BCCE: Block-Centric GPU Co-Design for Real-Time Range-Top-K Query at Scale

Chengying Huan
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
huanchengying@nju.edu.cn

Ziheng Meng
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
zhmeng.cn@gmail.com

Zhengyi Yang
University of New South Wales
Sydney, Australia
zhengyi.yang@unsw.edu.au

Yongchao Liu
Ant Group
Hangzhou, China
yongchao.ly@antgroup.com

Jie Zhang
Peking University
Beijing, China
jiez@pku.edu.cn

Qing Wang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
wangqing.cs@nju.edu.cn

Jing Wang
Shanghai Jiao Tong University
Shanghai, China
jing618@sjtu.edu.cn

Shaonan Ma
Qiyuan Lab
Beijing, China
mashaonan@qiyuanlab.com

Zhibin Wang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
wzbwangzhibin@gmail.com

Mingxing Zhang
Tsinghua University
Beijing, China
zhang_mingxing@mail.tsinghua.edu.cn

Rong Gu
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
gurong@nju.edu.cn

Baokun Wang
Ant Group
Hangzhou, China
yike.wbk@antgroup.com

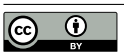
Guihai Chen
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
gchen@nju.edu.cn

Chen Tian
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
tianchen@nju.edu.cn

Abstract

Range-top-k queries retrieve the top-k elements within an arbitrary subrange of a large array and are a key primitive in real-time analytics. Unlike one-shot top-k selection, practical deployments issue large volumes of queries over varying and often overlapping ranges, frequently interleaved with streaming updates. In this setting, applying conventional GPU top-k kernels per query is inefficient: each query triggers range rescans or $O(n)$-scale passes that overwhelm HBM bandwidth, thrash on-chip caches, and provide little reuse across overlapping windows.

We present BCCE, a GPU-co-designed, block-centric engine that makes range-top-k efficient by exposing a reusable intermediate representation of the data. BCCE partitions the array into locally sorted blocks and builds a compact interval-aware auxiliary index, reducing each query to a small set of contiguous active slices that remain amenable to SIMT execution. Queries are answered via a two-layer search: a global rank-thresholding step identifies the candidate value interval, followed by block-local verification restricted to the corresponding slices. This design constrains the active working set to $O(\sqrt{n})$ and achieves $O(\sqrt{n}\log n)$ per-query time with largely coalesced accesses and high on-chip reuse. To further improve throughput, BCCE employs a DP-based cache placement policy to keep hot slices resident in L2 or shared memory, and a



HPDC '26, Cleveland, OH, USA

ACM ISBN 979-8-4007-2640-8/26/07
https://doi.org/10.1145/3806645.3807585

range-grouped batching scheme that amortizes PCIe transfers for out-of-core datasets by reusing fetched slices across queries. Finally, BCCE supports incremental, block-local insertions and deletions without global rebuilds, sustaining performance under continuous data evolution. Across 17 datasets, including up to 70B elements (256 GB), BCCE achieves sub-millisecond query latency and up to 56,308× higher throughput than state-of-the-art GPU baselines, while performing billion-scale dynamic updates in milliseconds.

CCS Concepts

• **Computer systems organization → Parallel architectures**; • **Computing methodologies → Parallel algorithms**.

Keywords

GPU computing, Range-Top-k, Parallel algorithms

ACM Reference Format:

Chengying Huan, Ziheng Meng, Zhengyi Yang, Yongchao Liu, Jie Zhang, Qing Wang, Jing Wang, Shaonan Ma, Zhibin Wang, Mingxing Zhang, Rong Gu, Baokun Wang, Guihai Chen, and Chen Tian. 2026. BCCE: Block-Centric GPU Co-Design for Real-Time Range-Top-K Query at Scale. In *The 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '26), July 13–16, 2026, Cleveland, OH, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3806645.3807585

1 Introduction

Top-k queries aim to identify the k smallest or largest elements from a dataset and serve as a core operation in numerous application domains, including online analytical processing (OLAP) [13, 27, 55, 57], approximate nearest neighbor search (ANNS) [1, 7, 25, 51, 52], retrieval-augmented generation (RAG) [14, 29, 53], sparse attention in long-context LLM inference [8, 32], scientific computing [22, 31, 41], information retrieval [10, 35, 38, 56], and recommendation systems [3, 6, 9, 18, 24, 44, 54].

While top-k queries have been extensively studied in the literature, many real-world systems issue a massive number of top-k queries over *varying subranges* of a large dataset. These queries are used to extract the most relevant or high-impact entries under strict real-time constraints. We refer to this workload as the **Range-Top-k Query**: given a specified subrange $[l, r]$ of the data array, the goal is to retrieve the top-k elements within that range. Range-top-k queries underpin a wide spectrum of real-time analytics workloads operating on high-volume, high-velocity data. In fraud detection, systems continuously retrieve the top-k highest-risk transactions within varying time windows for risk scoring. In dynamic Approximate Nearest Neighbor Search (ANNS) [30, 42, 43, 59], top-k vectors must be retrieved within temporal windows to support applications such as retrieval-augmented generation (RAG) and real-time video recommendations. Range-top-k queries likewise drive online marketing and personalization pipelines, where recent user interactions are ranked to identify the most relevant customer segments.

Example. A representative use case arises in real-time fraud detection (e.g., money laundering) for online payment platforms. Fraudsters often evade detection by dispersing illicit funds across many small transactions or by mixing small and large ones, thereby weakening traditional risk-scoring mechanisms. Modern risk-control systems therefore examine recent transaction histories and extract the top-k highest-value transactions within a given time window for risk assessment [33]. Listing 1 shows an SQL query retrieving the top 10 transactions for a user in the past 24 hours. For efficient processing, each user's transactions are stored in an append-only array, enabling the query to be executed as top-k selection over the subrange $[l, r]$ corresponding to the time window.

```
SELECT *
FROM transactions
WHERE user_id = 'USER_ID'
  AND transaction_time BETWEEN CURRENT_TIMESTAMP – INTERVAL 24
      HOUR AND CURRENT_TIMESTAMP
ORDER BY amount DESC
LIMIT 10;
```

Listing 1: Example of a Range-Top-k query in SQL

Existing Methods. Prior work on GPU top-k processing has largely focused on *one-shot* selection over the *entire* input array. To meet latency demands, these algorithms are commonly implemented on GPUs [26, 48, 60], leveraging massive SIMT parallelism and high HBM bandwidth. However, their algorithmic structure is fundamentally *query-ephemeral*: each invocation consumes the input once, produces the answer, and discards intermediate states. As a result, these methods provide limited opportunities to (i) amortize work across *many* queries, (ii) reuse computation under *overlapping* ranges, or (iii) support *continuous* updates without recomputation. Current GPU top-k engines can be broadly grouped into three paradigms. *(i) Sorting-based* methods sort the entire dataset and then take the top-k [4, 28, 49], incurring $O(n \log n)$ time and fully materializing the global order. *(ii) Partial-sorting-based* methods (e.g., BlockSelect [26] and Bitonic Top-K [48]) reduce the sorting scope using bitonic networks [5], typically achieving $O(n \log^2 k)$ and working well for small k under one-shot invocation. *(iii) Partition-based* methods [2, 11, 46, 58] perform iterative bucketization (e.g., radix-style) to isolate the top-k region; RadixSelect [18] can reach $O(n)$ time but may require non-trivial CPU participation, while Air Top-K [58] moves the iterations onto the GPU to reduce CPU–GPU communication, yet still executes $O(n)$-scale passes per query. Overall, these designs are effective when the objective is *one* top-k over *one* array snapshot, but they do not directly optimize for range restriction, reuse, or dynamic maintenance.

Limitations in the Range-Top-k Setting. Range-top-k workloads differ from one-shot selection in a crucial way: systems must serve *a large stream of queries* over *varying subranges* of a shared dataset, often with strong temporal locality (overlapping windows) and frequent data updates. In this setting, a one-shot kernel becomes the *wrong unit of work*: it (i) cannot reuse intermediate information across overlapping ranges, (ii) stresses the GPU hierarchy with large, non-resident working sets (especially out-of-core), and (iii) collapses under streaming updates that trigger global recomputation. We distill these challenges into three key observations and corresponding design objectives in Section 3.

Contributions. To bridge algorithmic and architectural gaps of one-shot GPU top-k kernels in the *range-top-k* regime, we propose **BCCE**, a GPU–co-designed **B**lo**C**k-**CE**ntric engine that jointly optimizes (i) the per-query computation model, (ii) the GPU memory hierarchy and out-of-core execution, and (iii) dynamic maintenance under streaming updates. The central idea is to expose a *reusable*

Table 1: Time and Space (Runtime) Complexity Comparison.

	GridSelect [58], Bitonic Top-K [48]	RadixSelect [18], Air Top-K [58]	**BCCE (Ours)**
Time	$O(n \log^2 k)$	$O(n)$	$\boldsymbol{O(\sqrt{n} \log n)}$
Space	$O(n)$	$O(n)$	$\boldsymbol{O(\sqrt{n})}$

intermediate representation by a compact global auxiliary index coupled with block-local ordering that reduces each query to a small set of *contiguous active slices* while preserving SIMT-regular execution. Thus, BCCE limits each query's *active working set* to $O(\sqrt{n})$ and its runtime to $O(\sqrt{n} \log n)$ (see Table 1), achieves sub-millisecond latency and high QPS at billion-scale, and supports block-local incremental updates without global rebuilds. Our key contributions are:

- **GPU-Co-Designed Hierarchical Execution Model.** We introduce a block-centric execution model for range-top-k queries that partitions the array into locally sorted blocks and answers each query via a *two-layer* search guided by a compact *interval-aware auxiliary index*. The first layer performs *global rank-thresholding* across covered blocks to identify the candidate value interval of the k-th order statistic, and the second layer performs *block-local verification* restricted to the corresponding active slices. This structure transforms per-query work from full-range rescans to slice-restricted searches, yielding $O(\sqrt{n} \log n)$ per-query complexity while maintaining SIMT-friendly kernels (uniform per-thread control flow, balanced per-block work, and high occupancy).
- **Memory-Efficient Querying with Cache- and I/O-Aware Optimization.** BCCE confines the GPU-side query footprint to $O(\sqrt{n})$ by accessing only the active slices implied by the auxiliary index, enabling these slices to remain cache-resident and to be accessed with largely coalesced memory patterns. Building on this block-centric locality, we develop a *dynamic programming (DP)-based cache placement policy* that prioritizes high-frequency slices for on-chip residency (L2/shared memory), reducing global-memory traffic and mitigating memory-stall-induced occupancy loss. For out-of-GPU-memory datasets, we further propose *range-grouped batching* that clusters queries by interval ID so that each required slice is transferred once and reused across many queries, amortizing PCIe transfers and enabling overlap between data movement and GPU computation.
- **Incremental, Block-Local Dynamic Maintenance under Updates.** BCCE supports streaming insertions and deletions without global rebuilds by making both the block layout and the auxiliary index *incremental-friendly*. Updates are localized to the affected blocks: insertions append or locally modify blocks, while deletions remove or adjust impacted regions; the auxiliary index is maintained through lightweight prefix-difference adjustments with constant-time updates per block. This yields low maintenance overhead with $O(m \log n)$ for inserting m elements and $O(\sqrt{n})$ for deletions, while preserving locality and sustaining query throughput under continuous data evolution.
- **Scalability and End-to-End Performance at Scale.** We implement BCCE on modern GPUs and evaluate it on 17 datasets (16 synthetic and 1 real) with up to **70 B** elements. BCCE achieves **sub-millisecond** latency and up to 56,308.3× higher throughput than state-of-the-art CPU and GPU baselines, with consistent gains across in-memory and out-of-core settings as well as under dynamic updates. In individual comparisons, BCCE attains speedups of 14659.3×, 771.1×, 957.7×, 1,502.4×, 13,964.1×, and 52,433.1× over Wavelet Trees, GridSelect, RadixSelect, Air Top-K, Bitonic Top-K, and BlockSelect, respectively.

