referred to as multi-vectors. The core retrieval problem Multi-Vector Retrieval (MVR) identifies the top-k documents with the highest similarity scores to a given query based on a predefined scoring function.

Accelerating the performance of MVR by leveraging the massive parallelism of GPUs is a promising direction, however, dedicated GPU-based MVR solutions for MVR remain limited. The state-of-the-art GPU-based method, PLAID-GPU [40], proposes a parallel-friendly document candidate generation strategy and scores candidates in parallel, achieving a 10× lower latency than its CPU counterpart. However, PLAID-GPU must process a large number of document candidates (e.g., tens of thousands). Orthogonally, the CPU-based state-of-the-art method IGP [16] employs an efficient candidate generation algorithm that drastically reduces the number of candidates, yielding a 3× speedup over PLAID. The key challenge lies in combining the strengths of both approaches: PLAID-GPU offers high parallelism but suffers from a large candidate set, while IGP operates with few candidates but is limited to single-threaded execution. *How can we exploit the advantages of both approaches simultaneously?*

In this work, we propose GIGP$^+$, a CPU-GPU co-processing engine that integrates the benefits of both methods. We first implement IGP on the GPU (referred to as GIGP), which parallelizes across the constituent vectors of a query and aggregates the results for candidate generation. However, GIGP has two limitations: (1) its degree of parallelism is much lower than the maximum threads supported by the GPU, and (2) it suffers from extensive inter-Streaming Multiprocessor (SM) communication. To overcome these issues, we propose GIGP$^+$, which introduces: (1) an advanced candidate generation kernel that enables massive document-level parallelism (Section 4.3), (2) a score reordering mechanism that reduces global memory synchronization (Section 4.4) and (3) an efficient scheduling strategy that hides the batch query latency (Section 4.5). Together, these innovations effectively address the aforementioned bottlenecks.

Our experiments show that GIGP$^+$ achieves an 11.0× higher queries per second and a 7.6× lower latency than PLAID-GPU,

while matching its retrieval accuracy (Section 5). As for cloud pricing, it delivers a 2.3× improvement in queries per dollar over all CPU-based baselines. These results validate the effectiveness of our proposed techniques (Table 6).

Our source code is open-sourced at [13].

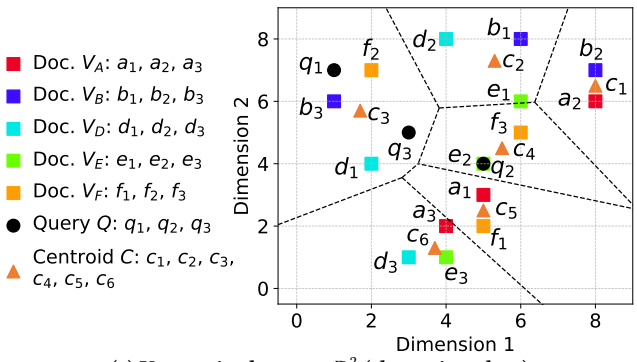
2 Related Work

Neural embedding models are the standard for large-scale information retrieval, broadly categorized into lexical-based, single-vector, and multi-vector approaches. Lexical-based models (e.g., SPLADE [23, 30]) leverage term weighting and neural techniques to generate sparse representations. Single-vector models, such as DPR and ANCE [28, 48], encode a passage into a dense embedding. While computationally efficient [17, 50], they often fail to capture fine-grained semantic interactions, limiting accuracy in complex scenarios [19, 29]. Multi-vector models (e.g., ColBERT [29]) address this by encoding a passage into multiple token-level vectors, enabling detailed semantic modeling. Recent research finds that multi-vector models achieve state-of-the-art performance across diverse tasks, including multi-modal search [35, 36], question answering [11, 12], and passage retrieval [19, 47].

While multi-vector retrieval models achieve high accuracy, they incur significant computational overhead during retrieval. To address this, recent solutions can be categorized into learning-based and indexing-based optimizations. Learning-based methods improve efficiency through two primary strategies: The first is to reduce the multi-vector size during encoding to lower scoring costs. For instance, CITADEL [32] limits token scoring via dynamic lexical routing, while Zong et al. [52] propose regularization losses for effective token pruning that minimize accuracy loss. The second is to design and adapt efficient scoring functions based on the multi-vector models. SLIM [33] transforms multi-vector retrieval into a single-vector problem via score bounds; SPLATE [22] maps multi-vector embeddings to a sparse vocabulary space for efficient sparse retrieval; and XTR [31] proposes a multi-vector score function that eliminates the costly document gathering stage.

Multi-vector indexing-based methods design specialized indexes for efficient retrieval, primarily targeting CPU architectures. PLAID [24, 40] utilizes a space partitioning method for candidate generation and a threshold-based heuristic for efficient pruning. EMVB [38] leverages product quantization [26] for compact representation and proposes SIMD-friendly threshold-based filtering methods with optimized bit vector. DESSERT [21] approximates vector score computations via signed random projections [20]. MUVERA [25] transforms the multi-vector retrieval problem into a single-vector retrieval problem using hashing techniques. IGP [16] exploits data distribution characteristics for efficient candidate generation with nearest-neighbor search indices. The proposed algorithm significantly reduces the number of candidates, thereby lowering the computation cost. WARP [42] proposes a two-stage score reduction and a candidate generation method for XTR-based search.

GPU-based acceleration is promising due to the high memory bandwidth and inherent parallelism of modern graphics processors. PLAID-GPU [40] represents the only existing GPU-based solution for multi-vector retrieval; it parallelizes both the space partitioning and the scoring stages of the PLAID framework on the GPU.



(a) Vectors in the space $\mathbb{R}^2$ (shown in colors)

Doc.	Vec.	q_1	q_2	q_3	Doc.	Vec.	q_1	q_2	q_3
	a_1	26	37	30		b_1	**62**	62	58
V_A:	a_2	**50**	**64**	**54**	V_B:	b_2	57	**68**	**59**
	a_3	18	28	22		b_3	43	29	33
	d_1	30	26	26		e_1	**48**	**54**	**48**
V_D:	d_2	**60**	**52**	**52**	V_E:	e_2	33	41	35
	d_3	10	19	14		e_3	11	24	17
	f_1	19	33	25					
V_F:	f_2	**51**	38	41					
	f_3	41	**50**	**43**					

(b) Inner product $\langle q_i, v_j \rangle$ for each pair (q_i, v_j)

Figure 1: Example of multi-vector retrieval

Specifically, the space partitioning phase computes partition scores in parallel to identify top candidates, while the scoring phase computes the document-level scores of different candidates in parallel. In this paper, we propose GIGP$^+$, a GPU-accelerated retrieval solution based on multi-vector models. We choose to study IGP because it is state-of-the-art in CPU-based Multi-Vector Retrieval [16].

3 Problem Definition and Indexing

We first define the multi-vector retrieval problem in Section 3.1. As background, we then introduce the data storage and indexing mechanism of ColBERTv2 [41] in Section 3.2, followed by the GPU architecture and performance metrics in Section 3.3.

3.1 Problem definition

We define multi-vectors and compute their similarity using the score function from ColBERTv2 [41] and related works [21, 25, 40].

Definition 1 (Multi-vector). A multi-vector V is a set of vectors in $\mathbb{R}^d$, denoted as $V = \{v_1, v_2, \ldots, v_m\}$. Each element $v_i \in V$ is called a constituent vector of V.

All vectors lie in the high-dimensional space $\mathbb{R}^d$ (e.g., $d = 128$).

Definition 2 (Similarity score). Given a query multi-vector Q and a document multi-vector V, their similarity score $\mathcal{F}(Q, V)$ is:

$$\mathcal{F}(Q, V) = \sum_{q \in Q} \max_{v \in V} \langle q, v \rangle \tag{1}$$

where $\langle q, v \rangle$ denotes the inner product between vectors q and v. Without the loss of generality, we assume the inner product score is non-negative. If it is not the case, we add a score offset to make it a positive value.

Example 1. Figure 1(a) displays vectors in $\mathbb{R}^2$. The multi-vector V_B consists of vectors b_1, b_2, and b_3 (blue squares), while the query multi-vector Q comprises q_1, q_2, and q_3 (black dots). To compute $\mathcal{F}(Q, V_B)$, we first evaluate the inner product for each pair (q_i, b_j), as shown in Figure 1(b). The score is obtained as $62 + 68 + 59 = 189$.

We now define the multi-vector retrieval problem as follows.

Definition 3 (Multi-vector retrieval)**.** Given an integer k, a query multi-vector Q, and a dataset $\mathcal{D}$ of multi-vectors, the multi-vector retrieval (MVR) problem returns the k multi-vectors from $\mathcal{D}$ with the highest similarity scores relative to Q.

