Parallel Top-K Algorithms on GPU: A Comprehensive Study and New Methods

Jingrong Zhang
NVIDIA
Shanghai, China
christinaz@nvidia.com

Akira Naruse
NVIDIA
Tokyo, Japan
anaruse@nvidia.com

Xipeng Li
NVIDIA
Beijing, China
xipengl@nvidia.com

Yong Wang*
NVIDIA
Shanghai, China
yongw@nvidia.com

ABSTRACT

The top-K problem is an essential part of many important applications in scientific computing, information retrieval, etc. As data volume grows rapidly, high-performance parallel top-K algorithms become critical. We propose two parallel top-K algorithms, AIR Top-K (Adaptive and Iteration-fused Radix Top-K) and GridSelect, for GPU. AIR Top-K employs an iteration-fused design to minimize CPU-GPU communication and device data access. Its adaptive strategy eliminates unnecessary device memory traffic automatically under various data distributions. GridSelect can process data on-the-fly. It adopts a shared queue and parallel two-step insertion to decrease the frequency of costly operations. We comprehensively compare 8 open-source GPU implementations and our methods for a wide range of problem sizes and data distributions. For batch sizes 1 and 100, respectively, AIR Top-K shows 1.98–21.48× and 8.01–574.78× speedup over previous radix top-K algorithm, and 1.44–7.34× and 1.38–31.91× speedup over state-of-the-art methods. GridSelect shows up to 882.29× speedup over its baseline.

KEYWORDS

Top-K, K-selection, Radix select, Parallel algorithms, GPU, CUDA

ACM Reference Format:
Jingrong Zhang, Akira Naruse, Xipeng Li, and Yong Wang. 2023. Parallel Top-K Algorithms on GPU: A Comprehensive Study and New Methods. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607062

1 INTRODUCTION

The top-K problem is about finding the smallest (or largest) K elements in a list. It is a fundamental task applied in numerous domains, like information retrieval [37], recommender system [7, 36], data services [19], and scientific computing [13]. As data volume and model size are growing rapidly nowadays, Graphics Processing Units (GPUs), which are massively parallel devices [8], are widely deployed to accelerate applications in these areas. The parallel top-K algorithm on GPU becomes essential to these applications and has attracted increasing research interests recently [12, 16, 31, 32].

Meanwhile, the scale of the top-K problem varies greatly. For large-scale training of deep learning model, top-K is used in Deep Gradient Compression [18] to select top 0.1% entries from millions or billions of gradient values in order to reduce communication cost. For drug discovery, high-throughput virtual screening identifies the top 50000 ligands from 10^8 molecules [13]. The K-nearest neighbors classifier and approximate nearest neighbor search [4, 5, 10, 16, 33, 34] require to find tens of best neighbors. Such a wide range of problem sizes challenges the efficiency of a general top-K algorithm.

Four categories of parallel top-K algorithms have been developed: sorting, partial sorting, partition-based methods, and hybrid methods [12, 35]. The most straightforward one is sorting, which parallelly sorts all elements [6, 27] and then extracts the first K items. With high-quality GPU implementation of parallel radix sort [20], this way can be efficient. However, sorting the full list is still time-intensive and unnecessary.

Partial sorting methods avoid sorting the whole list and are only required to identify and sort the best K elements. Heap is the typical data structure used for this purpose in a sequential algorithm, however, heap operations are difficult to parallelize. Thus, WarpSelect [16] is proposed for GPU. It is based on the bitonic sort algorithm, which is fully parallel. It also has the unique advantage of being able to process elements on-the-fly. Bitonic Top-K [32] is another partial sorting method based on bitonic sort.

Partition-based methods recursively distribute candidates into buckets by value and keep track of the number of elements in each bucket until all the top-K results are found. Typical algorithms are QuickSelect [9, 17], BucketSelect [1], SampleSelect [31] and RadixSelect [1]. Their main difference is the strategy of choosing pivots for dividing candidates. For example, RadixSelect chooses multiple pivots based on digits, while QuickSelect adopts one pivot. Partition-based approaches can dramatically reduce workload in each iteration, and multiple pivots are desirable for parallelization on GPU.

Because there are numerous algorithms and the problem sizes vary to a large extent, an immediate question is which algorithm is the most efficient one for a particular problem size. To the best of

*Corresponding author.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607062

our knowledge, no comprehensive comparison has been done. In this study, we try to answer this question via a detailed benchmark of state-of-the-art algorithms, covering a wide range of input sizes and K. Synthetic datasets of different distributions and real-world datasets are used to compare the performance of various algorithms thoroughly.

Besides a comprehensive benchmark, we also propose two parallel top-K algorithms, namely AIR Top-K (Adaptive and Iteration-fused Radix Top-K) and GridSelect, to improve performance and overcome the shortcomings of previous algorithms.

Both AIR Top-K and RadixSelect are parallel radix top-K algorithms. RadixSelect for GPU was proposed a decade ago [1] and several researchers have optimized it [12, 32]. Nevertheless, all of these parallel methods require non-trivial work from the CPU side, necessitating lots of data communication between the CPU and GPU. In contrast, CPU is only needed to launch AIR Top-K kernels to GPU. Except for that, all the computation of AIR Top-K runs on GPU due to the proposed iteration-fused design, which has several advantages. First, it minimizes the data transfer between CPU and GPU through PCIe, which has much higher latency and lower bandwidth than GPU memory. Second, it reduces the number of kernel launches considerably. Third, it reduces device data access, which is important for memory-bound applications. Fourth, it makes top-K computation asynchronous with respect to the CPU, that is, CPU can launch all kernels at once rather than launching one kernel and waiting for some intermediate data before launching a second kernel.

One shortcoming of radix top-K is that its performance varies with different data distributions [12]. To achieve good performance regardless of the data distribution, we design an adaptive strategy in AIR Top-K that alters automatically between two buffering behaviors in each iteration: one is suitable for evenly distributed data and the other works well for narrowly distributed data. This strategy can minimize memory traffic and reduce the footprint of device memory.

AIR Top-K can not process data on-the-fly. So we also propose GridSelect based on WarpSelect [16], which has the unique advantage of on-the-fly processing. WarpSelect works in the unit of a warp, which contains 32 threads. Each thread maintains a thread queue in GPU registers, to which new qualified elements are inserted. When any of the 32 thread queues are full, bitonic sorting and merging operations are used to update top-K results with elements in all thread queues. These operations are expensive but are used whenever any thread queue is full. In GridSelect, we replace the 32 thread queues with a shared queue and use parallel two-step insertion to add elements to this queue. In this way, insertion is still parallel, register footprint is reduced, and expensive operations are called less often so performance is improved.