2 Background

2.1 Problem Definitions

We denote a one-dimensional array as $A[1 \dots n]$ or simply $A[]$, where n is the total number of elements. Each element at position i is accessed using index notation $A[i]$, where $1 \le i \le n$. The subarray $[A[l], \dots, A[r]]$ refers to the contiguous segment of A from index l to r, inclusive, with $1 \le l \le r \le n$.

Definition 1 (Top-k Query). *Given an input data array $A[]$, the* top-k query *retrieves the k largest (or smallest) elements from the full array $[A[1], \dots, A[n]]$.*

In this paper, we extend the top-k query to a more practical setting: the *range-top-k query*, which retrieves the k largest (or smallest) elements within a specified subarray.

Definition 2 (Range-Top-k Query). *Given a query (l, r, k) over an input array $A[]$, the* range-top-k query *returns the k largest (or smallest[1]) elements in the subarray $[A[l], \dots, A[r]]$.*

2.2 GPU Architecture

GPUs are widely adopted in top-k query processing to achieve low latency and high throughput [15, 18–20, 23, 58]. In this subsection, we introduce the architecture of NVIDIA GPUs, which are specifically designed for high-throughput parallel computation.

Thread Hierarchy. A GPU *kernel* launches a *grid* of *thread blocks* (CTAs). Blocks are scheduled onto Streaming Multiprocessors (SMs) and contain *warps* of 32 threads executing in SIMT. Parallel efficiency is governed by warp utilization and *occupancy* (the fraction of active warps on an SM), which in turn depends on per-block use of registers and shared memory.

Memory Hierarchy. Modern GPUs employ a multi-level memory hierarchy comprising off-chip high-bandwidth memory (HBM, or global memory), a unified on-chip L2 cache that buffers global-memory accesses from all SMs, and per-SM shared memory. Global memory offers large capacity but incurs high latency, whereas shared memory is a small programmer-managed scratchpad with lower latency and higher on-chip bandwidth, enabling efficient data reuse within a thread block and reducing pressure on L2 and HBM.

3 Motivation and Design Principles

3.1 Motivating Workload: Range-Top-k at Scale

Range-top-k queries retrieve the top-k elements within an arbitrary subrange $[l, r]$ of a large array and serve as a foundational primitive in real-time analytics. In contrast to one-shot top-k selection, practical deployments must sustain (i) *high query volume* over (ii) *varying and often overlapping ranges*, frequently under strict tail-latency

[1] Since finding the k smallest elements trivially reduces to finding the k largest, we focus on the top-k largest case only throughout this work.

objectives and in the presence of (iii) *continuous updates* (appends, deletions, or sliding-window maintenance). These characteristics fundamentally change what dominates performance: the critical bottleneck is no longer a single kernel's throughput on a static array, but rather the system's ability to amortize work and data movement across a stream of correlated queries while preserving predictable GPU execution.

State-of-the-art GPU top-k kernels are optimized for *query-ephemeral full-pass execution*: each query triggers substantial scanning, partitioning, or partial sorting over the input (or the query subrange), then discards intermediate state. This structure becomes inefficient in the range-top-k regime for three reasons.

No GPU-Efficient Reuse Model for Many Overlapping Ranges. State-of-the-art GPU top-k kernels are organized as *full-pass* procedures: each query triggers scan or sorting passes that touch a substantial fraction of the queried data, and often degenerates to rescanning large regions. For example, Air Top-K performs $O(n)$ histogram and scan passes per query [58], effectively *restarting* the selection pipeline for every subrange. This is fundamentally mismatched to range-top-k workloads, where (i) adjacent queries overlap heavily and (ii) the system must sustain high QPS under tight per-query deadlines. Repeated full-pass executions saturate HBM bandwidth and offer almost no computational reuse across overlapping ranges. Empirically, on a 256 GB dataset (Fig. 1), Air Top-K reaches **3,200 ms** per query (compute only), scaling linearly with n. Achieving real-time range-top-k thus calls for a *different* GPU model:one that materializes a *reusable* intermediate representation (beyond transient per-query kernel state) while maintaining SIMT regularity. Naïvely adding indexing or pruning on GPUs is challenging: irregular control flow causes warp divergence, range-dependent work leads to load imbalance, and large per-query state reduces occupancy.

High I/O Overhead and Cache Thrashing. Range-top-k systems frequently operate on datasets that exceed GPU HBM capacity, forcing query-time data movement from host memory and amplifying I/O costs. Under high query volume, overlapping ranges further exacerbate cache thrashing: hot regions are repeatedly evicted and re-fetched, while the GPU spends most cycles stalled on memory. In the out-of-core regime, the system must *amortize* host–device transfers across multiple queries rather than incurring PCIe costs per query. Besides, one-shot pipelines typically maintain $O(n)$-scale runtime states (e.g., histograms, partitions, or per-pass buffers) and assume relatively uniform device-memory access, which prevents effective residency in L2 cache and yields minimal shared memory optimization. In our measurements (Fig. 1), Air Top-K consumes **463 GB** of device memory and incurs **9,088 ms** of I/O on a billion-scale dataset. Thus, the core requirement is to reduce each query's *active working set* to a scale that can be served by the GPU hierarchy (shared/L2/HBM), enabling stable reuse under concurrency.

Static Designs Collapse under Streaming Updates. Real-time analytics pipelines commonly interleave queries with continuous inserts and deletes. In one-shot top-k designs, even a modest update can invalidate intermediate artifacts and typically forces either (i) full recomputation or (ii) global rebuilds of derived structure, each triggering kernel relaunches, and large data movement. Under frequent

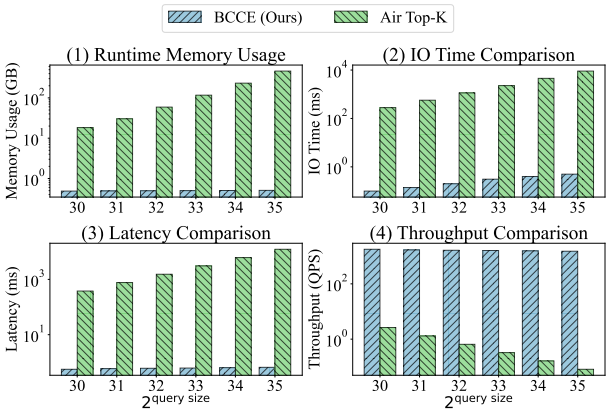


Figure 1: Comparison with AIR Top-K.

updates, these costs accumulate rapidly and overwhelm throughput. For example, on a billion-scale dynamic dataset with continuous query arrivals, Air Top-K achieves only **0.08 QPS**. A practical range-top-k engine must therefore support *fine-grained, incremental* maintenance that localizes work to small regions, avoids global rescans or rebuilds, and preserves locality so that query throughput remains stable as the data evolve.

Overall, treating each range-top-k query as an isolated one-shot selection kernel uses the GPU as a bandwidth-bound scanner rather than a locality-aware query engine.

3.2 Key Observations

We distill the workload into three observations based on the limitations of current works when applied to range-top-k query.

Observation 1: Overlap implies reuse, but only if the system exposes a reusable representation. Adjacent or concurrent range queries often share substantial overlap (e.g., sliding windows or bursty requests over nearby indices). Overlap is an opportunity only when the system maintains auxiliary structure that lets neighboring queries avoid repeating the same passes over data.

Observation 2: The GPU memory hierarchy rewards small, and reusable working sets. GPUs reach high efficiency when hot data can remain resident in L2 or shared memory for a longer time. In practice, Range-top-k becomes practical when each query can be reduced to a small set of contiguous slices, enabling efficient reuse across queries and batches.

Observation 3: Streaming updates must be handled as part of steady state. In real-time analytics, updates are not exceptional events. Any design that requires global rescans or rebuilds per update will collapse once queries and updates are interleaved at scale. Sustained throughput requires maintenance costs proportional to the affected locality, not proportional to the full dataset size.

Design Objectives. From these observations, we set three objectives that shape the system: **(1) Eliminate rescans:** avoid $O(|r-l+1|)$ full-range passes per query by pruning irrelevant data early; **(2) Exploit hierarchy:** shape query access so the active footprint fits in the GPU memory hierarchy and is accessed largely coalesced, enabling on-chip reuse under concurrency; **(3) Incremental maintenance:** support updates with localized work, sustaining throughput under continuous data evolution.

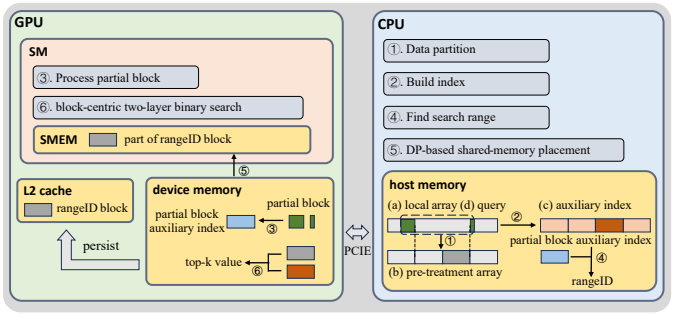


Figure 2: Workflow of BCCE.

3.3 Design Principles

To address the mismatch between one-shot GPU top-k kernels and range-top-k workloads, we distill the design into three principles that directly guide the system in Section 4.

P1: Hierarchical Pruning, Followed by Verification. Range-top-k must avoid the $O(|r-l+1|)$ full-pass work for each individual query. The engine should first *prune globally* to narrow the candidate value region (or candidate contributors), and then *verify locally* within the pruned region to produce the exact top-k results. This structure eliminates redundant rescans while maintaining the predictability of the overall query plan.

P2: Block-Centric Layout with Explicit Active Slices. GPU efficiency depends on regularity and contiguity. The engine should organize data into fixed-size blocks with block-local order and reduce each query to a small set of contiguous *active slices* within covered blocks. Active slices are the appropriate unit for SIMT execution and reuse; they enable bounded per-query footprints, and effective on-chip caching under overlapping queries.

P3: Reuse-aware Execution Under Concurrency and Updates. Range-top-k workloads are dominated by many-query throughput and steady-state operation under streaming updates. The engine should exploit reuse by batching or scheduling queries so that shared active slices are fetched and processed once and reused across multiple queries, especially in out-of-core settings. Simultaneously, maintenance should be localized so that updates modify only the affected blocks and auxiliary metadata, avoiding global rebuilds that would destabilize throughput.