For clarity, we refer to Q as the *query* and each $V \in \mathcal{D}$ as a *document*. We call $\mathcal{F}(Q, V)$ the document score and $\langle q, v \rangle$ the vector score. The results of MVR are termed the k nearest neighbors of Q. Following existing work on MVR [21, 25, 38, 40], we focus on efficiently finding approximate results.

Example 2. Figure 1(a) illustrates the constituent vectors of query Q and five documents V_A, V_B, V_D, V_E, and V_F. Based on the calculations in Figure 1(b), the top-2 documents are V_B and V_A, with scores of 189 and 168, respectively.

3.2 Data storage and indexing of ColBERTv2

We describe the data storage and indexing scheme of ColBERTv2 [41], which consists of (i) quantized storage for multi-vectors and (ii) an inverted file index. These components are also employed in existing methods [21, 38, 40] as well as in our approach.

Quantized data storage. Vector Quantization (VQ) generates a set of (centroid) vectors to approximate the set of all constituent vectors $\mathcal{D}_v$. It uses a parameter n_c to specify the number of centroids, which are obtained by applying K-means clustering (with $K = n_c$) to $\mathcal{D}_v$. For example, in Figure 1(a), the centroids are c_1 to c_6, and the dotted lines indicate the region closest to each centroid. Each vector (e.g., d_1) is approximately represented by the identifier (ID) of its nearest centroid (e.g., c_3).

Scalar Quantization (SQ) [14] approximates a floating-point number using a B-bit code, where B is a user-defined parameter controlling the precision. In our method, SQ is applied to encode the residual vector $v - c$. This allows for a more accurate reconstruction of the original vector v. A higher value of B generally leads to better reconstruction accuracy at the cost of increased memory usage. Following [40], we set $B = 2$ and $n_c = 16\sqrt{N_v}$, where N_v denotes the total number of document vectors.

Table 1: Inverted file structure

Centroid c	Document IDs IVF$[c]$
c_1	V_A, V_B
c_2	V_B, V_D, V_E
c_3	V_B, V_D, V_F
c_4	V_E, V_F
c_5	V_A, V_F
c_6	V_A, V_D, V_E

Inverted file. The inverted file maps centroids to the document IDs to which their constituent vectors are quantized. For a centroid c, it retrieves the IDs of documents approximated by c. Table 1 shows the inverted file using the example from Figure 1.

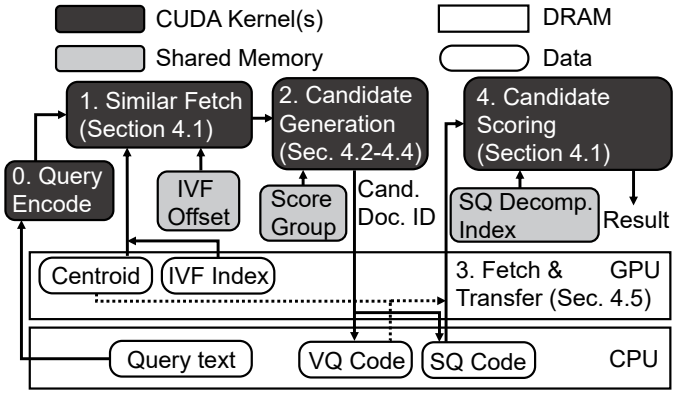


Figure 2: Workflow of GIGP.

3.3 GPU architecture and performance metrics

A GPU comprises multiple Streaming Multiprocessors (SMs), each analogous to a CPU core in a multicore processor. Each SM executes multiple thread blocks, with each thread block being analogous to a CPU process. A thread block contains numerous threads that are managed and scheduled on an SM. The degree of parallelism at both the inter-SM and intra-SM levels can be controlled by configuring the number of thread blocks and the size of each thread block. Each SM features a programmable shared memory that is private to the SM. Although its capacity is relatively small (e.g., typically around 100 KB) compared to global memory, shared memory offers significantly higher bandwidth and lower latency (approximately 10× better). Hence, optimizing memory access by utilizing shared memory rather than global memory is often highly beneficial [46].

Following [18], we evaluate two key performance indicators using NVIDIA Nsight Compute [8]: (1) *Achieved Occupancy (AO)* [3]: percentage of active warps per SM, indicating achieved parallelism (higher is better); (2) *Number of Cycles (NC) per query*: total computational cost in cycles, calculated as Executed Instructions (EI) [6] × Cycles Per Instruction (CPI) [5] (lower is better).

4 Efficient Candidate Generation

We first present the baseline method GIGP in Section 4.1, which employs IGP on GPU to reduce the Number of Cycles (NC). However, the basic candidate generation kernel exhibits two key limitations: low Achieved Occupancy (AO) and high synchronization cost. To address these, we propose: (1) an advanced candidate generation kernel (Section 4.3) for AO improvement, (2) a score reordering technique (Section 4.4) to reduce the synchronization overhead and (3) a scheduling strategy (Section 4.5) for efficient batch processing.

4.1 Baseline solution GIGP

Figure 2 illustrates the workflow of GIGP during the search phase. Following [16], we store the massive index (SQ code and VQ code) on the host (CPU) memory while keeping small indexes in device (GPU) memory. This data layout is designed to accommodate the large size of each index component. The algorithm comprises three CUDA computation steps and a fetch-and-transfer step. The workflow proceeds as follows. At zero step, the query text is transformed into a query embedding for subsequent processing. The first step

Algorithm 1 Similar fetch and PrepareData (Query multi-vector Q, centroid vectors C, inverted file IVF)

1: **function** SF_q.GetTopBatch(n)
2: $n_s \leftarrow \min_{c \in C} |\text{IVF}[c]|$; $k \leftarrow n/n_s$
3: $\{c_{(1)}, c_{(2)}, \ldots, c_{(k)}\} \leftarrow \text{ArgSort}(q^\top C, k)$ ▷ Descending
4: $\Delta \leftarrow \left[0, \left|\text{IVF}[c_{(1)}]\right|, \cdots, \sum_{i=1}^{k-1} \left|\text{IVF}[c_{(i)}]\right|\right]$ ▷ Offset array
5: Cache Δ in shared memory
6: Initialize tuple array $\mathcal{A}$ of length $\sum_{i=1}^{k} \left|\text{IVF}[c_{(i)}]\right|$
7: **parallel for** $i \in 1$ to k **do** ▷ Thread level
8: **for** $j \in 1$ to $\min\{\left|\text{IVF}[c_{(i)}]\right|, n - \Delta[i]\}$ **do**
9: $\mathcal{A}[\Delta[i] + j] \leftarrow \left(\text{IVF}[c_{(i)}][j], \langle c_{(i)}, q\rangle\right)$
10: **return** First n elements of $\mathcal{A}$
11: **function** PrepareTupleArray(Q, ϕ_{cand})
12: Initialize an empty tuple array $\mathcal{T}[q]$ for each $q \in Q$
13: **parallel for** each $q \in Q$ **do** ▷ Thread-block level
14: $\mathcal{T}[q] \leftarrow \text{SF}_q$.GetTopBatch($\phi_{cand}$)
15: **return** $\mathcal{T}$

employs a similar fetching mechanism to retrieve the top candidate document vectors [16].

Definition 4 (Similar fetch, SF_q.GetTopBatch(n))**.** Given a query vector q and the set of all document vectors $\mathcal{D}_v$, SF_q.GetTopBatch(n) returns the top-n most similar vectors with their scores $\langle q, v\rangle$ and document ID, ranked by $\langle q, v\rangle$.

Algorithm 1 shows an approximate similar fetch implementation SF_q and its usage. Here, thread-block-level parallelism (e.g., Line 13) assigns each iterative item to a thread block, enabling efficient data access through shared memory. When no further parallelization is specified, only a single thread within the block is used. In contrast, thread-level parallelism (e.g., Line 7) employs all available threads in the thread block(s) to maximize computational throughput, thereby achieving high efficiency. The procedure consists of three main steps: (1) computing a sufficient number of similar centroid vectors (Lines 2-3), (2) computing and caching the offset array Δ into the shared memory (Lines 4-5), and (3) fetching document vectors by the inverted file index (Lines 7-9). This ensures that each tuple array is sorted in descending order according to the approximate vector score. Function PrepareTupleArray shows the usage of the similar fetch. Given parameter ϕ_{cand}, it generates candidate document score arrays $\mathcal{T}$ through the similar fetch operations (Lines 12-14), which are subsequently used for downstream processing.