In summary, we make the following contributions in this study:

- We propose AIR Top-K, an iteration-fused top-K algorithm that runs fully on GPU except for kernel launch. It considerably reduces data transfer between CPU and GPU, kernel launch overhead, and device data access. We also introduce an early stopping strategy to avoid unnecessary work. Our algorithm shows 1.98–21.48× and 8.01–574.78× speedup over the previous radix top-K algorithm for batch size 1 and 100, respectively.
- To reduce the influence of data distribution, we design an adaptive strategy in AIR Top-K that automatically alters behavior according to data distribution. Compared with an algorithm without this adaptive strategy, it achieves up to 6.53× speedup under adversarial data distribution.
- We propose GridSelect based on WarpSelect. In GridSelect, we use a single shared queue with parallel two-step insertion to reduce costly operations and register pressure. Together with multi-block launch, GridSelect shows up to 882.29× speedup over BlockSelect, an extension of WarpSelect.
- By comparing 8 open-source GPU implementations with our new methods, we provide a comprehensive performance report of parallel top-K algorithms on both synthetic and real-world datasets, and demonstrate that our algorithms outperform state-of-the-art algorithms by a large margin in almost all cases. AIR Top-K shows 1.44–7.34× and 1.38–31.91× speedup over state-of-the-art algorithms for batch size 1 and 100, respectively.

The paper is organized as follows. Section 2 introduces the context, notation, and related work. Section 3 introduces the design of AIR Top-K. Section 4 shows the optimizations made in GridSelect. In Section 5, we provide a comprehensive benchmark to compare our methods with state-of-the-art algorithms and show the detailed performance of AIR Top-K and GridSelect.

2 BACKGROUND AND RELATED WORK

In this section, we first give the definition of the top-K problem. Then, we briefly survey parallel Top-K algorithms. Next, we explain radix top-K in detail as AIR Top-K is based on it. Finally, we introduce some basic concepts of GPU architecture used in this paper.

2.1 Problem Statement

The top-K problem asks to find the smallest (or largest) K elements in a list of N elements. Without loss of generality, we assume to find the smallest K elements in this paper.

Besides the top K values, we need their indices in the input list in many applications. For example, the search engine needs indices to find items having high ranking scores. A practical top-K algorithm should output the indices along with the values.

So, the inputs of a top-K algorithm are a list L of N elements and a number K, which is in the range of $[1, N]$. The outputs are two lists of length K, value list V and index list I, satisfying the following constraints: $L[I[i]] = V[i]$ and $V[i] \leq L[j], \forall i \in [1, K]$ and $\forall j \in [1, N]$ and $j \notin I$, where $L[i]$ denotes the ith element in list L.

2.2 Related Work

Notable partition-based parallel top-K algorithms are QuickSelect [9], BucketSelect [1], RadixSelect [1], and SampleSelect [31]. QuickSelect [9] is based on quick sort, but unlike sorting, QuickSelect iterates only on the sub-list containing the Kth element. The procedure repeats recursively till the chosen pivot eventually is the Kth elements, and the top-K elements are collected during the recursion. BucketSelect, SampleSelect, and RadixSelect distribute candidates into tens to hundreds of buckets using multiple pivots. The pivots of BucketSelect are decided by the minimum and the maximum of candidates. SampleSelect samples a small fraction of elements and sorts them to find more suitable pivots. RadixSelect, like the most significant-digit radix sort [14], divides elements into buckets based on their digits.

Compared with the other three partition-based methods, RadixSelect has several merits. First, its worst-case complexity is $O(N)$. Assuming that a r-bit number is split into b-bit digits in RadixSelect, $\lceil \frac{r}{b} \rceil$ iterations at most are required. So both the average- and worst-case complexity is $O(\lceil \frac{r}{b} \rceil N)$. In contrast, QuickSelect, in the worst case, can remove only one element per iteration. So N iterations of processing approximately N elements lead to $O(N^2)$ worst-case complexity. Second, with 8- or 11-bit digits, corresponding to 256 or 2048 buckets, RadixSelect is highly parallel and can reduce the workload more efficiently. Third, both BucketSelect and SampleSelect require calculating statistics of input data for choosing pivots, while the pivots of RadixSelect are independent of input data.

WarpSelect [16] and Bitonic Top-K [32] are partial sorting methods. By constructing and selecting ascending-descending sorted (bitonic) sequences, Bitonic Top-K reduces the workload by half in each iteration. WarpSelect maintains the top K elements processed so far, and new elements are placed in a few thread queues and merged into the top K results via bitonic sorting and merging. Due to the extensive use of shared memory and registers, the maximum K that Bitonic Top-K and WarpSelect are able to process is limited (256 for Bitonic Top-K and 2048 for WarpSelect). Meanwhile, WarpSelect has special merits: it can serve as a device function within other kernels, and it can process data on-the-fly because it maintains top-K results for all seen elements.

Besides these parallel top-K algorithms, researchers also build hybrid methods [12, 38]. For example, Dr. Top-K [12] computes top K delegates to reduce workload and performs a second top-K to get final results. As a hybrid method, it involves two top-K computations and needs a base top-K algorithm (like RadixSelect or Bitonic Top-K) as its building block, hence it benefits from a high-performance parallel top-K algorithm.

2.3 Parallel Radix Top-K

In radix top-K algorithms, a digit corresponds to a group of b continuous bits in the binary representation of an element. The algorithm processes an element from the most significant digit to the least significant digit, with one digit per iteration. For an element consisting of r bits, $\lceil \frac{r}{b} \rceil$ iterations are required.

Each iteration consists of four steps:

Compute histogram For every element, extract the b bits used in this iteration and transform them into a digit, which is in the range $[0, 2^b - 1]$. Compute a histogram to record the frequencies of the digits of all elements. That is, the ith counter of the histogram records the number of elements whose digits are equal to i.

Compute inclusive prefix sum of the histogram After this step, the ith entry of the prefix sum array, denoted as psum$[i]$, counts the number of elements that have the digit less than or equal to i.

Find target digit Find the digit that the Kth element should have. It is the j that satisfies psum$[j-1] < K$ and psum$[j] \geq K$.

Filtering The elements whose digits are less than j are guaranteed to be top-K elements so we store their values and indices in the results. The elements whose digits are larger than j are discarded as they definitely are not the top-K elements. The elements whose digits are equal to j could be the potential results so we store them and their indices to candidate buffers used as input lists in the next iteration.

At the end of an iteration, psum$[j-1]$ elements are identified as the results, so we update K to $(K - \text{psum}[j-1])$ and N to histogram$[j]$, which are used in the next iteration.

As an example, Fig. 1 illustrates how radix top-K finds the top 4 elements from a list of 9 elements.