Implications for System Design. These principles lead to a system that (i) maintains block-local order to enable efficient local verification, (ii) builds a compact auxiliary structure to identify active slices for effective global pruning, (iii) executes a hierarchical query plan that is naturally SIMT-friendly, (iv) co-optimizes cache residency and out-of-core batching around active slices for higher efficiency, and (v) supports block incremental updates with low maintenance overhead.

4 BCCE's System Design

4.1 Overview

Based on the above design principles, this paper proposes a GPU-co-designed range-top-k engine, BCCE, that unifies efficient computation, memory optimization, and dynamic maintenance within a single execution model. Fig. 2 sketches the overall workflow of BCCE: a lightweight *preprocessing* step builds a block-sorted layout and a compact interval-aware index, while *online execution* answers queries using a hierarchical execution model that incorporates a two-layer binary search.

Algorithm 1 BCCE Workflow.

Input: Array A; list $Queries$ of triples (l, r, K)
Output: TopK value V

```
1:  𝔸, ℝ, ℍ_2D ← PREPROCESS(A)
2:  for all Q ∈ Queries do
3:      intervalID ← FIND_INTERVALID(ℝ, ℍ_2D, Q)
4:      Q ← (l, r, K, intervalID)
5:  Queries ← SORT_BY_INTERVALID(Queries)
6:  for all Interval[I] ∈ ℝ do
7:      for all Q ∈ Queries where Q.intervalID == I do
8:          PROCESS_QUERY(Q, A, ℍ_2D)
9:  function PREPROCESS(A)
10:     𝔸 ← COMPUTE_PRETREATMENT_ARRAY(A)
11:     ℝ ← COMPUTE_DISTRIBUTION(A)
12:     ℍ_2D ← COMPUTE_AUXILIARY_INDEX(𝔸, ℝ)
13:     return 𝔸, ℝ, ℍ_2D
```

- **Preprocessing.** BCCE partitions the array into fixed-size *blocks* $\{B_1, \ldots, B_{N_B}\}$ and sorts each block in parallel on the GPU to form the *pre-treatment array*. It then discretizes the value domain into $\sqrt{n}$ intervals and builds a *interval-aware auxiliary index* $\mathbb{H}_{2D}$ (per-{range,block} prefix tallies), laid out to favor coalesced reads across blocks. Preprocessing cost is amortized over many top-k queries.
- **Online Execution.** Given a query (l, r, k), (i) BCCE identifies fully covered blocks and up to two *partial* edge blocks P; if needed, it locally sorts P on the GPU (radix sort [50]) to match block order; (ii) probes the **interval-aware index** via a binary search for the *intervalID* that contains the k-th order statistic and computes, per covered block, the contiguous *active slice* (Pos, Len) inside that range; (iii) performs a **block-centric two-layer binary search**: inter-block rank-thresholding followed by intra-block verification restricted to active slices, yielding workload balance. For batches, BCCE applies **range-grouped batching**: queries are sorted by *intervalID*, ach range's slices are transferred once asynchronously from host to device, and all queries for that range are served before moving on, reducing PCIe traffic and boosting QPS. In addition, A **DP-based shared-memory placement** pins the most frequently visited slices to maximize on-chip reuse and occupancy, complementing range-grouped batching to further boost throughput.

4.2 SIMT-aware Two-layer Binary Search

Building on the block-centric framework, we co-design a SIMT-aware hierarchical execution model on GPUs named *block-centric two-layer binary search*, which couples inter-block rank thresholding with intra-block verification on contiguous and cacheable slices to solve the recomputation challenge. It answers (l, r, k) with $O(\log n)$ probes while restricting computation to $O(\sqrt{n})$ active elements. We prefer binary search to linear probing because its lower time complexity more than compensates for the warp-divergence overhead. In addition, $\log n$ varies much less than $\sqrt{n}$ across workloads, yielding more balanced work under binary search.

Core Idea. We partition A into fixed-size, locally sorted blocks $\{B_1, \ldots, B_{N_B}\}$ (the *pre-treatment array*). Given a query, we first prune the blocks intersecting $[l, r]$ and use the interval-aware index

Algorithm 2 Block-Centric Two-layer Binary Search.

```
1:  function PROCESS_QUERY(Query Q, Array A, auxiliary index H_2D)
    /* process partial block P */
2:      C ← {P, B_i | P, B_i is covered by Q}
3:      P ← SORT(P)
    /* first-layer binary search */
4:      Initialize L ← MIN, R ← MAX
5:      while L < R do
6:          M ← (L + R)/2
7:          for all B ∈ C in parallel do
    /* second-layer binary search */
8:              local_count ← BINARY_SEARCH(POS, LEN, M)
9:          global_count ← REDUCE(local_count)
10:         if global_count < K then
11:             R = M
12:         else
13:             L = M
14:     return L
```

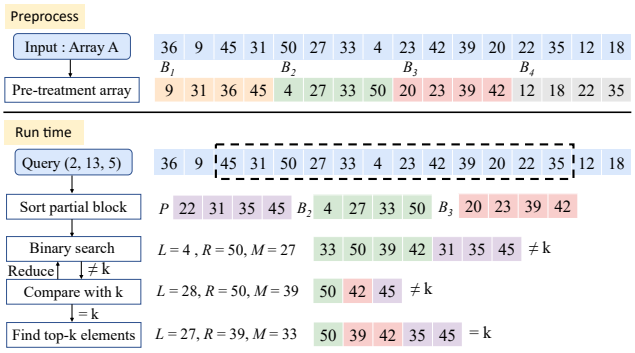


Figure 3: Workflow of two-layer binary search.

to compute their per-slice counts. A *global* binary search over the *value domain* finds the largest threshold M whose count of elements $\geq M$ within $[l, r]$ is at least k. We then *locally* verify M's rank by a binary search within each candidate slice to produce the exact top-k.

Algorithmic Workflow. Let S denote the multiset of active slices (one per covered block, at most two partial edges):

Step 1: Identify the covered block set $\mathbb{C}$ for each query, including the complete blocks B_i and the partial blocks P at the edges, which are denoted as $\mathbb{C} = \{P,\ B_i \mid l \leq L(B_i), R(B_i) \leq r\}$. Then, we will sort P to match the order of the complete blocks for binary search.

Step 2: Apply the two-layer binary search on all covered blocks in $\mathbb{C}$ to determine the top-k largest elements. Specifically, the two-layer process is as follows:

(1) **First Layer.** Initialize the search with L = Min and R = Max, where Min and Max denote the minimum and maximum elements of the array. Compute the midpoint $M = \frac{L+R}{2}$ as an estimate for the k-th largest element.

(2) **Second Layer.** For the elements of each block $B_i \& P \in \mathbb{C}$, perform a binary search to count the number of elements greater than M, denoted as S_i.

(3) **Reduce.** Aggregate the counts using a reduction operation to compute $S = \sum S_i$. If $S < K$, the candidate M is too large, then we will update the search range by setting $R = M$; otherwise, setting $L = M$.

This process repeats until $L \geq R$, at which point L corresponds to the K-th largest element. The full set of top-k largest elements

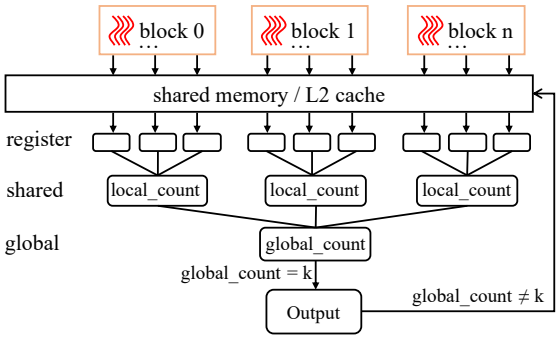


Figure 4: Parallel two-layer binary search on GPU.

can then be derived based on this threshold. Note that the above workflow assumes that the data array consists of integer values. For floating-point data, a minor modification is needed. Specifically, updating the binary search procedure to use $L = M$ and $R = M$ ensures convergence under non-discrete value distributions.

Case Study. Fig. 3 illustrates the workflow of the block-centric two-layer binary search. In preprocessing, we partition the array A into 4 equal-size blocks, sort each block, and obtain the pre-treatment array $\mathbb{A} = \{B_1, B_2, B_3, B_4\}$. For the top-$K$ query $(2, 13, 5)$ with range $[2, 13]$ and $K = 5$, blocks B_2 and B_3 are fully covered, while the boundary elements from B_1 and B_4 form a partial block $P = \{22, 31, 35, 45\}$. We therefore search in the covered block set $\mathbb{C} = \{B_2, B_3, P\}$. The first-layer binary search maintains a value range $[L, R] = [4, 50]$ for the K-th largest element and iteratively updates the midpoint $M = (L + R)/2$. For each M, the second layer performs a binary search within every block in $\mathbb{C}$ to count elements greater than M. When the count is larger than K (e.g., 7 for $M = 27$), we increase L; otherwise, we decrease R. Eventually, at $M = 33$, we obtain exactly 5 elements greater than M, so 33 is identified as the queried K-th largest value.

The block size is set to $\sqrt{n}$, which is provably optimal for both time complexity and parallel efficiency (Theorem 1).

THEOREM 1. *Choosing a block size of $|B_i| = \sqrt{n}$ yields an optimal time complexity of $O\left(\sqrt{n}\log^2 n\right)$.*

PROOF. Assume the input array A has a size of n, and it is partitioned into blocks of size m. The time complexity for sorting each partial block P is $O(2m \log m)$. For the two-layer binary search, the first layer requires $O(\log n)$ time, and the second layer, which performs a binary search within each block, requires $O(\log m)$ time. Given that there are $\frac{n}{m}$ blocks, the overall time complexity for the two-layer binary search is $O\left(2m \log m + \frac{n}{m} \cdot \log n \cdot \log m\right)$.

By taking the derivative with respect to m, one can show that setting $m = \sqrt{n}$ will yield near-optimal time complexity. Substituting this optimal block size yields an overall time complexity of $O\left(\sqrt{n} \cdot logn + \frac{\sqrt{n} \cdot \log^2 n}{2}\right)$, which is on the order of $O(\sqrt{n} \cdot \log^2 n)$. This completes the proof. □

Parallel Execution on GPU at Scale. When running on a GPU, the intra-block binary search runs warp-synchronously, with each thread operating with no inter-warp communication. As illustrated in Fig. 4, inter-block rank thresholding issues a single reduction per probe, avoiding data contention.

4.3 Interval-aware Auxiliary Index

Despite the SIMT-friendly mapping, a vanilla two-layer binary search still wastes work: it repeatedly probes *irrelevant* blocks and effectively requires materializing (or streaming) the entire array, yielding an $O(n)$ working set and amplified I/O, even though each probe is $O(\log n)$. To further reduce end-to-end latency and memory usage, we introduce an *interval-aware auxiliary index*, denoted as $\mathbb{H}_{2D}$, which efficiently narrows the top-k search space.

Auxiliary Index Construction. Formally, we first partition the data distribution of the array noted as $\mathbb{R}$ into several intervals, each containing an equal number of elements (typically around $\sqrt{n}$ elements per interval). Specifically, if the data distribution of the array is partitioned into several intervals:

$$\mathbb{R} = \{\text{Interval}[1], \text{Interval}[2], \cdots, \text{Interval}[\sqrt{n}]\},$$

each $|\text{Interval}[i]|$ corresponds to the length of $\sqrt{n}$. Secondly, we construct a two-dimensional auxiliary index $\mathbb{H}_{2D}$ of size n, with each $\mathbb{H}_{2D}[i][j]$ representing an aggregated value related to the prefix sum of the histogram for $\text{Interval}[i]$ within block B_j.