The candidate generation step (Step 2) processes the score tuple array $\mathcal{T}$ and outputs a small portion of candidate documents. These candidate documents, represented by their IDs, are transferred to the host to fetch quantization indices (Step 3), enabling finer vector reconstruction. Subsequently, Step 4 refines these candidate documents. This process involves (1) decompressing the quantization codes of VQ and SQ to reconstruct the original document vectors, (2) computing multi-vector similarity scores using $\mathcal{F}(Q, V)$ (Equation 1), and (3) returning the top-k documents with the highest scores. As part of this final step, the scalar quantization (SQ) decompression index—a lookup table that maps SQ codes to floating-point values—is cached in shared memory to accelerate data access.

Algorithm 2 Candidate generation kernel of GIGP (Query Q)

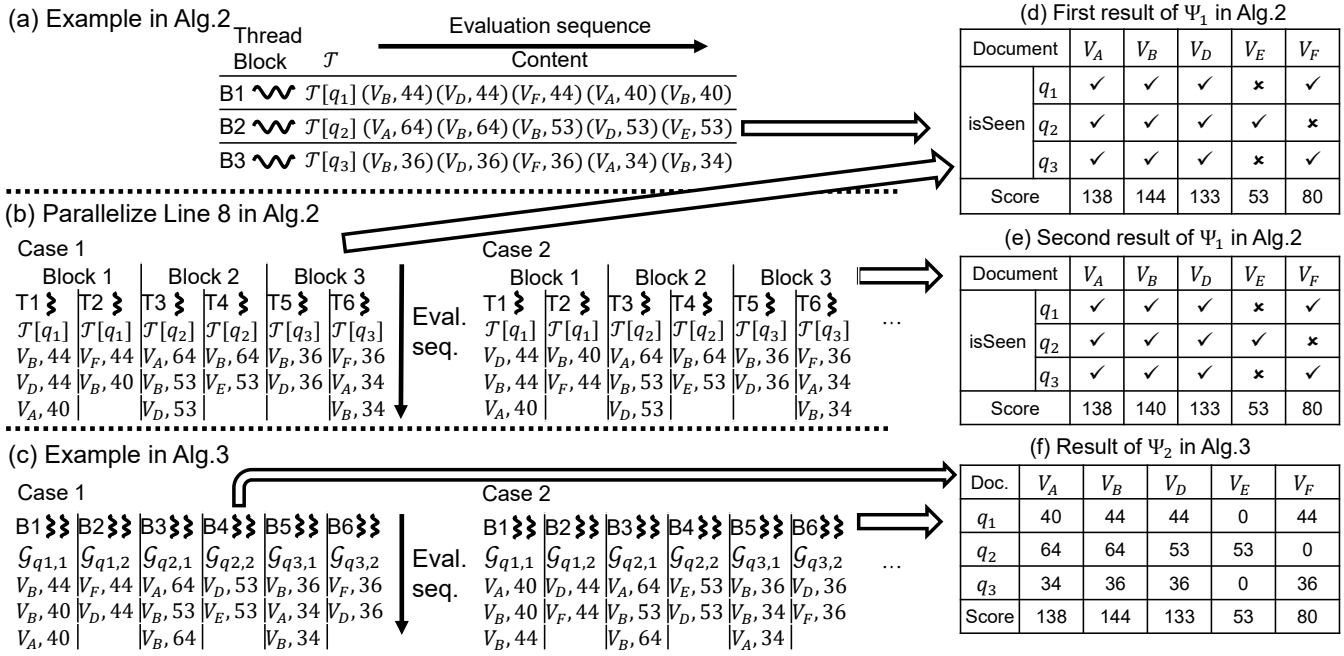
System parameters: ϕ_{cand}, ϕ_{ref}
1: $\mathcal{T} \leftarrow$ PrepareTupleArray(Q, ϕ_{cand}) ▷ Algorithm 1
2: $\Psi_1 \leftarrow$ Create a hash table keyed by document ID
3: **parallel for** each $q \in Q$, $(id, \langle c, q\rangle) \in \mathcal{T}[q]$ **do** ▷ Thread level
4: $\Psi_1[id]$.score $\leftarrow 0$
5: $\Psi_1[id]$.isSeen$[q] \leftarrow$ false
6: **parallel for** each $q \in Q$ **do** ▷ Thread-block level
7: **for** each $(id, \langle c, q\rangle) \in \mathcal{T}[q]$ **do**
8: **if** $\Psi_1[id]$.isSeen$[q]$ = false **then**
9: $\Psi_1[id]$.isSeen$[q] \leftarrow$ true
10: AtomicAdd($\Psi_1[id]$.score, $\langle c, q\rangle$)
11: $\mathcal{S}_1 \leftarrow$ top-ϕ_{ref} document IDs from Ψ_1 by $\Psi_1[id]$.score
12: **return** $\mathcal{S}_1$

Algorithm 2 details the candidate generation kernel of GIGP. It initializes (Lines 3-5) and updates (Lines 6-10) a hash table to compute the result. The initialization phase involves: (1) setting the initial score of each candidate document to 0, and (2) marking all document-query pairs as unseen. Subsequently, the algorithm iterates over each element in $\mathcal{T}$ to update the document scores (Lines 6–10). Specifically, when a document-query pair is encountered for the first time (i.e., marked as unseen), it is marked as seen, and the vector score is added to the document's cumulative score using an atomic operation. At Line 10, `AtomicAdd` ensures thread-safe score accumulation in concurrent executions [43]. This operation is equivalent to $\Psi_1[id]$.score $\leftarrow \Psi_1[id]$.score + $\langle c, q\rangle$. Finally, the top-ϕ_{ref} documents with the highest scores are selected from the hash table (Line 11) and returned as the intermediate result.

Next we show that under identical parameter settings for ϕ_{cand} and ϕ_{ref}, Algorithm 2 produces the same results as Algorithm 1 in [16]. The differences between the two algorithms are that IGP runs the single-thread version of Algorithm 2. Thus, the claim reduces to showing that each for loop outputs consistent results. At Line 10, concurrent threads may update $\Psi_1[id]$. score simultaneously. The `AtomicAdd` operation [4] ensures serialized score accumulation, guaranteeing that only one thread modifies the score at any time. The other loop code maintains correctness because each thread operates on independent data spaces for its respective query vectors. Consequently, the output of Algorithm 2 remains invariant to parallelization, completing the proof.

Two parameters control the trade-off between the filtering quality and computational efficiency: (i) ϕ_{cand}, which specifies the number of similar fetch operations per query vector, and (ii) ϕ_{ref}, which defines the number of candidate documents selected for refinement. These parameters jointly influence the latency-accuracy balance in retrieval. The parameter ϕ_{cand} is tuned via the equation $\phi_{cand} = \phi_{pb} \cdot n_{ivf}$, where n_{ivf} denotes the average number of documents associated with a centroid in the inverted file, and ϕ_{pb} represents the number of probes per search. Recommended parameter configurations are provided in Section 5.1.

Figure 4 presents the performance profiling results for GIGP. We decompose the processing time into three components: the query encode step (Step 0), the candidate generation kernel (Step 2),



(d) First result of Ψ_1 in Alg.2

Document		V_A	V_B	V_D	V_E	V_F
isSeen	q_1	✓	✓	✓	×	✓
	q_2	✓	✓	✓	✓	×
	q_3	✓	✓	✓	×	✓
Score		138	144	133	53	80

(e) Second result of Ψ_1 in Alg.2

Document		V_A	V_B	V_D	V_E	V_F
isSeen	q_1	✓	✓	✓	×	✓
	q_2	✓	✓	✓	✓	×
	q_3	✓	✓	✓	×	✓
Score		138	140	133	53	80

(f) Result of Ψ_2 in Alg.3

Doc.	V_A	V_B	V_D	V_E	V_F
q_1	40	44	44	0	44
q_2	64	64	53	53	0
q_3	34	36	36	0	36
Score	138	144	133	53	80

Figure 3: Comparison of candidate generation kernel implementations and their resulting hash table states.

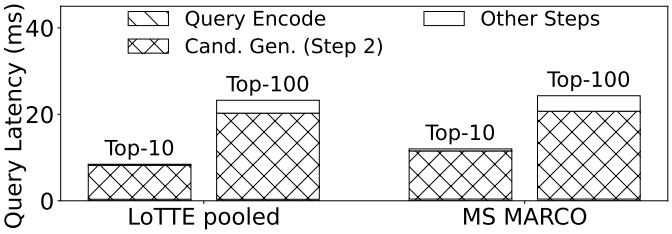


Figure 4: Performance profiling of GIGP.

and the remaining steps. The results indicate that Step 2 constitutes the primary bottleneck. The subsequent section addresses optimizations targeting this step. In particular, Achieved Occupancy (AO) and the synchronization overhead are enhanced and reduced through (1) an advanced candidate generation kernel (Section 4.3) and (2) the score reordering mechanism (Section 4.4), respectively.