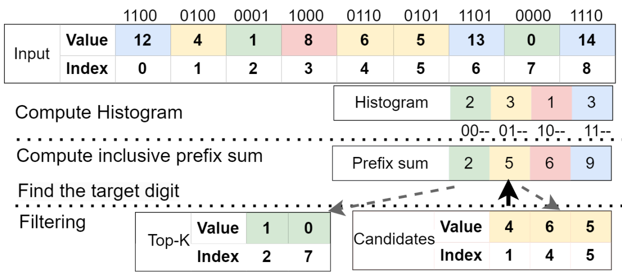


Figure 1: Radix top-K workflow for finding the top $K = 4$ elements from a list of $N = 9$ elements. Each element is represented with 4 bits and 2-bit digits are used. In the first iteration, we compute the histogram of the first 2-bit digits and convert it to prefix sum. Then we can infer that the target digit is '01' because its counter $5 \geq 4$. So values with digit '00' are top-K results, and values with digit '01' are candidates used as inputs in the next iteration.

2.4 GPU Architecture

The architecture of NVIDIA GPU is comprised of an array of multithreaded Streaming Multiprocessors (SMs) and is designed to execute thousands of threads concurrently. A function that runs on GPU is called a *kernel*.

Thread Hierarchy. NVIDIA GPUs group every 32 threads into a *warp* [24]. Warp-level primitives allow threads within a warp to communicate and exchange data collectively. Multiple warps (up to 32 warps) constitute a thread *block*. Threads within a block reside on the same SM and can communicate with each other via shared memory. Multiple blocks constitute a *grid*. A grid of blocks is launched for a kernel from CPU (*host*) and is executed on GPU (*device*).

Memory Hierarchy. Threads may access data from multiple memory spaces during their execution. There are three primary memory spaces. 1) *Device memory* is the largest memory on GPU. Although it is slower than other types of memory, device memory can be accessed by all threads. 2) *Shared memory* is a low-latency memory on SM and enables fast communication among threads within a block. 3) Each SM has a set of 32-bit *registers* that are partitioned among the threads. As the fastest memory on GPU, the number of registers consumed can influence the maximum number of blocks that can reside on one SM.

3 AIR TOP-K: ADAPTIVE AND ITERATION-FUSED RADIX TOP-K

We introduce AIR Top-K algorithm in this section. Algorithm 1 shows its pseudo-code, but a few details are left out for simplicity: output includes the value list V but not the index list I, and the early stopping strategy described in subsection 3.3 is omitted.

Algorithm 1: AIR Top-K

```
Input: A list $L$ of $N$ elements; $K$; parameter $\alpha$
Output: Value list $V$ of length $K$
1  $V \leftarrow [\,]; C' \leftarrow N; C \leftarrow N; target\_digit \leftarrow 0$
2  $buf \leftarrow [\,]$
3  for $iter \leftarrow 1$ to $\lceil \frac{r}{b} \rceil$ do
4      $buf \leftarrow$ iteration_fused_kernel($iter$, $buf$)
5  last_filter_kernel($\lceil \frac{r}{b} \rceil$, $buf$)
6  Function iteration_fused_kernel(iter, buf'):
7      if $C' \geq N/\alpha$ then
8          $buf' = L$
9      $buf \leftarrow [\,]$; $histogram \leftarrow [0, 0, \ldots, 0]$
10     parallel foreach $e \in buf'$ do
11         if $iter = 1$ then
12             $digit \leftarrow$ digit of $e$ for iteration 1
13             $histogram[digit] \leftarrow histogram[digit] + 1$
14         else
15             $digit' \leftarrow$ digit of $e$ for iteration $(iter - 1)$
16             if $digit' = target\_digit$ then
17                 if $C < N/\alpha$ then
18                     add $e$ to $buf$
19                 $digit \leftarrow$ digit of $e$ for iteration $iter$
20                 $histogram[digit] \leftarrow histogram[digit] + 1$
21             else if $digit' < target\_digit$ then
22                 add $e$ to $V$
23     if is the last thread block then
24         $psum \leftarrow$ inclusive prefix sum of $histogram$
25         $target\_digit \leftarrow j$ that satisfies $psum[j-1] < K$ and
             $psum[j] \geq K$
26         $C' \leftarrow C$
27         $C \leftarrow histogram[target\_digit]$
28         $K \leftarrow K - psum[target\_digit - 1]$
29     return $buf$
```

3.1 Iteration-fused Design

All previous parallel radix top-K implementations for GPU require host engagement during the processing because CPU is needed for either processing intermediate data or determining the termination of processing. Intermediate data is transferred between host and device via PCIe, which has much higher latency and lower bandwidth than GPU memory, and the overheads could dominate the whole computation when N is not large. Also, synchronization between the host and device prevents the implementation from being asynchronous with respect to the host.

One advantage of radix top-K over other partition-based methods is that the number of iterations does not depend on input. For 32-bit data and 8-bit digits, the processing is guaranteed to be complete in $\lceil \frac{32}{8} \rceil = 4$ iterations. Recognizing the maximum number of iterations eliminates the requirement of CPU assistance in determining the end of the processing. So it is possible to run all the computation of the radix top-K algorithm on GPU, and the CPU is only in charge of launching the kernels.

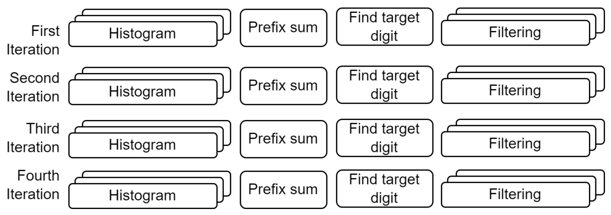


Figure 2: A naïve radix top-K algorithm for GPU. For 32-bit data and 8-bit digits, it takes 4 iterations and four kernel calls per iteration. So 16 kernel calls are used in total. Stacked rectangles indicate a kernel with multiple thread blocks, while the single rectangle marks a single thread block kernel.

A naïve way is to run the four steps mentioned in section 2.3, each as a kernel, in turn on GPU. Fig. 2 shows how the kernels are organized. However, it requires 4 kernel calls for each iteration and a total of 16 kernel calls in the example of Fig. 2. In each iteration, we need to load the input data twice, one for histogram computation and one for filtering, resulting in lots of device memory access. We design an iteration-fused kernel in AIR Top-K to reduce kernel calls and redundant data access.

By viewing all iterations as a whole, we find that the filtering for the current iteration and histogram computation for the next iteration can be done within the same kernel. The benefits are not only reducing the number of kernel calls but also decreasing the size of device memory loading. During the filtering, if an element is a candidate for the next iteration, we immediately extract the needed bits from it, transform bits to a digit, and increase the corresponding counter in the histogram. So the data loading for the histogram computation step is saved. Assume G_i is the data size loaded for the ith iteration. For 8-bit digits, the total data loading size of the naïve method is $\sum_{i=1}^{4} 2G_i$, whereas after the optimization it becomes $2G_1 + \sum_{i=2}^{4} G_i$. In the worst case, the first three iterations can not eliminate any candidate, which means $G_i = N$ and the total data loading size is reduced from $8N$ to $5N$ after the optimization. Lines 15–22 of Algorithm 1 show how the filtering kernel of the previous iteration is fused with the histogram kernel. Note that input elements are loaded once and used for both filtering and histogram computation. This optimization would reduce the 16 kernel calls in Fig. 2 to 13 calls.