The auxiliary index construction follows a two-step process. First, for each block j, we compute the histogram $\mathbb{H}_j$, where $\mathbb{H}_j[i]$ represents the number of elements falling within intervalID i in block j and these elements are noted as **active data**. Next, we compute the inclusive prefix sum of $\mathbb{H}_j$, denoted as $\mathbb{S}_j$. As illustrated in Fig. 5, the array is divided into four predefined intervals: $[4, 20)$, $[20, 31)$, $[31, 39)$, and $[39, 50]$, and is further partitioned into four blocks. The histogram $\mathbb{H}_j$ for each block j is then computed based on these intervals. For example, consider the first block containing elements $\{9, 31, 36, 45\}$. Within the specified ranges, this block contains 1 element in $[4, 20)$, 0 elements in $[20, 31)$, 2 elements in $[31, 39)$, and 1 element in $[39, 50]$. Consequently, the histogram $\mathbb{H}_j = [1, 0, 2, 1]$ and its inclusive prefix sum $\mathbb{S}_j = [1, 1, 3, 4]$. To facilitate coalesced memory access during computation, we organize the auxiliary structure $\mathbb{H}_{2D}$ with each *intervalID* in a row. The $\mathbb{H}_{2D}$ is iteratively constructed by accumulating prefix sums across blocks:

$$\mathbb{H}^T_{2D}[j] = \begin{cases} \mathbb{S}_0, & j = 0 \\ \mathbb{H}^T_{2D}[j-1] + \mathbb{S}_j. & j > 0 \end{cases} \quad (1)$$

Two-layer Binary Search Optimized By Auxiliary Index. For each range-top-k query, we first search the *intervalID* by performing a binary search over the auxiliary index (line 6 in Algorithm 1). Specifically, we compute the cumulative count of elements larger than the current *intervalID* for the given query. By comparing this value with k, we iteratively adjust the binary search bounds L and R until convergence. The final *intervalID* corresponds to the range that contains the k-th largest element. Subsequently, during the two-layer binary search, we restrict the search to the specific *intervalID* interval associated with the current block ($I = intervalID,\ J = BlockID$), which starts at the position Pos (Equation 2) and spans a length Len (Equation 3), rather than searching the entire block, which can significantly reduce memory usage.

$$Pos = \text{Interval}[I] + \mathbb{H}_{2D}[I, J+1] - \mathbb{H}_{2D}[I, J], \quad (2)$$

$$Len = \mathbb{H}_{2D}[I, J] + \mathbb{H}_{2D}[I+1, J+1] - \mathbb{H}_{2D}[I, J+1] - \mathbb{H}_{2D}[I+1, J]. \quad (3)$$

Case Study. As shown in Fig. 5, for the range-top-k query $(2, 13, 5)$, the covered blocks are B_2 and B_3, along with the partial block

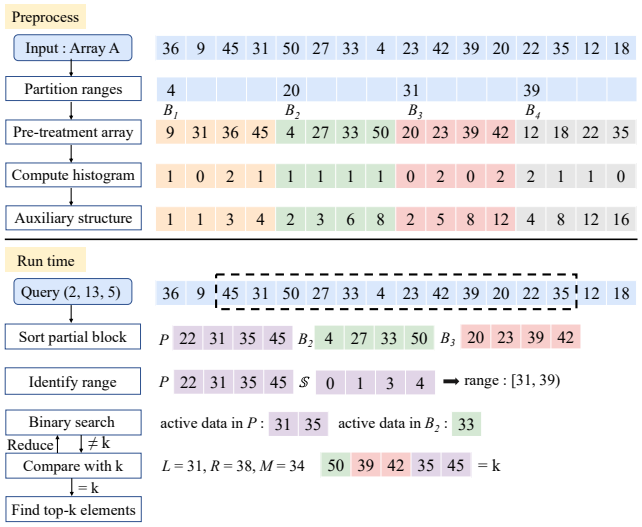


Figure 5: Workflow of two-layer binary search optimized by the interval-aware auxiliary index.

$P = \{22, 31, 35, 45\}$. The histogram of the partial block will be $\{0, 1, 2, 1\}$ and prefix sum is $\mathbb{S} = \{0, 1, 3, 4\}$. Using both the auxiliary structure $\mathbb{H}_{2D}$ and the partial block histogram $\mathbb{S}$, we determine that the search range for this query is $[31, 39)$. Then, we only apply the two-layer binary search on the sub-array of each block within the range $[31, 39)$ which is noted as active data for range $[31, 39)$, e.g., $\{31, 35\}$ for P and $\{33\}$ for B_2. In addition, the FIND_INTERVALID function is used for identifying the corresponding range for each top-k query in Algorithm 1 via a binary search over the auxiliary index, which runs in $O(\log n)$ time complexity.

THEOREM 2. *With auxiliary index, the parallel time complexity of the top-k search on the GPU is reduced to $O(\sqrt{n}\log n)$, while the runtime space complexity is reduced to $O(\sqrt{n})$.*

PROOF. Let S_i be the number of elements in each block B_i within the range of a given *intervalID*. The two-level search first locates the candidate blocks in $O(\log n)$ time and then performs a binary search within each block in $O(\sum_i \log S_i)$ time, for a total of $O(\sum_i \log S_i \cdot \log n)$.

Since each range contains at most $\sqrt{n}$ elements, we have $\sum_i S_i \le \sqrt{n}$, and thus $\sum_i \log S_i \le \sum_i S_i \le \sqrt{n}$. The partial sorting over these elements, therefore, costs $O(\sqrt{n}\log n)$, yielding a total per-query complexity of $O(\sqrt{n}\log n)$. Our co-designed execution model is data-independent and achieves an effective practical cost of $O\left(\frac{\sqrt{n}\log n}{\#\text{GPU Threads}}\right)$ under the GPU's massive parallelism, resulting in *near-constant latency*. For example, when executed on an A100 GPU ($216 * 1024$ threads in total) over a billion-scale dataset with 2^{31} elements, the query latency remains below 0.16 ms.

Regarding space, we only access the *intervalID* slice of each relevant block. Since $\sum_i S_i \le |\text{Range}[intervalID]| = \sqrt{n}$, the additional space usage is $O(\sqrt{n})$. □

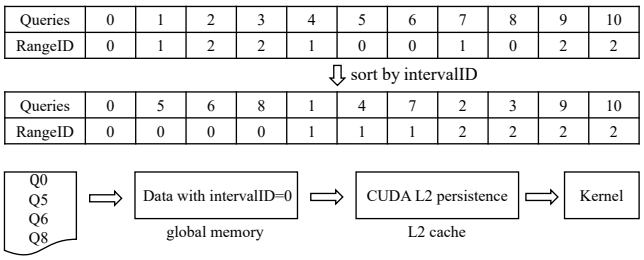


Figure 6: Batch optimization for out-of-GPU-memory.

5 I/O Reducing and Cache Locality Optimization

Having addressed the recomputation challenge with our SIMT-aware hierarchical execution, this section tackles on-chip capacity mismatch and I/O amplification through batching and a locality-optimized cache hierarchy.

5.1 Range-Grouped Batching for I/O Reducing

For large scale datasets that exceed the capacity of GPU HBM, BCCE requires transferring only $\sqrt{n}$ elements from DRAM to GPU HBM with the help of the interval-aware auxiliary index. In contrast, existing methods must scan the entire array, resulting in $O(n)$ data transfers between the CPU and GPU, which contribute to execution time due to high I/O overhead.

Furthermore, when handling a batch of top-k queries, we can group queries with the same $intervalID$, ensuring that the data of each block inside $Interval[intervalID]$ is loaded only once from the CPU to the GPU but is reused across multiple queries. This approach reduces redundant data transfers and improves overall performance. For example, as shown in Fig. 6, we first determine the $intervalID$ values $\{0, 1, 2\}$ for queries $\{0, 1, \cdots, 10\}$ via binary search. Next, we group the queries based on their corresponding $intervalID$, ensuring that queries with the same $intervalID$ are processed together. For instance, the queries $\{0, 5, 6, 8\}$ are grouped under $intervalID$ 0. In the two-layer binary search procedure, we initially apply binary search on $intervalID$ 0 by loading the sub-array of each block within $Interval[0]$ from CPU DRAM into the GPU L2 cache and then executing the computations for the queries $\{0, 5, 6, 8\}$ within the kernel. After completing this batch of queries, we sequentially load the data for $intervalID$ 1 and $intervalID$ 2 into the L2 cache and process the queries for each $intervalID$ accordingly.

5.2 DP-based Cache Hierarchy

With the interval-aware auxiliary index, each query touches only $O(\sqrt{n})$ *active slices* (contiguous subarrays within blocks). To reduce latency and batch throughput, we selectively retain the hottest slices in the L2 cache and fully exploit shared memory capacity to increase the cache hit rate and reduce off-chip memory traffic.

Dynamic Programming (DP)-based Selection Policy. The key component of the selection policy is the visiting frequency of each block when processing a batch of top-k queries. Formally, consider the top-k queries whose results fall within a specified range, denoted as $Interval[intervalID]$. For each block $B_j \in \mathbb{C}$ covered by these queries, we denote the sub-array of B_j that lies within $Interval[intervalID]$ as $B_j[intervalID]$. The visiting frequency of $B_j[intervalID]$, denoted as $F_j[intervalID]$, represents the number of queries that access block B_j. Given the limited size of the L2 cache (denoted as $Size$), we select a subset of blocks $B_j[intervalID]$ to load into the L2 cache with the goal of maximizing the total frequency $\mathbb{F} = \sum F_j[intervalID]$, subject to the constraint $\sum |B_j[intervalID]| \leq Size$. To achieve the optimal block selection policy, we adopt a dynamic programming (DP) strategy. Let $dp[i]$ be the maximum total frequency achievable using i units of L2 cache. For each block $B_j[intervalID]$, the state transition equation of DP is : $dp[i] = \max\{dp[i], dp[i - |B_j[intervalID]|] + F_j[intervalID]\}$. Note that, for each $dp[i]$, initially $dp[i] = dp[i - 1]$, and the value of $dp[Size]$ finally represents the maximum frequency. During this process, we also track the best selection strategy, which will allow us to determine the final optimal selection. With the DP-based selection policy, high-frequency slices are proactively placed in the L2 cache so that the majority of query accesses are served from on-chip storage rather than off-chip memory. This increases the L2 cache hit rate and lowers the pressure on global memory bandwidth.

Shared Memory Optimization. During block-centric two-layer binary search on the GPU, all CTAs process the same interval identified by `intervalID`, and each thread performs a binary search on one block B_j[intervalID] of the current query. Let S denote the shared-memory capacity per CTA. For its assigned block, each thread computes the required shared-memory offset and checks whether it fits within S. If it fits, the block is loaded into shared memory, and the binary search is performed there; otherwise, the search is conducted directly in global memory. Since each block has a size of $\sqrt{n}$ and the array is partitioned into $\sqrt{n}$ intervals, the expected length of B_j[intervalID] is 1. Thus, even under data imbalance, only a small fraction of blocks exceeds the shared-memory capacity, and the overall performance impact is negligible.