4.2 Race condition in parallel nested loop

As analyzed in Section 4.1, the performance bottleneck of Algorithm 2 stems from limited parallelism. In particular, the algorithm parallelizes each query vector (with the number of vectors 32) at Line 6, which under-utilizes the test device's 72 SMs (each supporting 1,024 concurrent threads). A natural optimization is to introduce nested parallelism: parallelizing the iteration over query vectors $q \in Q$ (Line 6) at the thread-block level, and processing the elements within $\mathcal{T}[q]$ (Line 7) at the thread level. However, parallelizing the inner loop over $\mathcal{T}[q]$ introduces a race condition [43]. The following example illustrates this issue.

Example 3. Figure 3 presents an example derived from Figure 1, with parameters $\phi_{cand} = 2$, $\phi_{ref} = 1$, and $k = 1$. Figure 3(a) depicts the sequential execution where Line 7 of Algorithm 2 is not parallelized. In this case, a single thread within a thread block is assigned to each query vector and processes the elements of $\mathcal{T}[q]$ sequentially. Consider the score computation for document V_B. Thread block B1 adds 44, B2 adds 64, and B3 adds 36, resulting in $\Psi_1[V_B]$. score = 44+64+36 = 144. With the computation for all document scores, we obtain the hash table shown in Figure 3(d), where V_B is correctly returned as the result. In this case, the algorithm consistently produces the same output.

However, parallelizing Line 7 can cause incorrect results due to race conditions. In the parallel scheme (Figure 3(b)), the tuple array $\mathcal{T}[q]$ for each query vector q is processed by a thread block containing multiple threads (two in this example). Their non-deterministic execution order leads to varying outcomes because the algorithm depends on the score update sequence. Consider pairs $(V_B, 44)$ and $(V_B, 40)$ in $\mathcal{T}[q_1]$ handled by block B1 with threads T1 and T2:

Case 1: T1 processes $(V_B, 44)$ first, followed by T2 processing $(V_B, 40)$. The score is updated as 44 + 53 + 36 = 144 (correct).

Case 2: T2 processes $(V_B, 40)$ first, followed by T1 processing $(V_B, 44)$. The score becomes 40 + 53 + 36 = 140, leading to the hash table state in Figure 3(e) and an incorrect output of V_A.

The race condition in this example arises because multiple score pairs with the same document identifier (e.g., $(V_B, 44)$ and $(V_B, 40)$ in $\mathcal{T}[q_1]$) are processed concurrently. Sequential execution enforces a deterministic order, whereas parallel execution breaks this ordering and introduces non-determinism.

4.3 Advanced candidate generation kernel

Algorithm 3 shows the candidate generation kernel of GIGP+. Compared with Algorithm 2, the proposed kernel incorporates three enhancements: (1) it replaces the `isSeen` data structure with a cached value $\Psi_2[id][q]$ (Line 6) to store the maximum score per query vector, $\max_{v \in V} \langle q, v \rangle$; (2) it adopts `AtomicMax` (Line 15) instead of `AtomicAdd` for scanning the tuple array, along with a score

Algorithm 3 Candidate generation kernel of GIGP$^+$ (Query Q)

System parameters: ϕ_{cand}, ϕ_{ref}, w
1: $\mathcal{T} \leftarrow$ PrepareTupleArray(Q, ϕ_{cand}) ▷ Algorithm 1
2: $\mathcal{S}_2 \leftarrow$ Set of unique document IDs in $\mathcal{T}$; $\mathcal{T}' \leftarrow \emptyset$
3: $\Psi_2 \leftarrow$ Create a hash table keyed by document ID
4: **parallel for** each $id \in \mathcal{S}_2$ **do** ▷ Thread level
5: Initialize $\Psi_2[id]$.score $\leftarrow 0$
6: Initialize $\Psi_2[id][q]$.score $\leftarrow 0$ for each $q \in Q$
7: **parallel for** each $q \in Q$ **do** ▷ Thread level
8: $\mathcal{G}_{q,1}, \mathcal{G}_{q,2}, \ldots, \mathcal{G}_{q,w} \leftarrow$ ScoreReorder($\mathcal{T}[q]$) ▷ Section 4.4
9: **parallel for** $i = 1$ to w **do** ▷ Thread level
10: $\mathcal{T}'$.append($\mathcal{G}_{q,i}$) ▷ Atomic append
11: **parallel for** every $\mathcal{G}_{q,i} \in \mathcal{T}'$ **do** ▷ Thread-block level
12: $\mathcal{S}_3 \leftarrow$ set of unique document IDs in $\mathcal{G}_{q,i}$
13: Cache $\Psi_2[id][q]$ in shared memory, for every $id \in \mathcal{S}_3$
14: **parallel for** each $(id, \langle c, q\rangle) \in \mathcal{G}_{q,i}$ **do** ▷ Thread level
15: AtomicMax($\Psi_2[id][q]$.score, $\langle c, q\rangle$)
16: **parallel for** each $id \in \mathcal{S}_2$ **do** ▷ Thread level
17: $\Psi_2[id]$.score $\leftarrow \sum_{q\in Q} \Psi_2[id][q]$.score
18: $\mathcal{S}_2' \leftarrow$ top-ϕ_{ref} document IDs from Ψ_2 by $\Psi_2[id]$.score
19: **return** $\mathcal{S}_2'$

addition reduction step (Lines 16-17); and (3) it partitions the score tuple array $\mathcal{T}[q]$ into several groups (Line 8), ensuring that tuples with the identical document IDs are grouped together (Lines 7-10, detailed in Section 4.4). Specifically, score group processing is organized hierarchically: each score group is handled at the thread-block level, while individual score tuples (Line 14) are processed in parallel at the thread level. Other loops in the algorithm are similarly parallelized at the thread level.

Theorem 1. *Under identical parameter settings for ϕ_{cand} and ϕ_{ref}, Algorithm 3 produces the same results as Algorithm 2.*

Proof. Both algorithms return the top-ϕ_{ref} documents based on the scores in the hash table. Denote $\mathcal{S}_1'$ as the set of keys in Ψ_1. Thus, it suffices to show that $\mathcal{S}_1' = \mathcal{S}_2$ and $\Psi_1[id]$.score $= \Psi_2[id]$.score for every document ID id.

First, we prove $\mathcal{S}_1' = \mathcal{S}_2$. This holds because the key of both hash tables is the set of document IDs in $\mathcal{T}$ and both algorithms compute $\mathcal{T}$ identically.

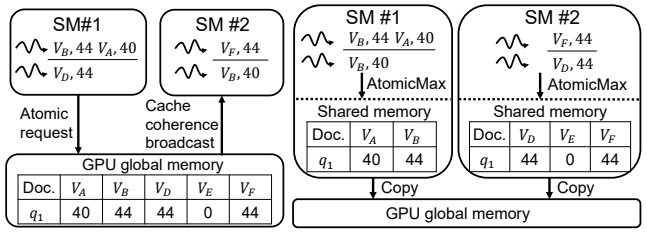
Next, we demonstrate $\Psi_1[id]$.score $= \Psi_2[id]$.score for every $id \in \mathcal{S}$. Define $\mathcal{T}[q][id] = \{\langle c, q\rangle \mid (id, \langle c, q\rangle) \in \mathcal{T}[q]\}$ as the set of scores associated with document id in $\mathcal{T}[q]$. We show that both algorithms compute the document score as:

$$\Psi[id].\text{score} = \sum_{q\in Q} scr_q, \text{where } scr_q = \begin{cases} 0; \text{ if } \mathcal{T}[q][id] = \emptyset, \\ \max(\mathcal{T}[q][id]); \text{ otherwise.} \end{cases}$$

For Algorithm 2, the score tuples in $\mathcal{T}[q]$ are sorted in ascending order of $\langle c, q\rangle$ (Lines 2-9 in Algorithm 1). When updating the hash table (Lines 6–10 in Algorithm 2), the algorithm adds the score only the first time a document-query pair is encountered, which corresponds to the maximum score in $\mathcal{T}[q][id]$. Thus, $scr_q = \max(\mathcal{T}[q][id])$ when $\mathcal{T}[q][id] \neq \emptyset$, and $scr_q = 0$ otherwise.

Table 2: Comparison of candidate generation kernels. AO: Achieved Occupancy (detailed in Section 3.3).