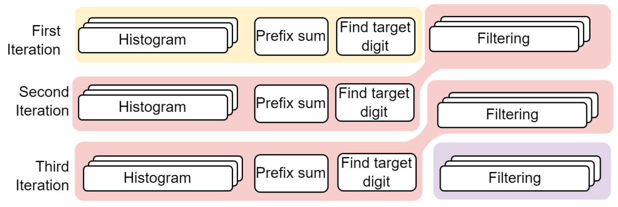


Figure 3: Iteration-fused design in AIR TOP-K. We can use 11-bit digits for 32-bit data, so it takes 3 iterations. The 4 computation steps are fused into an iteration-fused kernel, although it does not contain the filtering step for the first iteration. The last filtering step is done in a separate kernel. Only 4 kernel calls in total are needed.

The kernels for calculating prefix sum and finding the target digit can also be fused. The input for them is the computed histogram rather than elements. The length of the histogram is 2^b. If we set b to a suitable value, like 8 or 11, the length would not be too large for a single block with hundreds of threads to compute the prefix sum and find the target digit. Then these two kernels can be fused because a block-wide synchronization can ensure that prefix sum computation is finished before starting to find the target digit. Furthermore, because only one thread block is needed, the last running thread block for histogram computation can be used to continue computing the prefix sum. In this way, all four steps can be fused in one iteration-fused kernel. With this optimization, we reduce the number of kernel calls from 13 to 5.

Because the prefix sum is computed parallelly on GPU rather than sequentially on CPU, we can afford to set $b = 11$ rather than using a smaller one like $b = 8$. For 32-bit data, this reduces the number of iterations from 4 to 3. So only 4 kernel calls, including 3 calls of the iteration-fused kernel and a call of the filtering kernel at last, are needed as shown in Fig. 3. Lines 3–5 in Algorithm 1 shows the general workflow. It is possible to fuse the last filtering kernel too, but we do not adopt this strategy in our experiments because it reduces performance for adversarial distribution.

In conclusion, because of the iteration-fused design, AIR TOP-K runs fully on GPU except for kernel launch, minimizes intermediate data transfer between host and device, and reduces data loading size and kernel launch time substantially.

3.2 Adaptive Strategy Based on Data Distribution

A common concern of radix top-K is that its performance varies with data distributions. If the first 11 bits of all elements are evenly distributed, the first iteration of radix top-K is expected to reduce the workload to 1/2048 of the original size. In another extreme, if all the first 11 bits are identical, no element can be removed. This does occur in practice when the input value range is narrow. For example, for IEEE-754 floating point numbers within the range [1.0, 1.00049], their corresponding bits are in the range [0x3F800000, 0x3F800FFF], meaning that the first 20 bits are the same. We refer to this kind of distribution as a "radix-adversarial" distribution.

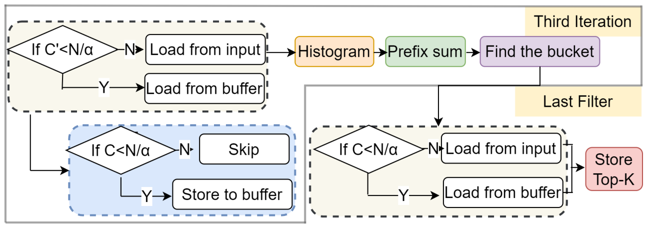


Figure 4: Adaptive strategy in the third iteration and the last filtering step. In the third iteration, the loading location of inputs depends on the number of candidates in the previous iteration (C'). Whether to store candidates in buffers depends on the number of candidates in the current iteration (C).

Radix top-K algorithm in section 2.3 filters the candidates of the next iteration (along with their indices) to buffers. For evenly distributed data, this decreases workload drastically, e.g., only 1/2048 elements will be loaded in the next iteration. However, for radix-adversarial distribution, it leads to large memory traffic waste because the candidates might remain the same after iterations, and keeping storing the same candidates is unnecessary. Even worse, it requires storing N values and N indices, which are twice the original size.

The alternative is using no candidate buffer and always loading data from the original input. This method avoids storing the candidates, thus reducing the memory traffic for radix-adversarial distribution. However, for evenly distributed data, we need to load the whole inputs in every iteration.

So, whether using candidate buffers is beneficial depends on data distribution, which is usually unknown in advance. To address this dilemma, we seek additional information from the radix top-K algorithm. The key observation is that the histogram represents a rough estimate of the data distribution, and particularly it records the number of candidates for the next iteration. We can adaptively decide whether to store candidates based on this.

We use C to indicate the number of candidates and N as the total number of inputs. Here we introduce a parameter α. Only when $C < N/\alpha$, we store candidates in buffers (line 17 in Algorithm 1). Because candidate storing might be uncoalesced, the optimal value of α should be determined by experiments in practice. However, we can still infer its lower bound. Using candidate buffers involves storing and loading C values and C indices, so $4C$ memory access in total. Not using candidate buffers means loading N values. So we should only use candidate buffers when $4C < N$. It means the lower bound of α is 4.

We also need to determine where to load the elements at the beginning of the iteration-fused kernel as shown at line 7 in Algorithm 1, where C' denotes the number of candidates in the previous iteration. If $C' \geq N/\alpha$, we know that candidate buffers are not used and we should load from the original input L. Fig. 4 depicts the adaptive strategy used in the last iteration and in the last filtering step.

Note that this adaptive strategy automatically chooses different buffering behavior in each iteration. For data distribution where some bits are adversarial distributed and other bits are uniformly distributed, it might avoid storing candidates in the first few iterations and adopt candidate buffers in later iterations. In general, it saves memory access when possible.

The adaptive strategy brings one more benefit. It implies the maximum size of the candidate buffer is N/α. For tasks with more rigorous memory requirements, we can increase α to decrease memory footprint. In extreme cases, setting $\alpha = N$ ensures no candidate buffer is required.

3.3 Early Stopping

A trivial case of the top-K problem is that K equals N. Then all the input elements must be the top-K elements. Although this case is rarely used in practice, it does happen during the radix top-K computation. At the end of each iteration, we update N to be the number of candidates and decrease K by the number of top-K results found in this iteration, as shown at lines 27–28 in Algorithm 1. It is possible that the updated K equals the updated N. When this happens, the work of the next iteration becomes trivial because all the elements in the candidate buffer belong to the results. Thus, we can stop earlier without further processing.

4 GRIDSELECT: OPTIMIZE WARPSELECT WITH SHARED QUEUE AND MULTIPLE THREAD BLOCKS

WarpSelect [16] maintains a thread queue for each thread and there are 32 thread queues in total for a warp. Each thread can insert a potential candidate into its own queue independently. When any thread queue is full, a bitonic sorting is carried out to sort all thread queues and then a bitonic merging is used to merge thread queues into the results. However, such bitonic sorting and merging are costly operations. If qualified elements are centralized in a certain thread queue, WarpSelect must frequently call these expensive operations even if other thread queues are empty. This will greatly hinder performance. Another issue is that the thread queues reside in registers. Although operations in registers are highly efficient, it has the disadvantage of increasing register pressure.