Remark. DP pinning raises on-chip hit rates and converts repeated HBM probes into shared-memory hits, cutting per-probe latency, reducing PCIe traffic in out-of-core mode (via range-grouped batching), and improving SM efficiency. In aggregate, this yields higher QPS without sacrificing occupancy.

6 Dynamic Index Updating Support

In this section, we discuss the dynamic index updating under insertions and deletions. Formally, let $\mathbb{D} = [1, \ldots, N]$ denote an unbounded data stream. At any point in time, the system maintains a valid snapshot $A[1 \ldots n]$, corresponding to a contiguous window $\mathbb{D}[L \ldots R]$ of the stream, where $n = R - L + 1$ and $A[i] = \mathbb{D}[L + i - 1]$. As the stream evolves, the window may advance to a new range $\mathbb{D}[L' \ldots R']$ with $L \leq L'$ and $R \leq R'$, producing an updated snapshot $A[1 \ldots n']$. Each range-top-k query (l, r, k) is issued over the current snapshot $A[]$.

To support range-top-k queries over dynamically updated snapshots, the pre-treatment array and interval-aware auxiliary index must be kept up to date. Since the index structure is block-centric, only the blocks affected by the incoming updates need to be incrementally modified. Formally, let $\mathbb{D}$ denote the streaming data, with $\mathbb{I}$ representing newly appended data and $\mathbb{U}$ representing data

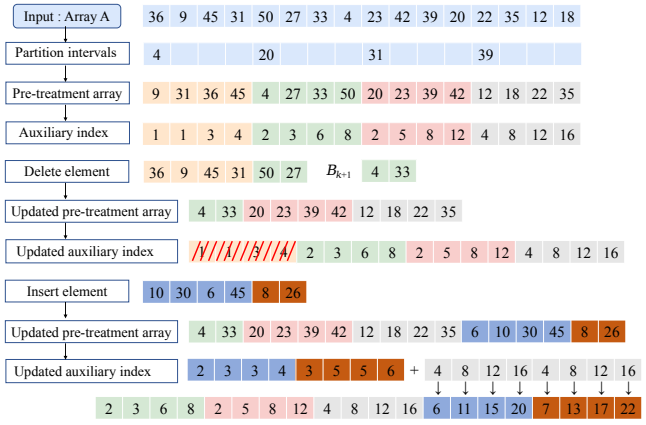


Figure 7: Index update for element insertion and deletion.

to be deleted. The data array changes from $A[] = \mathbb{D}[L \dots R]$ to $A[] = \mathbb{D}[L' \dots R']$, where $L' = L + |\mathbb{U}|$ and $R' = R + |\mathbb{I}|$.

To update the pre-treatment array $\mathbb{A}$, we proceed as follows. For the insertion set $\mathbb{I}$, we partition it into several blocks of length $\sqrt{n}$, sort each block, and denote the result as $\mathbb{I} = \{I_1, \dots, I_m\}$. These new blocks are appended to the end of the existing pre-treatment array. For the deletion set $\mathbb{U}$, we first identify and remove the blocks that are fully covered, i.e., $\mathbb{U} = \{B_1, \dots, B_k\}$. If the deletion partially overlaps with block B_{k+1}, we remove the corresponding elements from it and re-sort the result to form an updated block B'_{k+1}.

To update the auxiliary index $\mathbb{H}_{2D}$, we proceed as follows. For the insertion set $\mathbb{I}$, we compute the auxiliary index for each new block, denoted as $\{\mathbb{H}_{2D}[I_1], \cdots, \mathbb{H}_{2D}[I_m]\}$. and update the histogram using the last block of the current array: $\mathbb{H}_{2D}[I][J] = \mathbb{H}_{2D}[I][J] + \mathbb{H}_{2D}[B_n][J]$. The new auxiliary index entries are appended to the existing index. For the deletion set $\mathbb{U}$, we first remove the blocks that are fully covered. Since $\mathbb{H}_{2D}[I][J]$ represents an aggregated value related to the prefix sum of the histogram for Interval$[J]$ within block B_I, we subtract the prefix sum contribution of the deleted blocks: $\mathbb{H}_{2D}[I][J] = \mathbb{H}_{2D}[I][J] - \mathbb{H}_{2D}[k][J]$. If the deletion partially overlaps with block B_{k+1}, we remove the corresponding elements from blocks B_1 to B_{k+1} and reprocess the remaining elements to generate a new histogram for B_{k+1}. Denote the old and new auxiliary index of block B_{k+1} by $\mathbb{H}^{(\text{old})}_{2D}[k+1]$ and $\mathbb{H}^{(\text{new})}_{2D}[k+1]$, respectively. The updated index is then: $\mathbb{H}_{2D}[I][J] = \mathbb{H}_{2D}[I][J] - \mathbb{H}^{(\text{old})}_{2D}[k+1] + \mathbb{H}^{(\text{new})}_{2D}[k+1]$.

Fig. 7 summarizes the streaming update workflow for the toy example in Fig. 3. In the **deletion stage**, the deletion set covers all of block B_1 and part of block B_2. We therefore drop B_1 entirely and locally update B_2 by removing the deleted elements and re-sorting the remaining ones, yielding $B'_2 = \{33, 4\}$. The auxiliary index is updated by discarding the entries for the first k blocks and retaining those from B_{k+1} onward. In the **insertion stage**, the insertion set is partitioned into two blocks, $I_1 = \{10, 36, 6, 45\}$ and $I_2 = \{8, 26\}$, each sorted and appended to the pre-treatment array. We then compute their prefix sum under the original partition ranges, obtaining $\mathbb{H}_{2D}[I_1] = \{2, 3, 3, 4\}, \mathbb{H}_{2D}[I_2] = \{3, 5, 5, 6\}$, and offset these by the cumulative counts of the last existing block, $\mathbb{H}_{2D}[B_n] = \{4, 8, 12, 16\}$, to produce the updated auxiliary indices $\mathbb{H}_{2D}[I_1] = \{6, 11, 15, 20\}, \mathbb{H}_{2D}[I_2] = \{7, 13, 17, 22\}$.

Time Complexity Analysis. For insertion, the time complexity of adding new blocks to the pre-treatment array is $O(m)$, and updating the auxiliary index incurs a cost of $O(m \log n)$, where m is the number of inserted elements. For deletion, re-sorting takes $O(\sqrt{n})$ using radix sort on the GPU. The auxiliary index $\mathbb{H}_{2D}$, maintained as a prefix sum array, supports constant-time updates of $O(1)$ per block. In summary, regardless of the number of deleted elements m, the overall time complexity of data deletion remains $O(\sqrt{n})$.

Index Rebuilding Condition. We rebuild the block index only when interval skew is high, i.e., when $\max_i |\text{Interval}[i]| > \alpha \cdot \min_i |\text{Interval}[i]|$ for a user-defined $\alpha > 1$. Because rebuilding is performed entirely on the GPU, the overhead is modest (3.856s for $n = 2^{31}$), and in our experiments with $\alpha = 1.5$, the resulting range-top-k performance loss remains below 10%.

7 Evaluation

BCCE is implemented in approximately 5,000 lines of C++ and CUDA code, supporting in-memory, out-of-GPU-memory, and distributed execution modes.

7.1 Experimental Setup

Hardware. We use a machine with dual Intel Xeon Gold 6330 CPUs and two NVIDIA A100-PCIE-40GB GPUs [39]. The system operates on Debian 12 (Bookworm). All code was compiled using NVIDIA CUDA 11.8 [40] and GCC 12.2.0.

Datasets. We evaluate on synthetic and real-world datasets (Table 2). Following [58], we generate two synthetic unsigned-integer distributions: uniform over $[0, 2^{32}-1]$ and normal with a mean 2^{16} and a standard deviation 2^{16}. We vary top-k and the dataset size n from 2^{25}–2^{31} up to 2^{36} (256 GB). We also evaluate a real-world transaction dataset from a major online payment platform.

Baselines. Table 3 summarizes our baseline methods. Following the benchmark [58], we evaluate five GPU-side Top-k baselines, including three partial sorting approaches—BlockSelect [26] (via Faiss [36]), GridSelect [58] (via RAFT [45]), and Bitonic Top-K [48] (via Dr. Top-K [17]), as well as two partition-based methods, Air Top-K [58] (RAFT) and RadixSelect [18] (Dr. Top-K). To reflect CPU-side range indexes and order-statistics data structures, we include Wavelet Trees [16], a classic CPU range-query data structure that supports range top-k by maintaining a priority queue of candidate nodes and repeatedly expanding the most promising candidate to report the answers.

7.2 Full-in-GPU-Memory Mode

We benchmark all algorithms on fixed-size arrays A of length n, processing multiple randomly generated range queries (l, r, k). We evaluated performance for various values of n and k. Note that the reported execution time (latency) includes only the top-k query phase and excludes preprocessing. For each (n, k), all outputs are verified for correctness. Missing entries indicate configurations where resource limits or correctness issues prevent reporting: BlockSelect supports only $k \leq 2048$ due to its heavy use of shared memory and

Table 2: Statistics of Datasets.

Type	Dataset	Array Size (n)	Raw Size
Synthetic	Small to Median	2^{25} to 2^{31}	128MB to 8G
	Large	$2^{36} \approx 70$ billion	256GB
Real-World	Transactions	3,551,440,605	16GB

Table 3: State-of-the-art top-k engines.

Engine	Library	Category	Platform
BlockSelect[26]	Faiss [36]	Partial Sorting	GPU
GridSelect[58]	RAFT [45]	Partial Sorting	GPU
Bitonic Top-K [48]	Dr. Top-K [17]	Partial Sorting	GPU
RadixSelect [18]	Dr. Top-K [17]	Partition-based	GPU
Air Top-K [58]	RAFT [45]	Partition-based	GPU
Wavelet Trees[16]	SDSL [21]	Partial Sorting	CPU

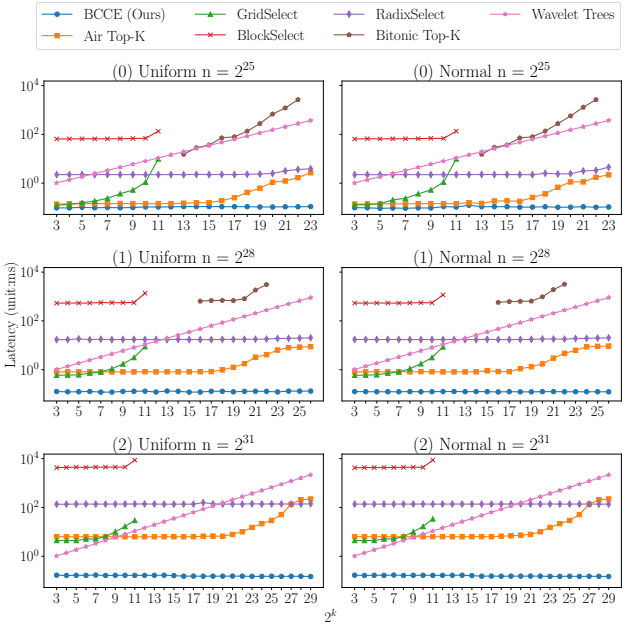


Figure 8: Latency comparison for different array sizes (n values) with uniform and normal distributions.

registers, and Bitonic Top-k results for small n are omitted because, although the program terminates, its outputs fail verification.