Algorithm	Example Fig.	AO	Recall
Alg. 2	Fig. 3(a)	Low	High
Alg. 2 with Line 7 parallelized	Fig. 3(b)	Medium	Low
Alg. 3	Fig. 3(c)	**High**	**High**



(a) Without score reorder **(b) With score reorder**

Figure 5: Comparison with and without score partition

For Algorithm 3, because tuples sharing the same document ID are assigned to the same score group, the algorithm (Line 15) ensures that $\Psi_2[id][q]$.score stores $\max(\mathcal{T}[q][id])$ whenever $\mathcal{T}[q][id]$ is non-empty, and remains 0 otherwise. Summing these values across all query vectors (Line 17) yields the same document score as in Algorithm 2. Therefore, both algorithms produce identical results. □

Example 4. Figures 3(c) and (f) depict the execution of Algorithm 3 and the corresponding hash table state, respectively. The parameters are set as $\phi_{\text{cand}} = 2$, $\phi_{\text{ref}} = 1$, $w = 2$, and $k = 1$. Each score group $\mathcal{G}_{q,i}$ is assigned to a thread block containing two threads. Consider the processing of $\mathcal{G}_{q_1,1}$, thread block B1 and the score calculation for the pair (q_1, V_B) in the hash table. According to Lines 10–14 in Algorithm 3, the score for (q_1, V_B) is determined by the maximum score associated with V_B in $\mathcal{G}_{q_1,1}$, which is 44. This yields the per-query scores shown in Figure 3(f). The document score is computed by summing all per-query scores, resulting in V_B being returned as the top result, consistent with the outcome in Figure 3(d).

Table 2 summarizes the characteristics of the three algorithms compared. The variant "Alg. 2 with Line 7 parallelized" fails to preserve accuracy due to race conditions. We refer to Section 3.3 for Achieved Occupancy (AO) and to Section 5.1 for Recall. We shall verify these characteristics in Table 6 in the experimental study. The results demonstrate that Algorithm 3 simultaneously maintains high accuracy and achieves a high degree of parallelism, outperforming the other approaches.

Hash table implementation. Hash tables often employ separate chaining [15] or open addressing [18, 27] as underlying data structures, which can introduce hash collisions and increase lookup overhead. In Algorithm 3, since the keys (document IDs) are enumerable, we implement the hash table using a direct-addressed array. Specifically, after Line 2, we compute an index array mapping each candidate document to a unique position. This index array is then used to directly address key-value pairs in the hash table, ensuring $O(1)$ lookup cost without hashing overhead.

4.4 Reducing inter-SM communication

The performance bottleneck stems from extensive atomic max operations (Line 15) performed across the GPU global memory. Specifically, such step (Line 15 in Algorithm 3) is attributed for 40% of the overall process in MS MARCO. This bottleneck arises because the hash table Ψ_2 (Line 3)—with a size of 33 MB for MS MARCO—exceeds the capacity of shared memory (typically ~100 KB), thereby forcing atomic updates to the global memory. To address this limitation, we introduce score reordering through document-level partitioning, as illustrated in Figure 5. The baseline approach (Figure 5(a)) incurs costly inter-SM communication because the hash table cannot reside entirely in shared memory. In contrast, our score reordering technique (Figure 5(b)) restructures the computation so that scores with the same document ID are aggregated within an individual SM. This optimization enables efficient intra-SM communication through shared memory, significantly reducing the inter-SM synchronization overhead.

The score reordering mechanism operates in two phases. In the offline phase, we uniformly and sequentially partition documents to each score group and record the positions of the score groups in the inverted file. During retrieval, the algorithm forms a score group by gathering the corresponding score group for each centroid from the inverted file (Line 8). Each thread block then scans the gathered score group to compute the score pairs. Each group $\mathcal{G}_{q,i}$ is processed in a dedicated thread block (Line 11), enabling efficient use of shared memory. Let n_s be the size of shared memory. The number of groups w is set to $|\mathcal{D}|/n_s$, a value chosen so that each size of score group occupies the full capacity of the available shared memory.

Sensitivity to Vector Dimensionality. Higher dimensionality increases GPU memory usage only slightly and has no effect on the latency of atomic operations or synchronization overhead. This is because only the centroid vectors and scalar quantization (SQ) codes are dependent on dimensionality. In our implementation, centroid vectors reside in GPU memory (requiring 64 MB for MS MARCO), while SQ codes are stored in host memory. Atomic operations and synchronization involve only the inner product scores, which are independent of the vector dimensionality.

4.5 Efficient scheduling

The workflow of GIGP+ (Figure 2) includes the computation (steps 1, 2, and 4) and transfer (step 3). A naive solution serially performs computation and transfer. However, this schedule suffers from GPU underutilization because the CUDA cores and transfer bandwidth are idle alternately. To avoid this, GIGP+ implements a concurrent pipeline that runs across different queries. It identifies each step as a task and schedules the tasks on the CPU. Each CPU thread manages the tasks of a query by atomic variables [10], which are marked as occupied during resource access and released upon completion. This ensures efficient inter-query parallelism without resource contention.

5 Experimental Study

We first introduce the experimental settings in Section 5.1. Then, Section 5.2 compares our method with state-of-the-art GPU-based approaches. Ablation studies and indexing performance analysis are presented in Section 5.3. Finally, we compare the performance with CPU-based methods in Section 5.4.

Table 3: Statistics of the experiment datasets

Dataset	# Document	# Vector	# Vector per set	# Query
Touche-2020	382K	40M	105	49
Quora	523K	8M	16	1,000
LoTTE pooled	2.4M	266M	109	2,931
NQ	2.7M	229M	85	3,452
DBPedia	4.6M	297M	64	67
HotpotQA	5.2M	283M	54	7,405
MS MARCO	8.8M	596M	67	6,980
Wikipedia	21.0M	2,468M	117	6,515

5.1 Experimental settings

Datasets. We use the datasets (see [1, 2, 7] for source) in Table 3 for performance evaluation, which are widely used in information retrieval. In particular, Touche-2020, Quora, NQ, DBPedia, HotpotQA [49] and MS MARCO [39] belongs to the BEIR [44] benchmark. LoTTE pooled [41] and Wikipedia [28] aim for passage retrieval, whose query and passages are extracted from StackExchange, and Wikipedia articles, respectively. Following [40], we use ColBERTv2 [41] to generate the query and document multi-vectors. The vector dimensionality is 128 from the model settings. We note that our solution also applies in the multi-modal settings [34, 35].

Competitors. To demonstrate the indexing and query performance of our method, we compare our solutions GIGP+ with GPU-based methods and CPU-based methods. GPU-based methods includes the GPU version of PLAID [40][1] (called PLAID-GPU) and our basic solution GIGP. The CPU-based methods include IGP[2] [16], PLAID, DESSERT[3] [21], EMVB[4] [38], MUVERA[5] [25] and XTR/WARP[6] [42], the six state-of-the-art solutions in multi-vector retrieval. We use all threads and the GPU to build the index.

Parameter settings. We tune the parameters of the competitors from their recommendation settings and show their best performance in terms of efficiency and accuracy. As shown in Section 4.1 and Section 4.4, we set the parameter of both GIGP and GIGP+ as (ϕ_{pb}, ϕ_{ref}) as $(4, 200)$ and $(12, 1000)$ when $k = 10$ and $k = 100$, respectively. For IGP, we set $(n_b, \phi_{pb}, \phi_{ref})$ as $(8, 8, 400)$ and $(32, 32, 1000)$ when $k = 10$ and $k = 100$, respectively. For EMVB, we set (nprobe, thresh, out-second-stage, thresh-query, n-doc-to-score) as $(1, 0.3, 64, 0.4, 200)$ and $(6, 0.4, 256, 0.4, 256)$ when $k = 10$ and $k = 100$, respectively. In CPU-based multi-threaded scenarios, XTR/WARP, DESSERT and PLAID specify the thread count as a parameter, while IGP, EMVB and MUVERA assign each thread to process a separate query. In QPS-accuracy comparison, we test over the recommended settings in [37] for PLAID. The parameter settings of IGP-based methods (i.e., GIGP and GIGP+) are tuned as follows. When $k = 10$, the parameter ϕ_{pb} is finetuned in the range $\{1, 2, 4, 8, 16, 32\}$ and ϕ_{ref} is finetuned in $\{20, 50, 100, 200, 300, 400,$

[1] https://github.com/stanford-futuredata/ColBERT/tree/fast_search
[2] https://github.com/DBGroup-SUSTech/multi-vector-retrieval
[3] https://github.com/ThirdAIResearch/Dessert
[4] https://github.com/CosimoRulli/emvb
[5] https://github.com/viig99/muvfde
[6] https://github.com/jlscheerer/xtr-warp

Table 4: Performance comparison in GPU-based solution.