We propose a single shared queue with parallel two-step insertion to solve both issues. The insert operation is still parallel so it remains efficient, and it might be done in two steps to ensure that sorting and merging occur only when the queue is full and that the shared memory footprint is small.

First, instead of using per-thread queues, we use a queue stored in shared memory that can be accessed by all 32 threads in a warp. To limit the footprint of shared memory, the size of the shared queue is set to 32.

Second, parallel insertion is done in two steps. Each thread compares an input element to the Kth element known so far. If the element is smaller, it is qualified to be inserted in the shared queue. To prevent multiple threads from storing elements in the same place, we compute parallelly a storing position for every thread using the warp ballot function, a collective primitive that broadcasts which threads hold qualified candidates. By counting how many preceding threads will perform insertion, a thread knows its unique storing position. Threads with storing positions smaller than the queue size can insert elements immediately. After that, if the queue is full, sorting and merging operations are performed to update the top-K results and the queue is cleared. Then threads with storing positions equal to or larger than 32 can perform insertion by shifting their positions down by 32. So the insertion might be done in two steps, however, the second step is needed only when the queue is full. Also, expensive sorting and merging operations are called only when the queue is full. In Fig. 5, we explain our strategy with 8 threads.

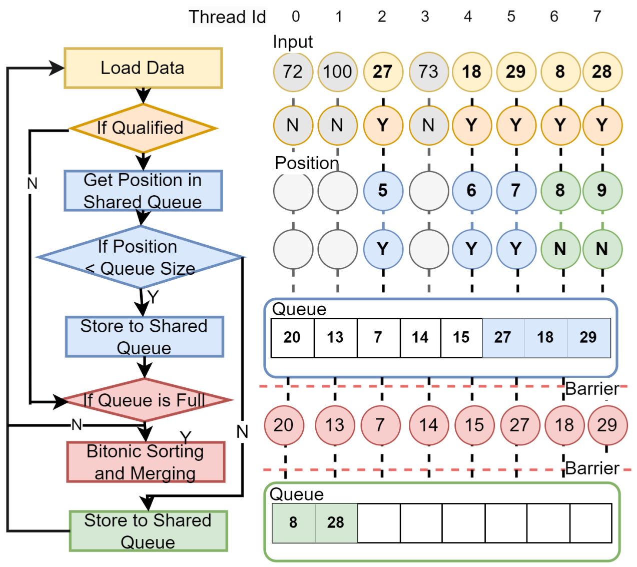


Figure 5: Parallel two-step insertion. Assume a warp contains 8 threads. Each thread loads an element and decides whether it is qualified to be inserted into the shared queue. Then the storing positions are computed with the warp ballot function. Insertion is done in two steps. Elements 27, 18 and 29 (marked blue) are inserted immediately because their storing positions are smaller than the queue size. After that, as the queue is full, bitonic sorting and merging operations are used to update results. Elements 8 and 28 (marked green) are inserted in a second step after the queue is flushed because their storing positions exceed the queue size.

We also improve WarpSelect by adopting multiple thread blocks. WarpSelect uses only one warp to compute top-K results, while BlockSelect from Faiss library [10] extends it to use a thread block containing up to 4 warps. Because a thread block runs on one SM and a GPU can have a hundred SMs, WarpSelect and BlockSelect could not fully utilize a GPU. So We further extend them to use multiple thread blocks to increase parallelism. Following the naming convention, we refer to our implementation as GridSelect. Note that GridSelect uses a shared queue rather than per-thread queues, although the name does not reflect this.

5 PERFORMANCE EVALUATION

We have open-sourced AIR top-K in RAFT (Reusable Accelerated Functions and Tools) library from NVIDIA RAPIDS [28] and use it for performance comparison in this section. In all experiments, the parameter α is set to 128, a value determined empirically. Experiments are conducted with NVIDIA CUDA 12.0 [23] mainly on NVIDIA A100 GPU [8], which is widely used in AI, data analytics, and HPC. The data type is 32-bit floating point. All the reported running time is the average of 100 runs.

5.1 Comprehensive Benchmarks for Parallel Top-K Algorithms

We compare the performance of our parallel algorithms with 8 open-source implementations listed in Table 1. For sorting, the highly efficient radix sort from the CUB library [29] is included. For partial sorting methods, we use the Bitonic Top-K from the DrTopK library [11], and WarpSelect and BlockSelect from the widely used Faiss library [10]. For partition-based methods, QuickSelect, BucketSelect, and SampleSelect from the GpuSelection library are included. The most recent RadixSelect implementation that we can find is from the DrTopK library and it is also included.

Note that we use the base top-K implementations from the DrTopK library, not the Dr. Top-K algorithm [12] itself, which is a hybrid algorithm built upon these base implementations, and is orthogonal to and can benefit from our new methods.

Table 1: Previous Algorithms in the Benchmark

Algorithm	Library	Category
Sort [20]	CUB [29]	Sorting
WarpSelect [16]	Faiss [10]	Partial Sorting
BlockSelect [16]	Faiss [10]	Partial Sorting
Bitonic Top-K [32]	DrTopK [11]	Partial Sorting
QuickSelect [9]	GpuSelection [30]	Partition-based
BucketSelect [1]	GpuSelection [30]	Partition-based
SampleSelect [31]	GpuSelection [30]	Partition-based
RadixSelect [12]	DrTopK [11]	Partition-based

We use three different kinds of data distributions: uniform distribution in the range of $(0, 1]$, normal distribution with mean 0 and standard deviation 1, and a specially designed radix-adversarial distribution, where the first M bits of all the input elements are the same. In this benchmark, we set $M = 20$.

For comprehensive comparisons of these different algorithms, we explore their performance for a wide range of N and K values. Fig. 6 and Fig. 7 recorded all the results that passed the correctness verification. There are constraints for some algorithms hence no result, for example, the largest K that WarpSelect, BlockSelect, and GridSelect support is 2048.

Besides various N and K values, we also evaluate the performance for different batch sizes. To increase throughput, several top-K problems of the same N and K can be packed into a batch and solved at once. The number of problems in a batch is called the *batch size*. In addition to batch size 1, which is used in most studies, we also test batch size 100, which is usually large enough for online services.

Each sub-figure in Fig. 6 plots the running time with respect to K varying from 2^3 to 2^{20} and a constant N (one of 2^{15}, 2^{20}, 2^{25} and 2^{30}). From these 12 sub-figures, sorting and partition-based methods show stable performance for various K values when N is fixed. In contrast, there is a significant increase in time for partial sorting algorithms as K increases. It is because the complexity of the underlying bitonic sorting network they use is $O(\log^2 K)$. GridSelect are more likely to be faster than AIR Top-K when $K < 256$, although the performance gap is small except for radix-adversarial distribution. AIR Top-K remains the fastest for other cases.