Each sub-figure in Fig. 8 plots latency versus k with fixed $n \in \{2^{25}, 2^{28}, 2^{31}\}$. BCCE consistently achieves sub-millisecond latency, while baselines slow down as k and n increase due to k-/ n-dependent costs. In contrast, BCCE remains nearly insensitive to k and varies only slightly with n (e.g., 0.1 ms at $n = 2^{25}$ vs. 0.16 ms at $n = 2^{31}$), since it runs in $O(\log n)$ time via two-layer binary search and a range-aware auxiliary index. In addition, in the run-time kernel, only $\sqrt{n}$ threads perform useful work. As a result, the available parallelism is insufficient to fully utilize all SMs on an NVIDIA A100 (108 SMs, up to 1024 threads per block) when $n < 2^{30}$. We also outperform the CPU wavelet-tree range top-k: its best-first traversal is inherently sequential (priority-queue driven), limiting parallelism. Overall, BCCE delivers 1.6–14,659.3× over Wavelet Trees, 1.3–178.4× speedup over GridSelect, 23.1–957.7× over RadixSelect, 1.4–1502.4× over Air Top-K, 138.9–13,964.1× over Bitonic Top-K, and 668.3–**52,433.1×** over BlockSelect.

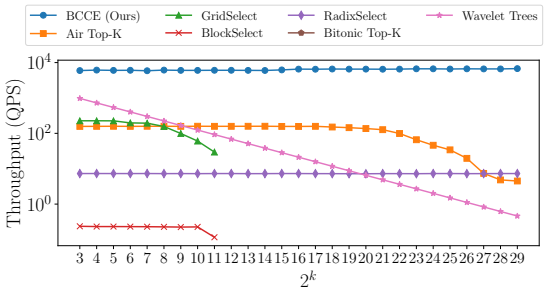


Figure 9: Throughput comparison.

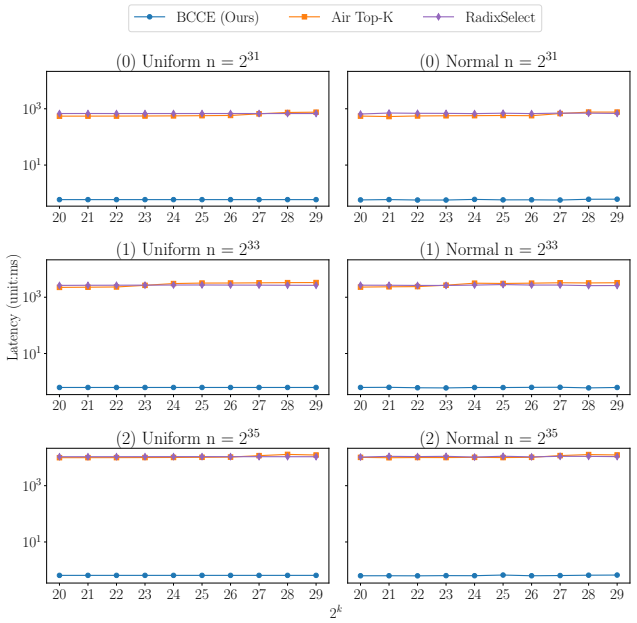


Figure 10: Latency comparison for different query sizes (n) under out-of-GPU-memory mode.

Fig. 9 shows the throughput performance when $n = 2^{31}$. BCCE achieves up to 6,757 top-k queries per second, whereas state-of-the-art methods achieve between 0.12 and 226 top-k queries per second as k varies from 2^3 to 2^{29}. This translates to a throughput improvement ranging from 29.8× to **56,308.3×**, which we attribute to our novel dynamic programming-based cache locality strategy.

7.3 Out-of-GPU-Memory Mode

When the array exceeds GPU memory, we keep it in CPU memory and transfer only the queried segments to the GPU. We evaluate BCCE under constrained GPU memory using a fixed array of length 2^{36} (≈256 GB) and three query sizes: 2^{31}, 2^{33}, and 2^{35}. Baselines transfer each queried segment independently, incurring heavy CPU–GPU I/O. In contrast, BCCE batches overlapping queries to reuse transfers, reducing the per-query transfer volume to $O(\sqrt{n})$. We compare against Air Top-K and RadixSelect, since other baselines exceed GPU memory limits in this setting. All reported latencies include CPU–GPU I/O time.

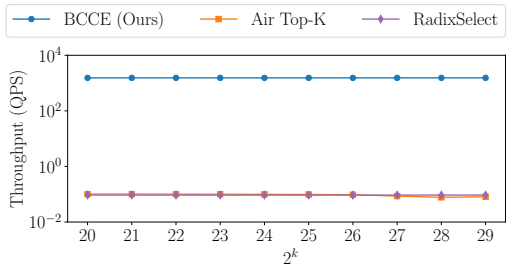


Figure 11: Throughput comparison (out-of-GPU-memory).

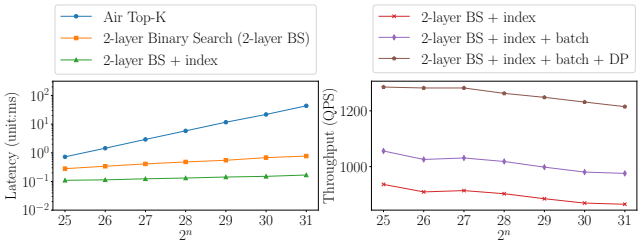


Figure 12: Performance breakdown for various optimizations, including the two-layer binary search with the range-aware auxiliary index, the dynamic programming (DP)-based cache locality strategy, and the batch optimization.

Fig. 10 shows that BCCE sustains sub-millisecond and near-constant latency with k; the latency increases slightly with n, primarily due to the growing host–device I/O cost, which scales sub-linearly as $O(\sqrt{n})$ in our design. In contrast, Air Top-K and RadixSelect quickly become I/O-bound as n increases (up to 95% of total time). At $n = 2^{33}$ with 10^6 queries, BCCE amortizes I/O to roughly one transfer per six queries (about 362 KB each), whereas each baseline query triggers a separate 32 GB transfer. Consequently, the gap widens with n, yielding speedups of 904.6×–16,444.9× over RadixSelect and 1008.8×–18,904.6× over Air Top-K.

Furthermore, Fig. 11 presents the throughput results for $n = 2^{35}$. BCCE handles up to 1,883 top-k queries per second, while baseline methods achieve only 0.08 to 0.11 queries per second as k varies from 2^{20} to 2^{29}. This corresponds to a throughput improvement of 17,118× to 23,537×, driven by our batch optimization strategy.

7.4 Efficiency of Different Optimizations

Here, we demonstrate the efficiency of the various optimizations implemented in BCCE in Fig. 12.

Efficiency of Two-Layer Binary Search. Benefiting from block-level pre-sorting (offloading work from runtime) and a two-layer binary search that avoids per-iteration global-memory accesses, our method achieves lower latency than AIR-TopK. As shown in the left panel of Fig. 12, it attains a 2.5×–56.7× speedup over AIR-TopK.

Efficiency of Auxiliary Index. As shown in the left panel of Fig. 12, Interval-aware auxiliary index delivers 6.6× and 260.1× speedups at $n = 2^{25}$ and $n = 2^{31}$, respectively, consistent with our analysis that the gain scales with n and demonstrating strong effectiveness on large datasets.

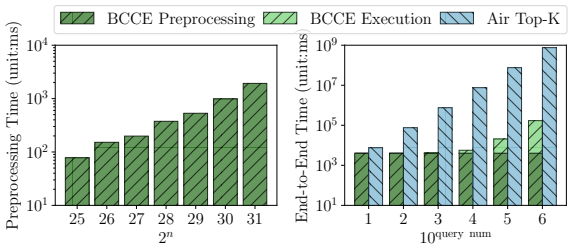


Figure 13: End-to-end comparison.

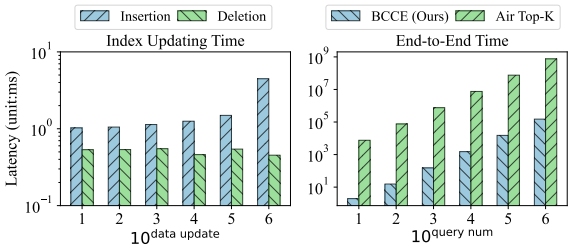


Figure 14: Evaluation on streaming data.

Efficiency of Batch Optimization. The right panel of Fig. 12 compares BCCE with and without batch optimization under an out-of-GPU-memory setting. Batch optimization improves throughput by 21.7% to 24.6% across all dataset sizes. This gain comes from reducing data transfers: without batching, each query transfers $\sqrt{n}$ elements to the GPU, while with batching, one transfer serves multiple queries sharing the same *rangeID* (see Section 5.1).

Efficiency of DP-Based Cache Locality. We compared BCCE with and without the DP-based cache locality optimization. This optimization reduces latency by 10% and improves throughput by 12%, primarily by increasing the cache hit rate during the binary search phase. Fig. 15 also shows that the L2 cache hit rate improved by 20% on average due to DP-Based Cache Locality.

7.5 Preprocessing and End-to-end Comparison

To assess the end-to-end efficiency of BCCE, we compare both preprocessing and top-k query execution times, using Air Top-K as the baseline. We measure preprocessing cost across input sizes n and evaluate scalability by varying the query count at $n = 2^{31}$ in out-of-GPU-memory settings. Fig. 13 reports the preprocessing time and overall execution time for different dataset sizes and query volumes.

Preprocessing. As detailed in Section 4, preprocessing in BCCE consists of three primary steps: (1) constructing the pre-treatment array, (2) computing the data distribution, and (3) building auxiliary index. Each step is implemented using dedicated CUDA kernels with time complexities of $O(n)$, $O(n)$, and $O(n \log n)$. Even for billion-scale datasets (e.g., $n = 2^{31}$), the entire preprocessing phase completes in just 3.856 seconds, demonstrating near-linear scalability with respect to n. Note that all the indexes have the same asymptotic size as the underlying array, i.e., they require $O(n)$ additional space for n elements, so their memory footprint remains comparable to that of the data itself, as illustrated in Fig. 1.

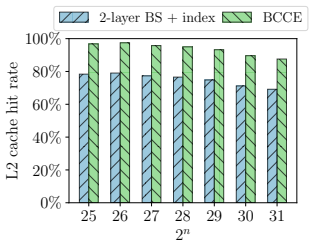


Figure 15: L2 Cache Hit Rate.

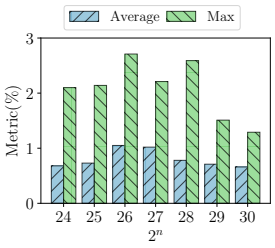


Figure 16: SM workload.

End-to-End Comparison. To ensure a fair comparison with Air Top-K, we include preprocessing time in end-to-end evaluation. As shown in Fig. 13, BCCE consistently outperforms Air Top-K across all query volumes due to Air Top-K's higher time complexity and BCCE 's efficient preprocessing and top-k execution. As query volume grows, preprocessing cost becomes negligible; for example, it accounts for only 2.4% of the total time at 10^6 queries. Overall, BCCE achieves 1.89×-4424.6× end-to-end speedup.