Method	$k = 10$							$k = 100$						
	QPS	Lat.(ms)	MRR	Recall	AO(%)	NC	# Cand.	QPS	Lat.(ms)	MRR	Recall	AO(%)	NC	# Cand.
	Dataset: LoTTE pooled													
PLAID-GPU	129.3	7.7	54.3	51.9	45.9	158.6M	14K	73.5	13.6	55.3	71.1	46.9	263.5M	26K
GIGP	185.3	5.4	53.6	50.7	12.2	43.2M	200	65.6	15.3	55.0	71.2	10.7	87.9M	1K
GIGP$^+$	**930.2**	**1.3**	53.7	50.7	65.3	37.7M	200	**555.1**	**2.0**	55.0	71.2	62.5	86.8M	1K
	Dataset: HotpotQA													
PLAID-GPU	127.7	7.9	85.2	67.6	47.7	209.5M	18K	53.9	18.6	85.7	79.8	47.3	351.7M	33K
GIGP	155.1	6.5	84.2	65.3	11.8	53.1M	200	62.3	16.1	85.3	78.8	10.9	102.3M	1K
GIGP$^+$	**604.4**	**1.7**	84.2	65.3	66.0	49.3M	200	**339.5**	**2.9**	85.3	78.8	63.6	99.5M	1K
	Dataset: MS MARCO													
PLAID-GPU	116.3	8.6	39.3	67.6	49.5	191.2M	16K	66.2	15.1	40.5	90.4	47.6	324.3M	29K
GIGP	129.3	7.7	38.7	66.3	11.3	62.9M	200	62.7	15.9	40.3	90.3	11.1	98.1M	1K
GIGP$^+$	**588.9**	**1.9**	38.7	66.4	66.7	60.9M	200	**448.6**	**2.4**	40.3	90.3	64.6	94.8M	1K
	Dataset: Wikipedia													
PLAID-GPU	59.7	16.7	53.9	41.1	47.5	512.1M	44K	33.0	30.3	54.7	69.1	48.1	874.0M	82K
GIGP	63.5	15.7	52.8	38.8	11.3	106.9M	200	37.2	26.9	54.3	68.2	11.7	219.0M	1K
GIGP$^+$	**267.6**	**3.7**	52.8	38.8	64.8	99.7M	200	**167.0**	**6.0**	54.3	68.1	64.2	210.2M	1K

600}. When $k = 100$, the parameter ϕ_{pb} is finetuned in the range $\{1, 2, 4, 8, 16, 32\}$ and ϕ_{ref} is finetuned in {100, 200, 500, 1K, 1.2K, 1.4K, 1.6K, 1.8K, 2K}.

Performance metric. Following [40], we use *Mean Reciprocal Rank (MRR)* and *Recall* as the accuracy measurement. We report *Query Per Second* (QPS), *Query Latency* (Lat., milliseconds per query) and the number of candidates (# Cand.) for efficiency metric. We report *Achieved Occupancy* (AO) and *the Number of Cycles* (NC) for the explanation of the searching time, as explained in Section 3.3.

Platform. The experiment machine is equipped with an Intel(R) Xeon(R) Gold 5318Y@2.10 GHz CPU with 48 threads, a NVIDIA A10 GPU with 22GB device memory, and 512GB main memory with Linux version 4.15.0. All codes for the searching procedure are written in C++17 and compiled with the -O3 optimization flag.

5.2 Query processing evaluation

Table 4 and Table 5 compares the performance of all GPU-based methods with different k, respectively. GIGP$^+$ offers the best overall query performance on all datasets. In particular, GIGP$^+$ achieves QPS improvement of up to 11.0x and latency reduction of up to 7.65x. This experimentally validates our design goal: (1) a highly parallelized algorithm for efficient candidate generation, (2) a score reordering mechanism that reduces the synchronization cost, and (3) an efficient scheduling strategy for batch processing. The reason GIGP$^+$ achieves the best performance can be concluded as follows: (1) Compared with PLAID-GPU, GIGP$^+$ leverages an effective pruning strategy inspired by IGP [16] and enjoys a small number of document candidates. This can be reflected in the significant reduction in NC. (2) Compared with GIGP, GIGP$^+$ proposes an optimized candidate generation kernel that enhances parallelism and a score reordering technique that reduces the number of global memory synchronizations, leading to improved AO.

Figure 6 and Figure 7 compare the QPS-accuracy tradeoff with $k = 10$ and $k = 100$, respectively. We carefully finetune the retrieval parameter of the baseline methods and GIGP$^+$ and select the Pareto frontier to plot the figure. GIGP$^+$ consistently outperforms the baseline on all of the benchmarks. On average, GIGP$^+$ achieves 6-10× speedup on the same accuracy level. This shows GIGP$^+$ is faster than the baseline methods.

5.3 Ablation study and indexing performance

Section 4.4 introduce the score reordering technique with parameter w. Figure 8 shows the influence of different w. Throughput decreases as w increases and reaches a maximum at $w = |\mathcal{D}|/n_s$. This is because a larger w requires more thread blocks, thereby reducing the throughput. Therefore, we suggest setting w so that the score group fully occupies the shared memory.

In this subsection, we evaluate the individual contribution of each key component in GIGP$^+$ to the overall performance. We disable specific modules one at a time and compare the QPS and Recall@100. In particular, As shown in Table 6, the removal of different components leads to varying degrees of performance degradation. *w. Alg.2* substitutes the advanced candidate generation kernel (Algorithm 3) with the basic version (Algorithm 2). *w. Alg. 2 L7 parallelized* modifies the basic candidate generation kernel (Algorithm 2) by parallelizing Line 7. While this version yields a higher QPS and Occupancy compared to the simple kernel replacement, it fails to maintain practical search accuracy. The improvement in QPS comes at the cost of introducing race conditions, which compromise the correctness of the results. This contrast highlights the effectiveness of Algorithm 3 in achieving both high accuracy and high parallelism. *w/o Section 4.4* disables the score reordering module in Section 4.4. *w/o Section 4.5* disables scheduling strategy in Section 4.5, forcing the system to compute and transfer sequentially. The absence of each module incurs a decrease in QPS, confirming the effectiveness of this technique. The optimal performance of GIGP$^+$ relies on their combined effect. Removing all of them (i.e., GIGP) leads to a severe degradation, with QPS dropping by over 80%+.

Table 5: Performance comparison in other datasets.

Method	$k = 10$			$k = 100$			$k = 10$			$k = 100$		
	QPS	MRR	Recall	QPS	MRR	Recall	QPS	MRR	Recall	QPS	MRR	Recall
	Dataset: Touché-2020						Dataset: Quora					
PLAID-GPU	144.8	49.6	16.7	100.1	50.7	47.6	162.8	82.8	88.5	120.7	83.2	97.9
GIGP	256.6	51.2	16.5	79.8	52.2	47.8	362.1	82.5	88.7	110.8	82.8	97.9
GIGP+	**1081.1**	51.2	16.4	**671.6**	52.2	47.7	**1855.3**	82.5	88.6	**1828.2**	82.8	97.9
	Dataset: NQ						Dataset: DBpedia					
PLAID-GPU	85.2	50.4	74.8	51.7	51.4	92.7	164.9	85.6	32.0	102.5	85.9	61.1
GIGP	112.9	49.3	73.0	36.6	50.4	92.4	182.1	86.6	29.3	80.7	84.7	56.5
GIGP+	**947.0**	49.4	73.2	**597.4**	50.4	92.4	**657.0**	86.6	29.0	**514.9**	84.7	57.1

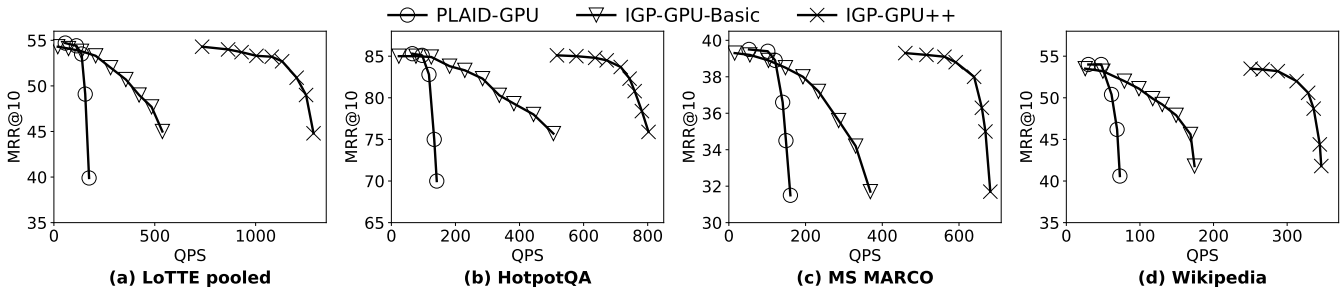


Figure 6: QPS-MRR comparison with the number of retrieved documents $k = 10$. A higher QPS is better.