Fig. 7 shows the performance variation with increased N and a constant K (one of 32, 256, and 32768). Besides batch size 1, Fig. 7 also shows the performance for batch size 100. The first two columns show that the curves of WarpSelect and BlockSelect rise more sharply than other methods for batch size 1. This variation comes from their limited parallelism. The third row of Fig. 7 shows that the performance of many partition-based methods deteriorates under radix-adversarial distribution. AIR Top-K and GridSelect are much faster than other algorithms for all cases, even for radix-adversarial distribution.

Table 2: Summary of Speedup Range

Batch	Distribution	AIR Top-K *vs.* RadixSelect	GridSelect *vs.* BlockSelect	AIR Top-K *vs.* SOTA
1	Uniform	2.02–21.48	1.09–880.6	1.62–6.81
1	Normal	1.99–21.22	1.09–882.29	1.53–7.34
1	Adversarial	1.98–10.78	1.09–875.11	1.44–5.0
100	Uniform	13.54–574.17	1.11–9.82	1.56–27.43
100	Normal	10.26–574.78	1.19–9.82	1.42–31.91
100	Adversarial	8.01–540.15	1.14–9.83	1.38–26.71

Table 2 summarizes the speedup ranges from all the experiments in Fig. 6 and 7. To quantify the performance of our algorithms with respect to all previous algorithms, we regard the best performance of all previous algorithms for each combination of N, K, and batch size as the performance of a virtual state-of-the-art algorithm, referred to as SOTA. The rightmost column in Table 2 records the speedup of AIR Top-K over SOTA. AIR Top-K achieved 1.44–7.34× and 1.38–31.91× speedup for batch sizes 1 and 100, respectively. This demonstrates that the performance of AIR Top-K is superior to all previous algorithms under all three distributions. In Fig. 6 and 7, AIR Top-K is faster than others by one order of magnitude in most cases, only our GridSelect is faster than it in just a few cases. Even for the radix-adversarial distribution, AIR Top-K is still the most competitive one.

As for choosing which one to use from our two parallel algorithms, we list the following guidelines: 1) To process data on-the-fly, use GridSelect. 2) For large N and small K ($<$ 256), their relative performance varies under different data distributions. 3) In most other cases, use AIR Top-K.

5.2 Performance Analysis for AIR Top-K

In this part, we show the performance of our optimizations for AIR Top-K. Unless otherwise specified, uniform distribution is used.

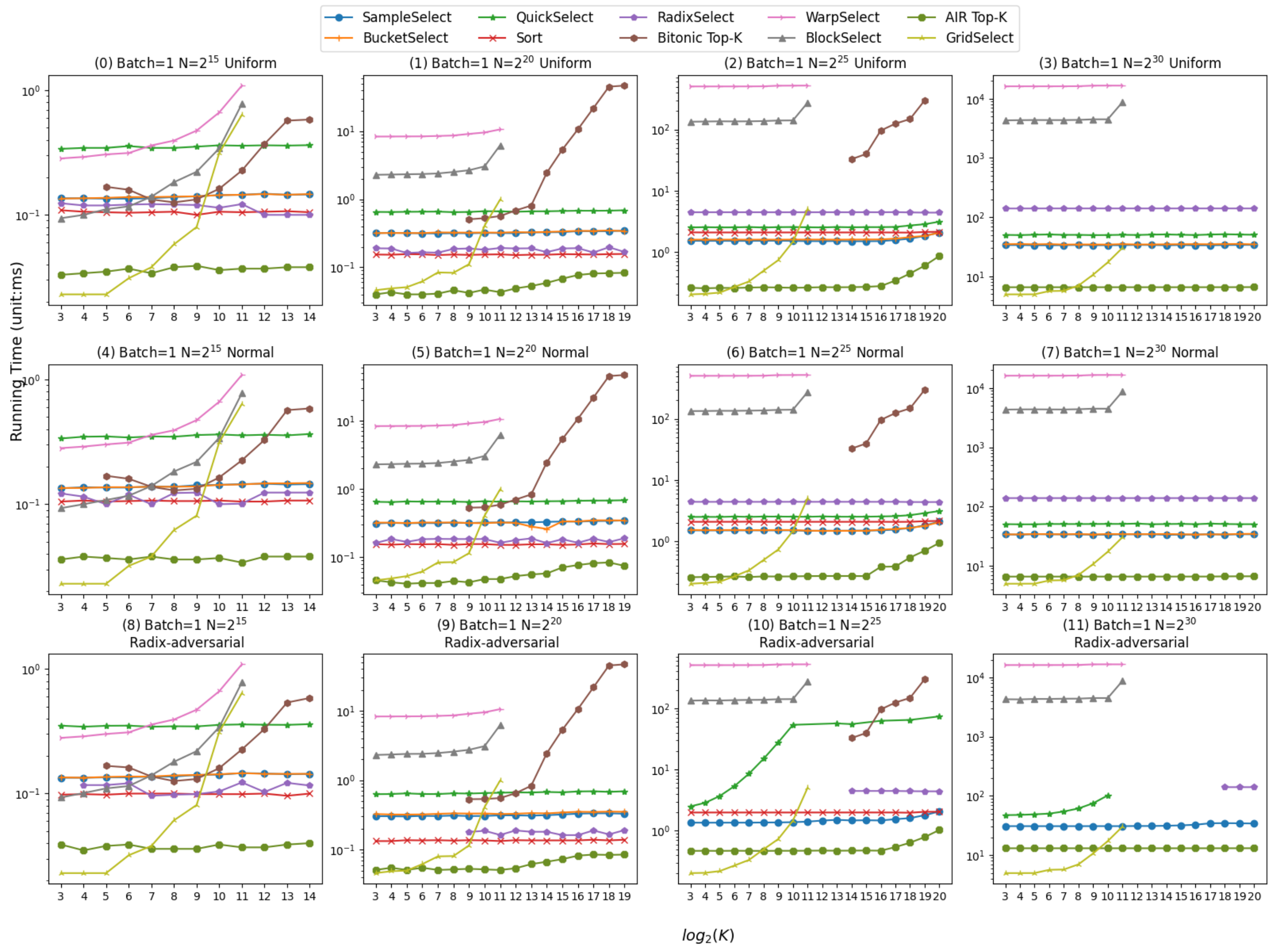


Figure 6: Performance comparison for different K values with uniform, normal, and radix-adversarial distributions

5.2.1 Iteration-fused Design. As shown in Table 2, AIR Top-K achieves 1.98–21.48× and 8.01–574.78× speedup over RadixSelect for batch sizes 1 and 100, respectively.