7.6 Index Updating for Streaming Data

We evaluate the performance of BCCE on data updates in a streaming setting. Specifically, we perform insertions and deletions of varying sizes on a dataset with $n = 2^{31}$, and measure both the index update time and the end-to-end query execution time in Fig. 14 with Air Top-K as a baseline. For the `insert` operation, Fig. 14 shows that the insertion time remains nearly constant at about 1.3 ms for $m < 10^5$, since only the portion corresponding to the newly added data is updated, and small m underutilizes the GPU. When $m > 10^6$, the insertion time increases as the GPU becomes fully utilized and the $O(m \log n)$ complexity dominates; for example, $m = 10^6$ is 4.4 ms slower than $m = 10^5$. For the deletion operation, we observe that the index update time remains nearly constant regardless of the number of deleted elements. As discussed in Section 6, this is due to the upper bound of $O(\sqrt{n})$ time complexity, which is independent of the deletion size m. For example, whether deleting 10 elements or 10^6, the update time consistently remains around 0.55 ms.

We also compare end-to-end performance accounting for both top-k query execution and index updates. Although Air Top-K incurs no index maintenance cost, BCCE still outperforms it. Specifically, when processing 1 million data updates (inserting and deleting 0.5 million elements each), BCCE achieves a speedup of 364× ~5010× over Air Top-K across workloads ranging from 10 to 10^6 queries. Speedup increases with more queries, as index updates are amortized across queries.

7.7 Workload Balance

To evaluate workload balance, we record per-SM execution times and quantify load imbalance as $\frac{\text{Max}-\text{Min}}{\text{Min}}$. Fig. 16 reports the average and maximum imbalance over 10^5 queries with $k = 2^{13}$ across different n, and the consistently low values indicate near-uniform SM workloads. Because all threads follow the same two-level binary search, any imbalance can only stem from the second level; partitioning the array into $\sqrt{n}$ blocks and $\sqrt{n}$ intervals yields *rangeID*s that receive ≈ 1 element per block in expectation, leading to uniform per-thread work across warps/blocks. We further profile occupancy with NVIDIA Nsight Compute: Table 4 shows achieved occupancy close to the theoretical maximum, increasing resident warps and improving latency hiding. Together, the low imbalance and high occupancy demonstrate that BCCE effectively exploits SIMT execution.

Table 4: Workload balance testing.

n	Theoretical Occupancy(%)	Achieved Occupancy(%)
2^{25}	50.00	49.96
2^{28}	50.00	49.96
2^{31}	50.00	49.98

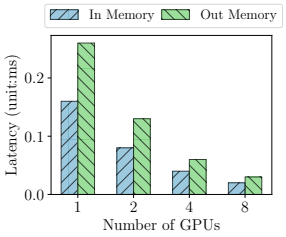


Figure 17: Multi-GPU test.

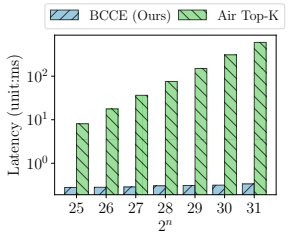


Figure 18: Case study.

7.8 Scalability

To evaluate the scalability of BCCE, we conducted a performance analysis across multiple GPUs, focusing on range query workload partitioning. Given its minimal memory requirements (up to $\sqrt{n}$) and the batch optimization, there is no necessity to partition the input array. Instead, various GPUs are utilized to manage distinct groups of queries that share the same *rangeID*, demonstrating significant scalability. Fig. 17 illustrates the scalability of both the in-memory and out-of-GPU-memory versions of BCCE. Notably, BCCE achieves up to 7.4× and 7.1× speedups in both configurations.

7.9 Case Study: Real-World Workload

To demonstrate BCCE under real-world workloads, we compare it with Air Top-K on the production risk-control dataset (Section 1), which contains billions of daily electronic payment transactions represented as (`uid`, `transaction_time`, `transaction_amount`). In risk-control pipelines, transactions of target users are filtered by time window, and the top-k largest amounts are retrieved to derive user-level risk features. These features, together with labels from historical data, are fed into GPU-trained models (e.g., decision trees, random forests, XGBoost). We extracted several days of transaction records for a target user group, yielding up to 2^{32} records and randomly sampled time intervals and issued top-k range queries (k=1000) to return the highest-amount transactions (Fig. 18). To reflect end-to-end deployment, we include CPU–GPU I/O in all measurements. BCCE consistently outperforms Air Top-K, and the gap widens with larger n; the trends match those on synthetic data. Across all cases, BCCE answers each query in 0.31 ms, delivering 29×–1783× speedups as n increases from 2^{25} to 2^{31}.

8 Related Work

Range-Query Data Structures. Existing range-query indexes such as LSM trees [12, 34] and streaming B$^+$-trees [47] optimize *keyed* lookups or joins on CPUs, but neither computes exact order statistics nor maps well to SIMT due to pointer chasing and irregular control flow. In contrast, range-top-k must locate the k-th order statistic within $[l, r]$ and materialize the top-k, requiring rank-thresholding and verification with cache-friendly access. This mismatch motivates our GPU–memory-hierarchy co-design.

Sliding-Window Based Top-k. Range-top-k workloads are often approximated by sliding-window top-k methods [37], which maintain a *single* top-k for a *fixed-length* window that moves sequentially over a stream. In contrast, online range-top-k queries ask for top-k over *many arbitrary* subranges of the same snapshot, violating the single window assumption and requiring range-aware, non-sequential access and recomputation across non-adjacent intervals.

9 Conclusion

We presented BCCE, a block-centric, GPU–co-designed engine for range-top-k queries. By combining a two-layer binary search with an interval-aware auxiliary index, BCCE reduces each query's working set and enables fully coalesced, SIMT-regular execution. Extensive evaluations show that BCCE achieves sub-millisecond latency and delivers up to four orders of magnitude speedup over state-of-the-art GPU baselines while maintaining high throughput under dynamic updates.

Acknowledgments

We sincerely thank the reviewers for their insightful comments. This work is funded in part by the National Natural Science Foundation of China (NO. 62502193), the Key Program of Natural Science Foundation of Jiangsu under grant (NO. BK20251190, BK20243053), fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM901), Nanjing "U35" Talent Cultivation Program (No. U (2024) 001), National Key Research and Development Plan of China (2023YFB4502305), Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University. Zhengyi Yang, Yongchao Liu, and Shaonan Ma are the corresponding authors of this paper.

References

[1] Mohammad Reza Abbasifard, Bijan Ghahremani, and Hassan Naderi. 2014. A survey on nearest neighbor search methods. *International Journal of Computer Applications* 95, 25 (2014).

[2] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. 2012. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)* 17 (2012), 4–1.

[3] Hannah Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. 2006. IO-Top-k: Index-access Optimized Top-k Query Processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 475–486. http://dl.acm.org/citation.cfm?id=1164169

[4] Nathan Bell and Jared Hoberock. 2012. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*. Elsevier, 359–371.

[5] Gianfranco Bilardi and Alexandru Nicolau. 1989. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.* 18, 2 (1989), 216–228.

[6] Minmin Chen, Alex Beutel, Paul Covington, Sagar Jain, Francois Belletti, and Ed H Chi. 2019. Top-k off-policy correction for a REINFORCE recommender system. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. 456–464.

[7] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems* 34 (2021), 5199–5212.

[8] Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, et al. 2024. Magicpig: Lsh sampling for efficient llm generation. *arXiv preprint arXiv:2410.16179* (2024).

[9] Paolo Ciaccia and Davide Martinenghi. 2024. Directional Queries: Making Top-k Queries More Effective in Discovering Relevant Results. *Proc. ACM Manag. Data* 2, 6 (2024), 232:1–232:26. https://doi.org/10.1145/3698807

[10] J Shane Culpepper, Matthias Petri, and Falk Scholer. 2012. Efficient in-memory top-k document retrieval. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. 225–234.

[11] Ali Dashti, Ivan Komarov, and Roshan M D'Souza. 2013. Efficient computation of k-nearest neighbour graphs for large high-dimensional data sets on GPU clusters. *Plos one* 8, 9 (2013), e74113.

[12] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[13] Harish Doraiswamy, Huy T Vo, Cláudio T Silva, and Juliana Freire. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *2016 IEEE 32nd International conference on data engineering (ICDE)*. IEEE, 1086–1097.

[14] Wenqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. 2024. A survey on rag meeting llms: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 6491–6501.

[15] Yuan Feng, Seonjin Na, Hyesoon Kim, and Hyeran Jeon. 2025. Barre Chord: Efficient Virtual Memory Translation for Multi-Chip-Module GPUs. In *Proceedings of the 51st Annual International Symposium on Computer Architecture* (Buenos Aires, Argentina) *(ISCA '24)*. IEEE Press, 834–847. https://doi.org/10.1109/ISCA59077.2024.00065

[16] Travis Gagie, Simon J Puglisi, and Andrew Turpin. 2009. Range quantile queries: Another virtue of wavelet trees. In *International Symposium on String Processing and Information Retrieval*. Springer, 1–6.

[17] Anil Gaihre. 2022. Anil-Gaihre/DrTopKSC. https://github.com/Anil-Gaihre/DrTopKSC.

[18] Anil Gaihre, Da Zheng, Scott Weitze, Lingda Li, Shuaiwen Leon Song, Caiwen Ding, Xiaoye S Li, and Hang Liu. 2021. Dr. Top-k: delegate-centric Top-k on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[19] Tianao Ge, Xiaowen Chu, and Hongyuan Liu. 2025. Interleaved Bitstream Execution for Multi-Pattern Regex Matching on GPUs. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 385–400. https://doi.org/10.1145/3725843.3756052

[20] Minseong Gil, Dongho Ha, Simla Burcu Harma, Myung Kuk Yoon, Babak Falsafi, Won Woo Ro, and Yunho Oh. 2025. Avant-Garde: Empowering GPUs with Scaled Numeric Formats. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 153–165. https://doi.org/10.1145/3695053.3731100

[21] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*. 326–337.

[22] David E Graff, Eugene I Shakhnovich, and Connor W Coley. 2021. Accelerating high-throughput virtual screening through molecular pool-based active learning. *Chemical science* 12, 22 (2021), 7866–7881.

[23] Rodrigo Huerta, Mojtaba Abaie Shoushtary, José-Lorenzo Cruz, and Antonio Gonzalez. 2025. Dissecting and Modeling the Architecture of Modern GPU Cores. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 369–384. https://doi.org/10.1145/3725843.3756041

[24] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.* 40, 4 (2008), 11:1–11:58. https://doi.org/10.1145/1391729.1391730

[25] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems* 32 (2019).

[26] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Trans. Big Data* 7, 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[27] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and interactive cube exploration. In *2014 IEEE 30th international conference on data engineering*. IEEE, 472–483.

[28] Peter Kipfer and Rüdiger Westermann. 2005. Improved GPU sorting. *GPU gems* 2 (2005), 733–746.

[29] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[30] Yiwei Li, Yuxin Jin, Boyu Tian, Huanchen Zhang, and Mingyu Gao. 2025. ANSMET: Approximate Nearest Neighbor Search with Near-Memory Processing and Hybrid Early Termination. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1093–1107. https://doi.org/10.1145/3695053.3731013

[31] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[32] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. 2024. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *arXiv preprint arXiv:2409.10516* (2024).