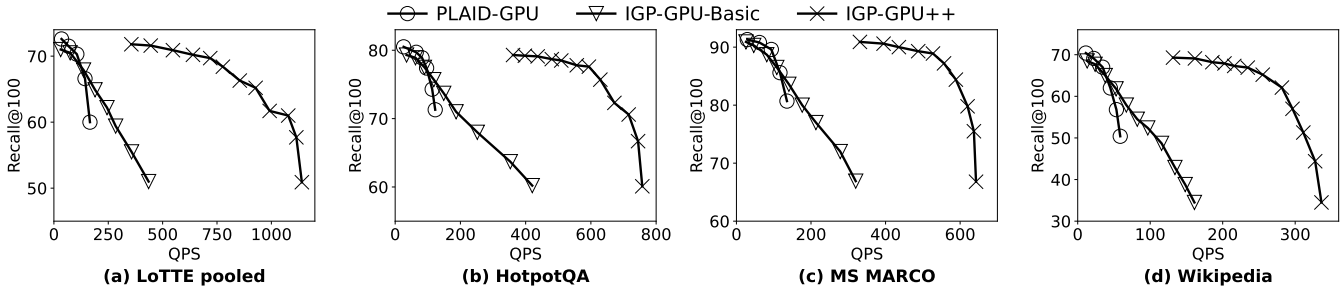


Figure 7: QPS-Recall comparison with the number of retrieved documents $k = 100$. A higher QPS is better.

Table 6: Ablation study with $k = 100$

Method	LoTTE pooled			MS MARCO		
	QPS	AO(%)	Recall	QPS	AO(%)	Recall
GIGP+	555.1	62.5	71.2	448.6	64.6	90.3
w. Alg.2	78.2	11.4	71.2	71.3	13.5	90.3
w. Alg. 2 L7 parallelized	216.4	39.8	3.6	122.6	38.4	2.7
w/o Section 4.4	401.1	58.6	71.2	436.9	59.7	90.3
w/o Section 4.5	361.7	53.0	71.2	273.8	49.6	90.3
GIGP	65.6	10.7	71.2	62.7	11.1	90.3

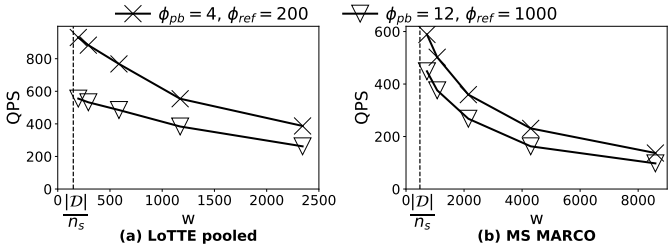


Figure 8: Comparison of QPS with different w.

Table 7: Comparison of Index Size and Index Time.

Method	LoTTE pooled		MS MARCO	
	Index Size	Index Time	Index Size	Index Time
DESSERT	37.0 GB	0.6 h	109.4 GB	1.4 h
MUVERA	45.8 GB	0.2 h	167.8 GB	0.5 h
PLAID	9.7 GB	1.0 h	22.3 GB	1.9 h
EMVB	12.1 GB	1.1 h	18.1 GB	2.0 h
IGP	9.8 GB	1.1 h	22.5 GB	2.0 h
XTR/WARP	18.6 GB	2.3 h	27.5 GB	4.5 h
PLAID-GPU	9.7 GB	1.0 h	22.3 GB	1.9 h
GIGP	9.7 GB	1.0 h	22.3 GB	1.9 h
GIGP+	9.7 GB	1.0 h	22.3 GB	1.9 h

Table 7 compares the Index Size (in device memory and host memory) and the Index Time of different methods. Two key findings are observed: (1) GIGP+'s index size is significantly smaller than MUVERA and DESSERT. This reduction in memory footprint is achieved through Vector Quantization and Scalar Quantization,

Table 8: Comparison with CPU-based methods.

Method	LoTTE pooled								MS MARCO							
	$k = 10$				$k = 100$				$k = 10$				$k = 100$			
	QPS	Q/$	MRR	Recall	QPS	Q/$	MRR	Recall	QPS	Q/$	MRR	Recall	QPS	Q/$	MRR	Recall
CPU-based methods (using single thread)																
DESSERT	18.9	47K	52.5	48.4	11.2	28K	54.0	67.4	12.7	32K	34.9	58.4	9.9	24K	36.3	78.4
MUVERA	22.8	57K	53.7	50.6	12.7	32K	54.5	69.2	10.9	27K	38.3	67.5	9.4	23K	39.0	87.0
PLAID	21.5	53K	54.3	52.0	13.4	33K	55.3	71.0	11.8	29K	39.4	67.6	9.1	22K	40.5	90.4
EMVB	30.1	75K	53.8	50.3	18.6	46K	55.3	70.2	19.5	48K	38.3	64.8	15.8	39K	40.2	89.2
IGP	43.0	107K	53.9	50.9	21.8	54K	54.9	70.5	33.6	83K	38.6	66.5	19.8	49K	40.1	89.7
XTR/WARP	9.5	24K	51.3	50.5	27.2	68K	49.6	70.6	7.9	20K	35.3	63.5	12.8	32K	36.2	88.0
CPU-based methods (using all threads)																
DESSERT	319.5	793K	52.5	48.4	166.6	414K	54.0	67.4	153.4	381K	34.9	58.4	130.3	324K	36.3	78.4
MUVERA	144.1	358K	53.7	50.6	53.2	132K	54.5	69.2	90.2	224K	38.3	67.5	25.7	64K	39.0	87.0
PLAID	50.1	124K	54.3	52.0	36.6	91K	55.3	71.0	44.8	111K	39.4	67.6	32.7	81K	40.5	90.4
EMVB	311.2	773K	53.8	50.3	239.1	594K	55.3	70.2	217.3	539K	38.3	64.8	194.3	482K	40.2	89.2
IGP	455.6	1131K	53.9	50.9	320.5	796K	54.9	70.5	358.9	891K	38.6	66.5	222.5	552K	40.1	89.7
XTR/WARP	90.9	226K	51.3	50.5	239.3	594K	49.6	70.6	75.5	187K	35.3	63.5	115.9	288K	36.2	88.0
GPU-based methods																
PLAID-GPU	133.1	333K	54.3	51.9	82.5	207K	55.3	71.1	141.2	353K	39.3	67.6	84.8	212K	40.5	90.4
GIGP	213.3	534K	53.6	50.7	71.1	179K	55.0	71.2	145.8	365K	38.7	66.3	71.3	179K	40.3	90.3
GIGP$^+$	**1051.5**	**2629K**	53.7	50.7	**599.9**	**1500K**	55.0	71.2	**665.3**	**1663K**	38.7	66.4	**509.2**	**1273K**	40.3	90.3

which encode each vector into a compact high-precision representation. In contrast, DESSERT requires storing extensive hash values for each document vector, and MUVERA needs to store Fixed Dimensional Encodings (FDEs) for all multi-vectors. Both alternative approaches scale directly with dataset size, making them less suitable for large-scale applications (2) All methods exhibit comparable indexing time. This is because the time-consuming part of all methods is accelerated by GPUs, making these steps less of a practical concern. In particular, the most time-consuming step—clustering—is GPU-accelerated in XTR/WARP, PLAID, DESSERT, EMVB, and GIGP$^+$. Similarly, MUVERA employs a sketching-based method that can be parallelized per multi-vector.

5.4 Comparison with CPU-based methods

Following [45, 51], we adopt *queries per dollar* (Q/$) as a key metric for comparing cost-performance against CPU-based methods. It measures the number of queries executable per US dollar spent and is calculated by dividing the number of queries executed per hour by the instance cost per hour. We use two hardware configurations from *Alibaba Cloud* [9] with hourly billing for operating costs: (1) 32 vCPUs with 256 GB memory, costing US$ 1.45 per hour, dedicated to CPU-based methods; (2) 8 vCPUs, 30 GB memory, and one NVIDIA A10 Tensor Core GPU, costing US$ 1.44 per hour, for GPU-based methods.

Table 8 presents a performance comparison with CPU-based methods. Detailed settings of CPU-based methods can be referred to Section 5.1. The results show that GIGP$^+$ achieves a 1.8–2.3× improvement in Q/$ compared to CPU-based methods and 7–11.8× improvements compared to GPU-based methods. This enhancement stems from the effective use of the GPU architecture's inherent high parallelism and low synchronization overhead, combined with reduced computational overhead.