To illustrate the advantage of iteration-fused design, we record the breakdown timeline of RadixSelect and AIR Top-K for the case $N = 2^{23}$ and $K = 2048$ in Fig. 8. First, there are notable white spaces in the timeline of RadixSelect. These denote the overheads of host-device synchronizations or indicate that the GPU is idle while the CPU is processing intermediate data. In contrast, the timeline of AIR top-K is quite tight, suggesting the synchronization overheads are minimized due to the iteration-fused design. Second, RadixSelect has lots of data transfer between host and device, shown as "MemecpyHtoD" (memory copy from host to device) and "MemecpyDtoH" (memory copy from device to host) in the figure. AIR Top-K has no such data exchange. Third, AIR Top-K has fewer kernel calls. There are three "iteration_fused_kernel" calls in its timeline although the gaps between these calls are too narrow to be observed. Fourth, the kernel "CalculateOccurence" from RadixSelect takes much longer time than "iteration_fused_kernel" from AIR Top-K. One reason is that the iteration-fused design can reduce device memory access hence make high utilization of GPU.

To further investigate the GPU utilization, we use Nsight Compute [26] to analyze the kernels of AIR Top-K. The metric "GPU Speed Of Light Throughput (SOL)" reports the achieved percentage of utilization with respect to the theoretical maximum throughput of compute and memory resources. We set $N = 2^{30}$ and $K = 2048$, and list the "Compute SOL" and "Memory SOL" in Table 3. We use a large N for this analysis because the data size needs to be large enough to saturate GPU resources. The column "Time Percentage" records the percentage of the running time for each kernel call. The first and second calls of "iteration_fused_kernel" take up the majority of the time. The "Memory SOL" of these two kernels are 91.27% and 89.08%, respectively, indicating that "iteration_fused_kernel" achieves quite high memory utilization. Their "Compute SOL" are 49.29% and 50.30%, so the memory resource is more heavily utilized than the compute resource. In brief, AIR Top-K achieves high memory utilization and is memory-bound.

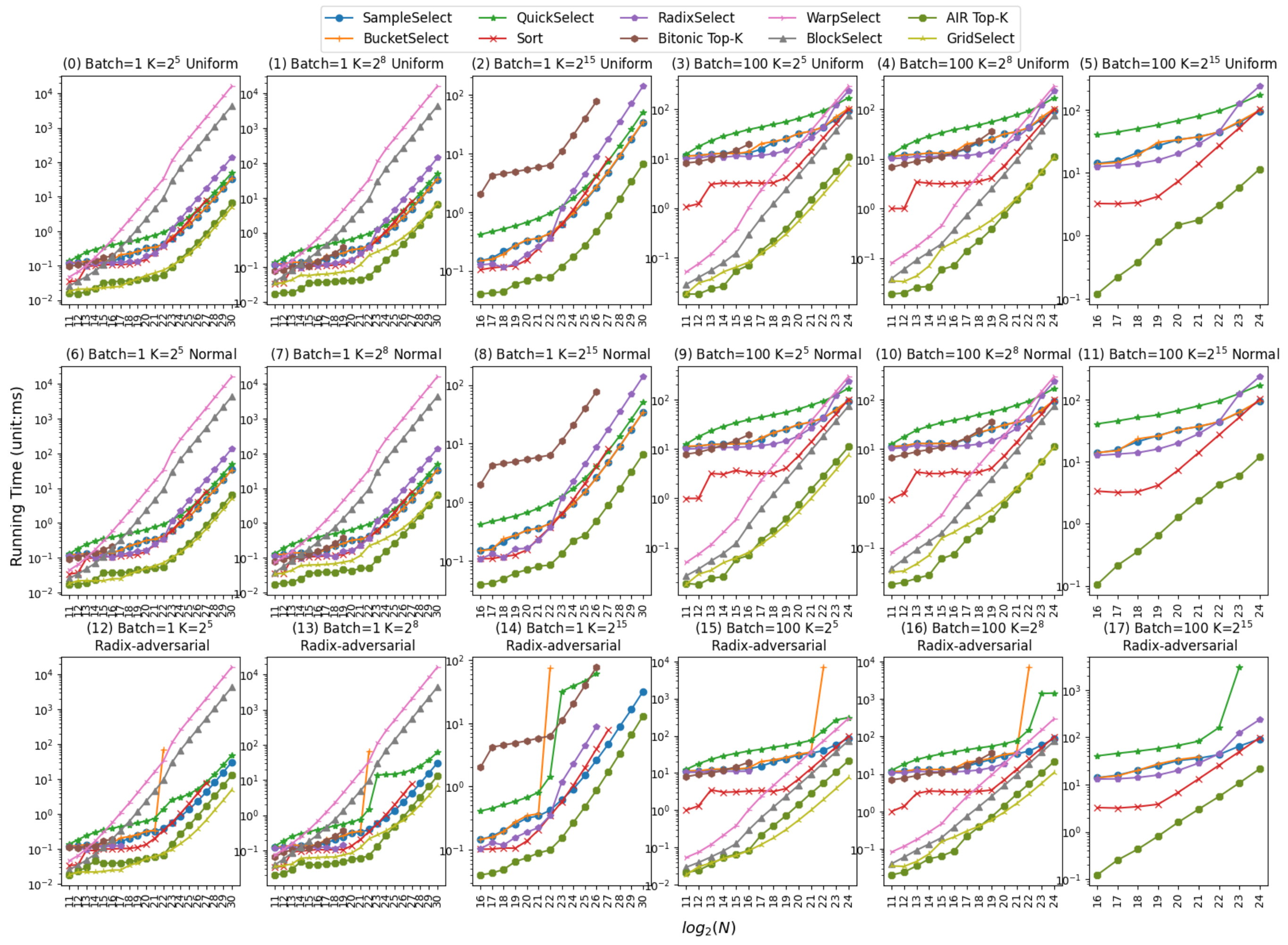


Figure 7: Performance comparison for different N values with uniform, normal, and radix-adversarial distributions

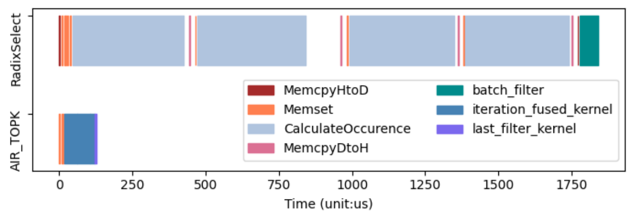


Figure 8: Timelines of RadixSelect and AIR Top-K.