[33] Fanzhen Liu, Zhao Li, Baokun Wang, Jia Wu, Jian Yang, Jiaming Huang, Yiqing Zhang, Weiqiang Wang, Shan Xue, Surya Nepal, and Quan Z. Sheng. 2022. eRiskCom: an e-commerce risky community detection platform. *The VLDB Journal* 31, 5 (2022), 1085–1101.

[34] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[35] Nicolás Madrid and Pavel Rusnok. 2019. A Top-K Retrieval algorithm based on a decomposition of ranking functions. *Information Sciences* 474 (2019), 136–153.

[36] Inc. Meta Platforms. 2022. Faiss v1.7.3. Retrieved March 1, 2023 from https://github.com/facebookresearch/faiss.

[37] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. 2006. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 635–646.

[38] Gonzalo Navarro and Yakov Nekrich. 2012. Top-k document retrieval in optimal time and linear space. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 1066–1077.

[39] Nvidia. 2020. NVIDIA A100 Tensor Core Gpu. https://www.nvidia.com/en-us/data-center/a100/

[40] Nvidia. 2022. CUDA 11.8 Release Notes. https://docs.nvidia.com/cuda/archive/11.8.0/cuda-toolkit-release-notes/index.html

[41] HweeHwa Pang, Xuhua Ding, and Baihua Zheng. 2010. Efficient processing of exact top-k queries over disk-resident sorted lists. *The VLDB Journal* 19 (2010), 437–456.

[42] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating Retrieval-Augmented Generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 15–32. https://doi.org/10.1145/3669940.3707264

[43] Derrick Quinn, E. Ezgi Yücel, Martin Prammer, Zhenxing Fan, Kevin Skadron, Jignesh M. Patel, José F. Martínez, and Mohammad Alian. 2025. DReX: Accurate and Scalable Dense Retrieval Acceleration via Algorithmic-Hardware Codesign. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1108–1124. https://doi.org/10.1145/3695053.3731079

[44] Saladi Rahul and Yufei Tao. 2019. A Guide to Designing Top-k Indexes. *SIGMOD Rec.* 48, 2 (2019), 6–17. https://doi.org/10.1145/3377330.3377332

[45] Rapidsai. 2022. Rapidsai/raft: RAFT contains fundamental widely-used algorithms and primitives for data science, Graph and machine learning. https://github.com/rapidsai/raft

[46] Tobias Ribizel and Hartwig Anzt. 2020. Parallel selection on GPUs. *Parallel Comput.* 91 (2020), 102588.

[47] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2020. Parallel index-based stream join on a multicore cpu. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2523–2537.

[48] Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient Top-K Query Processing on Massively Parallel Hardware. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1557–1570. https://doi.org/10.1145/3183713.3183735

[49] Erik Sintorn and Ulf Assarsson. 2008. Fast parallel GPU-sorting using a hybrid algorithm. *J. Parallel and Distrib. Comput.* 68, 10 (2008), 1381–1388.

[50] Elias Stehle and Hans-Arno Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on gpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 417–432.

[51] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2024. Scalable Billion-point Approximate Nearest Neighbor Search Using {SmartSSDs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1135–1150.

[52] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–27.

[53] Zijie J Wang and Duen Horng Chau. 2024. MeMemo: On-device Retrieval Augmentation for Private and Personalized Text Generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2765–2770.

[54] Feng Xue, Xiangnan He, Xiang Wang, Jiandong Xu, Kai Liu, and Richang Hong. 2019. Deep item-based collaborative filtering for top-n recommendation. *ACM Transactions on Information Systems (TOIS)* 37, 3 (2019), 1–25.

[55] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 157–168.

[56] Hamed Zamani, Susan T. Dumais, Nick Craswell, Paul N. Bennett, and Gord Lueck. 2020. Generating Clarifying Questions for Information Retrieval. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 418–428. https://doi.org/10.1145/3366423.3380126

[57] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.

[58] Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. 2023. Parallel top-k algorithms on gpu: A comprehensive study and new methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[59] Niansong Zhang, Wenbo Zhu, Courtney Golden, Dan Ilan, Hongzheng Chen, Christopher Batten, and Zhiru Zhang. 2025. Characterizing and Optimizing Realistic Workloads on a Commercial Compute-in-SRAM Device. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 1011–1025. https://doi.org/10.1145/3725843.3756132

[60] Vasileios Zois, Vassilis J Tsotras, and Walid A Najjar. 2019. Efficient main-memory top-k selection for multicore architectures. *Proceedings of the VLDB Endowment* 13, 12 (2019).

A Appendix

A.1 Artifact Description

In-memory Version. For the in-memory implementation, we employ `cub::DeviceMergeSort` to sort the partial blocks efficiently. A dedicated CUDA kernel is then used to construct the auxiliary index for each partial block. Within this kernel, each thread processes a specific element from the `ranges` array and utilizes binary search to efficiently map the given range to its corresponding entry in the auxiliary structure. To minimize data I/O, rather than transferring the sorted partial blocks to the host via `cudaMemcpy`, we determine the `rangeID` directly on the device using binary search. When computing the top-k elements, each thread performs a binary search within a block, leveraging the auxiliary structure to obtain both the search position and the block size. The binary search results are then aggregated using a reduction operation to compute the total number of elements, which is subsequently compared against K.

Out-of-GPU-memory Version. For out-of-GPU-memory scenario, we adopt a similar approach to obtain the sorted partial blocks and their corresponding auxiliary structures. However, since the auxiliary structure cannot entirely reside in device memory, we perform the binary search on the host to determine the `rangeID`. When computing the top-k elements, only the array and auxiliary index corresponding to the determined `rangeID` are loaded into device memory, thereby effectively reducing I/O overhead.

Distributed BCCE. Traditional distributed top-k algorithms partition the input array into disjoint subarrays, each assigned to a separate GPU to compute local top-k results, which are then merged via inter-GPU communication to obtain the final ranking. In contrast, BCCE leverages its intrinsic space complexity of $O(\sqrt{n})$, enabling it to process large arrays on a single GPU by storing only the elements associated with a specific *rangeID*. This design eliminates the need for array partitioning and inter-GPU communication, thereby reducing both computational and synchronization overhead. Instead, we adopt a **rangeID-based workload distribution strategy**, in which queries sharing the same rangeID are assigned to the same GPU. This ensures that each GPU handles a self-contained portion of the workload, avoiding cross-device data exchange and enabling efficient parallel execution.

I/O and Computation Pipeline. In out-of-GPU-memory settings, batch optimization enables efficient pipelining between data transfer and computation. While the two-layer binary search is executed on data from Range[rangeID], the PCIe interface concurrently transfers data for Range[rangeID + 1] in preparation for the next iteration. Since each transfer involves only $O(\sqrt{n})$ elements, the GPU HBM can accommodate data for two ranges simultaneously. This pipelined execution significantly reduces I/O overhead and improves overall throughput.

Grid Synchronization. In the *Top-k range query search* phase, each thread processes a block by determining its corresponding range position and size. A critical step in this process is performing a reduction operation within a `while` loop inside the binary search. After each reduction, all threads must obtain the aggregated result before proceeding to the next iteration. However, conventional CUDA kernels do not support global synchronization (`grid.sync()`), preventing all thread blocks from synchronizing within a single kernel execution. A common workaround is to break the computation into multiple kernel launches, enforcing synchronization across executions. However, this introduces significant overhead due to frequent kernel restarts and explicit global memory exchanges. To eliminate these inefficiencies, we leverage `cudaLaunchCooperativeKernel`, which enables inter-block synchronization at runtime via `grid.sync()`. This ensures that all threads obtain the reduction results within the same kernel execution before proceeding to the next iteration, avoiding the overhead of multiple kernel launches and excessive global memory accesses.

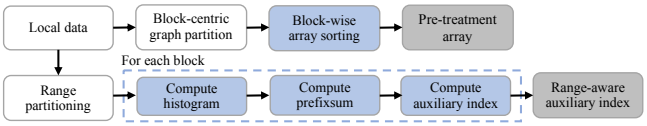


Figure 19: Workflow of preprocessing.

Range-Aware Auxiliary Index Transposition. During preprocessing, each thread determines the range position and size of its assigned block by accessing the *range-aware auxiliary index* stored in global memory. If the index is structured in a block-centric manner—where each row corresponds to the auxiliary index of a specific block—adjacent threads may access memory locations that are far apart. Since CUDA achieves optimal performance when consecutive threads access adjacent memory locations (coalesced memory access), this layout results in inefficient memory transactions and increased latency. To optimize memory access patterns, we transpose the *range-aware auxiliary index*. Instead of organizing rows by block ID, we restructure them so that each row corresponds to a single rangeID. In this layout, auxiliary index values for consecutive block IDs within the same range are stored contiguously in memory. Since threads within a warp typically process consecutive block IDs sharing the same rangeID, this transformation enhances memory coalescing, reduces global memory latency, and improves overall performance.

Preprocessing Optimization. Given an input array, the preprocessing phase consists of two main steps: *generating the pre-treatment array* $\mathbb{A}$ and *constructing the auxiliary index* $\mathbb{H}_{2D}$. Instead of a sequential CPU-based approach, we accelerate preprocessing using GPU parallelization.

In the *pre-treatment array generation* step, we apply parallel processing based on a block-centric graph partitioning scheme. To further speed up intra-block sorting, we leverage GPU parallelism for efficient sorting within each block. Specifically, within the kernel, we assign one CTA (Cooperative Thread Array) per block and utilize `cub::BlockRadixSort` for intra-block sorting.

For the *range-aware auxiliary index construction*, we begin with a full scan of the input array to collect global statistics and determine the range partitions. Subsequently, for each block, we compute a local auxiliary index through three sub-steps: histogram computation, prefix sum calculation, and auxiliary index derivation using Equation 1. To accelerate these steps, we leverage `cub::DeviceHistogram` for histogram computation, `cub::DeviceScan` for prefix sum operations, and a custom CUDA kernel for vector-wise addition to derive the final auxiliary index. Once the auxiliary indices are generated, we apply a matrix transposition kernel to reorganize the

index layout. These GPU-accelerated optimizations significantly reduce preprocessing latency.

A.2 Artifact Evaluation

We evaluate BCCE on an NVIDIA A100 GPU using NVIDIA CUDA 11.8. Our computational artifact is available as an anonymized repository at https://anonymous.4open.science/r/BCCE. The repository includes build instructions and scripts to reproduce the key experimental results.

To reproduce the results in our paper:

(1) Set up the Python environment for figure generation: Go to directory “script”, and run “bash setup.sh”.
(2) Generate datasets: Compile "generate_dataset.cpp" and run the dataset generator with "./generate_dataset <query_size_power> <k_size_power> <num_queries>". <query_size_power>: Specifies the size of each query as ($2^{\texttt{query\_size\_power}}$). <k_size_power>: Specifies the size of k as ($2^{\texttt{query\_size\_power}}$). <num_queries>: The number of queries to generate.
(3) Generate the binary: In the directory "src/topk", run “make”. A binary file named “topk” is generated.
(4) Run the Program: Run the resulting executable with "./BCCE <data_directory> <verify_correctness>"<data_directory>: Path to the directory containing the generated dataset files. <verify_correctness>: A boolean flag (true or false) indicating whether to verify the correctness of the results.