6 Conclusion

In this work, we improve the performance of the multi-vector retrieval problem on GPU. Existing GPU-based solution PLAID-GPU suffers from a large number of candidates. Although state-of-the-art CPU-based method IGP reduces the number of candidates, it fails to fully leverage the massive parallelism and the high memory bandwidth inherent in GPU, leading to suboptimal performance. Motivated by them, we propose GIGP$^+$, a novel solution that incorporates three key optimizations. First, we introduce a highly-parallelized candidate generation kernel, building upon the principles of IGP, achieving a low computation cost. Second, we design a score ordering mechanism that reduces the synchronization across different Streaming Multiprocessors (SM). Third, we propose a scheduling strategy across CPU and GPU for efficient batch processing. Experimental results on standard benchmarks demonstrate that GIGP$^+$ achieves an 11.0× improvement in query per second (QPS) and reduces latency by 7.6× compared to PLAID-GPU. As for cloud pricing, it yields a 2.3× improvement on queries per dollar over CPU-based solutions, at the same retrieval accuracy level. As future work, we plan to extend GIGP$^+$ to support distributed execution across multiple GPUs and compute nodes.

7 Acknowledgment

We sincerely thank the reviewers for their insightful comments. This work was partially supported by National Science Foundation of China (Grant Nos. 62422206, 62532007, 62532001), Hong Kong Research Grants Council (GRF 152043/23E).

References

[1] 2018. Wikipedia Data Source. https://ir-datasets.com/dpr-w100.html.
[2] 2021. MS MARCO, Quora and HotpotQA Data Source. https://huggingface.co/datasets/BeIR/beir.
[3] 2022. *Achieved Occupancy*. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator
[4] 2022. *Atomic Add*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomicadd
[5] 2022. *Cycles Per Instruction*. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#id34
[6] 2022. *Executed Instructions*. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#id37
[7] 2022. Lotte Data Source. https://github.com/stanford-futuredata/ColBERT/blob/main/LoTTE.md.
[8] 2022. *NVIDIA Nsight Compute*. https://docs.nvidia.com/nsight-compute/index.html
[9] 2025. *Alibaba Cloud*. https://www.alibabacloud.com/en?_p_lc=1
[10] 2025. *C++ condition variable*. https://en.cppreference.com/w/cpp/thread/condition_variable.html
[11] 2025. *GTE-ModernColBERT*. https://huggingface.co/lightonai/GTE-ModernColBERT-v1
[12] 2025. *Vespa Long-Context ColBERT*. https://blog.vespa.ai/announcing-long-context-colbert-in-vespa/
[13] 2026. GIGP+ implementation. https://github.com/DBGroup-SUSTech/multi-vector-retrieval.
[14] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. 2023. Similarity Search in the Blink of an Eye with Compressed Indices. *PVLDB* (2023), 3433–3446.
[15] Saman Ashkiani, Martin Farach-Colton, and John D Owens. 2018. A dynamic hash table for the GPU. In *IPDPS*. IEEE, 419–429.
[16] Zheng Bian, Man Lung Yiu, and Bo Tang. 2025. IGP: Efficient Multi-Vector Retrieval via Proximity Graph Index. In *SIGIR*. ACM.
[17] Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. Efficient inverted indexes for approximate retrieval over learned sparse representations. In *SIGIR*. 152–162.
[18] Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. 2025. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *SIGMOD* 3, 3 (2025), 1–26.
[19] Antoine Chaffin and Raphaël Sourty. 2025. Pylate: Flexible training and retrieval for late interaction models. *arXiv preprint arXiv:2508.03555* (2025).
[20] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *STOC*. 380–388.
[21] Joshua Engels, Benjamin Coleman, Vihan Lakshman, and Anshumali Shrivastava. 2023. DESSERT: An Efficient Algorithm for Vector Set Search with Vector Set Queries. In *NeurIPS*.
[22] Thibault Formal, Stéphane Clinchant, Hervé Déjean, and Carlos Lassance. 2024. Splate: Sparse late interaction retrieval. In *SIGIR*. 2635–2640.
[23] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE: Sparse lexical and expansion model for first stage ranking. In *SIGIR*. 2288–2292.
[24] Kaili Huang, Thejas Venkatesh, Uma Dingankar, Antonio Mallia, Daniel Campos, Jian Jiao, Christopher Potts, Matei Zaharia, Kwabena Boahen, Omar Khattab, et al. 2025. ColBERT-serve: Efficient Multi-Stage Memory-Mapped Scoring. In *ECIR*. Springer, 21–30.
[25] Rajesh Jayaram, Laxman Dhulipala, Majid Hadian, Jason D Lee, and Vahab Mirrokni. 2024. MUVERA: Multi-Vector Retrieval via Fixed Dimensional Encoding. *NeurIPS* 37 (2024), 101042–101073.
[26] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* (2011).
[27] Daniel Jünger, Robin Kobus, André Müller, Christian Hundt, Kai Xu, Weiguo Liu, and Bertil Schmidt. 2020. Warpcore: A library for fast hash tables on gpus. In *HiPC*. IEEE, 11–20.
[28] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering. In *EMNLP*.
[29] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *SIGIR*.
[30] Carlos Lassance, Hervé Dejean, Stéphane Clinchant, and Nicola Tonellotto. 2024. Two-step SPLADE: simple, efficient and effective approximation of SPLADE. In *ECIR*. Springer, 349–363.
[31] Jinhyuk Lee, Zhuyun Dai, Sai Meher Karthik Duddu, Tao Lei, Iftekhar Naim, Ming-Wei Chang, and Vincent Zhao. 2023. Rethinking the role of token retrieval in multi-vector retrieval. *NeurIPS* 36 (2023), 15384–15405.
[32] Minghan Li, Sheng-Chieh Lin, Barlas Oguz, Asish Ghoshal, Jimmy Lin, Yashar Mehdad, Wen-tau Yih, and Xilun Chen. 2023. CITADEL: Conditional Token Interaction via Dynamic Lexical Routing for Efficient and Effective Multi-Vector Retrieval. In *ACL*.
[33] Minghan Li, Sheng-Chieh Lin, Xueguang Ma, and Jimmy Lin. 2023. Slim: Sparsified late interaction for multi-vector retrieval with inverted indexes. In *SIGIR*. 1954–1959.
[34] Weizhe Lin, Rexhina Blloshmi, Bill Byrne, Adrià de Gispert, and Gonzalo Iglesias. 2023. Li-rage: Late interaction retrieval augmented generation with explicit signals for open-domain table question answering. In *ACL*. 1557–1566.
[35] Weizhe Lin, Jinghong Chen, Jingbiao Mei, Alexandru Coca, and Bill Byrne. 2023. Fine-grained late-interaction multi-modal retrieval for retrieval augmented visual question answering. *NeurIPS* 36 (2023), 22820–22840.
[36] Weizhe Lin, Jingbiao Mei, Jinghong Chen, and Bill Byrne. 2024. PreFLMR: Scaling Up Fine-Grained Late-Interaction Multi-modal Retrievers. In *ACL*. 5294–5316.
[37] Sean MacAvaney and Nicola Tonellotto. 2024. A reproducibility study of PLAID. In *SIGIR*. 1411–1419.
[38] Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. 2024. Efficient Multi-vector Dense Retrieval with Bit Vectors. In *ECIR*.
[39] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A Human Generated MAchine Reading COmprehension Dataset. In *NeurIPS*, Vol. 1773.
[40] Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. 2022. PLAID: An Efficient Engine for Late Interaction Retrieval. In *CIKM*.
[41] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and Efficient Retrieval via Lightweight Late Interaction. In *NAACL-HLT*.
[42] Jan Luca Scheerer, Matei Zaharia, Christopher Potts, Gustavo Alonso, and Omar Khattab. 2025. WARP: An Efficient Engine for Multi-Vector Retrieval. In *SIGIR*. 2504–2512.
[43] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2018. *Operating System Concepts* (10th ed.).
[44] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models. In *NeurIPS*. https://openreview.net/forum?id=wCu6T5xFjeJ
[45] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Hai Jin, Xuecang Zhang, Junhua Zhu, and Yu Zhang. 2025. Towards High-throughput and Low-latency Billion-scale Vector Search via {CPU/GPU} Collaborative Filtering and Re-ranking. In *FAST*. 171–185.
[46] Vasily Volkov and James W Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC*. IEEE, 1–11.
[47] Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, et al. 2025. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. In *ACL*. 2526–2547.
[48] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. 2021. Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval. In *ICLR*.
[49] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. In *EMNLP*. 2369–2380.
[50] Cheng Zhang, Jianzhi Wang, Wan-Lei Zhao, and Shihai Xiao. 2025. Highly Efficient Disk-based Nearest Neighbor Search on Extended Neighborhood Graph. In *SIGIR*. 2513–2523.
[51] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2024. Fast Vector Query Processing for Large Datasets Beyond {GPU} Memory with Reordered Pipelining. In *NSDI*. 23–40.
[52] Yuxuan Zong and Benjamin Piwowarski. 2025. Towards Lossless Token Pruning in Late-Interaction Retrieval Models. In *SIGIR*. 2407–2417.