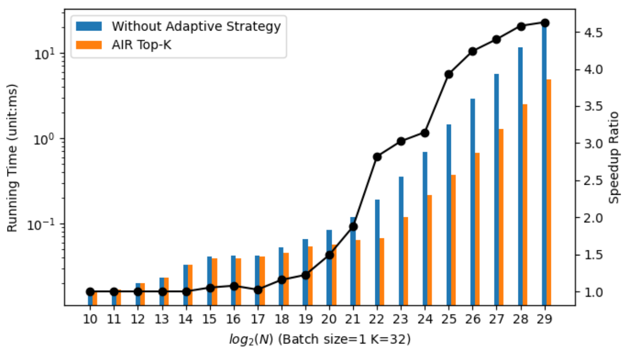
Table 3: Kernels Performance Analysis for AIR Top-K

Kernel Call	Time Percentage	Memory SOL	Compute SOL
iteration_fused_kernel(1)	49.29%	91.27%	31.43%
iteration_fused_kernel(2)	50.30%	89.08%	44.69%
iteration_fused_kernel(3)	0.29%	8.22%	20.92%
last_filter_kernel	0.12%	4.68%	21.15%

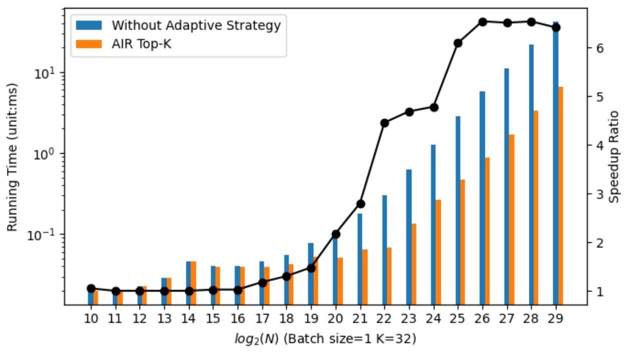
5.2.2 *Adaptive Strategy Based on Data Distribution.*

In section 3.2, we introduce the adaptive strategy to mitigate the performance degradation under radix-adversarial distribution. To illustrate the performance difference with and without the adaptive strategy, we use radix-adversarial data distributions with $M = 10$ and $M = 20$ in Fig. 9. The adaptive strategy improves performance significantly in these two scenarios. Also, the speedup increases as N increases, because the adaptive strategy reduces a larger amount of device memory traffic as data size increases.

The speedup for $M = 10$ and $M = 20$ can reach up to 4.62× and 6.53×, respectively. It is greater for $M = 20$ because a larger M means a more centralized data distribution and a greater demand for device memory access. The adaptive strategy is more beneficial in this case.



(a) Radix-adversarial distribution with $M = 10$



(b) Radix-adversarial distribution with $M = 20$

Figure 9: Running time of AIR Top-K with and without adaptive strategy

5.2.3 Early Stopping. To show the benefit of the early stopping strategy, we implement a version without early stopping and compare it with AIR Top-K. From Fig. 10, we can find that early stopping helps to reduce the running time, achieving up to 18.7% performance improvement.

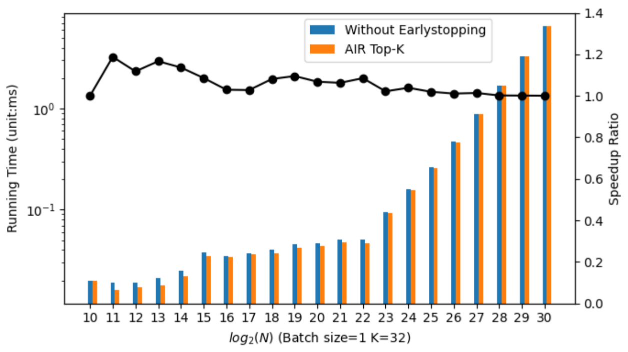


Figure 10: Running time of AIR Top-K with and without early stopping

5.3 Performance Analysis for GridSelect

BlockSelect extends WarpSelect by using a thread block containing up to 4 warps. In Fig. 6 and 7, BlockSelect outperforms WarpSelect consistently. So we use BlockSelect as the baseline to evaluate the performance of our method GridSelect. As shown in Table 2, GridSelect achieves up to 882.29× speedup compared with BlockSelect. The speedup is especially high when the batch size is one and N is large. This is the result of using multiple thread blocks in GridSelect so that most GPU resources are utilized. In contrast, BlockSelect uses a single thread block hence only one SM out of the 108 SMs on A100 GPU is utilized.

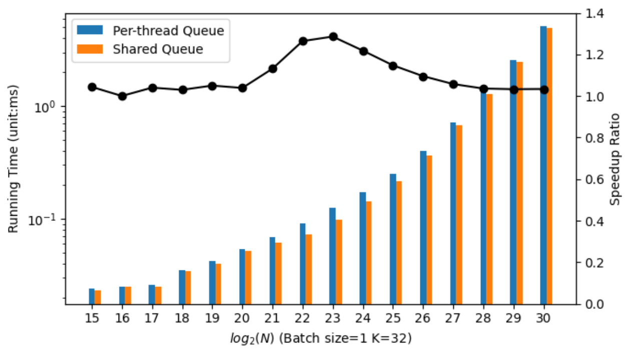


Figure 11: Running time of GridSelect with per-thread queues and shared queue

To demonstrate the benefit of the shared queue with parallel two-step insertion, we implemented a variant of GridSelect using per-thread queues, like BlockSelect does, and compared it with the GridSelect using our proposed shared queue. The shared queue implementation achieves up to 1.28× speedup, as shown in Fig. 11.

5.4 Performance on Different GPUs

Besides A100 GPU, we also run experiments on its successor H100 SXM GPU[25], as well as a representative deep learning inference GPU A10 [22], to show the device-wise performance difference. In this part, we compare our algorithms with SOTA for $N = 2^{30}$ under the uniform distribution. Fig. 12 shows that AIR Top-K achieves 5×, 5× and 3× speedup over SOTA on A100, H100 and A10, respectively. And comparing AIR Top-K with GridSelect, we can find that GridSelect is faster when $K \leq 512$ on A10 and when $K \leq 128$ on A100 and H100.

AIR Top-K achieves 3× speedup on A100 over A10, and 2× speedup on H100 over A100. As the memory bandwidths of A10, A100, and H100 are 0.6 TB/s [22], 1.55 TB/s [21], and 3.35 TB/s [25], respectively, the performance differences roughly align with the memory bandwidth differences because AIR Top-K is memory-bound as described in section 5.2.1.

5.5 Performance on Real-world Datasets

Besides synthetic datasets, we also compare the performance of top-K algorithms using real-world datasets. Approximate Nearest Neighbor (ANN) search is an important machine learning algorithm

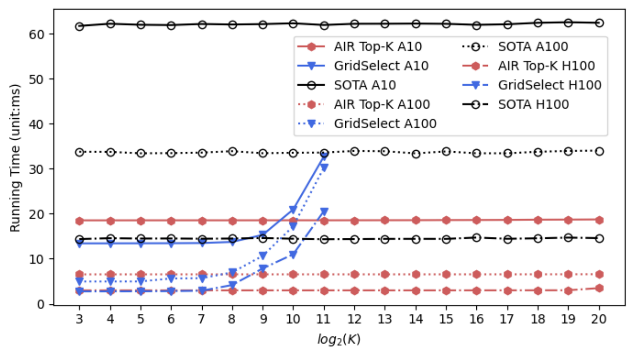


Figure 12: Running time on different GPUs

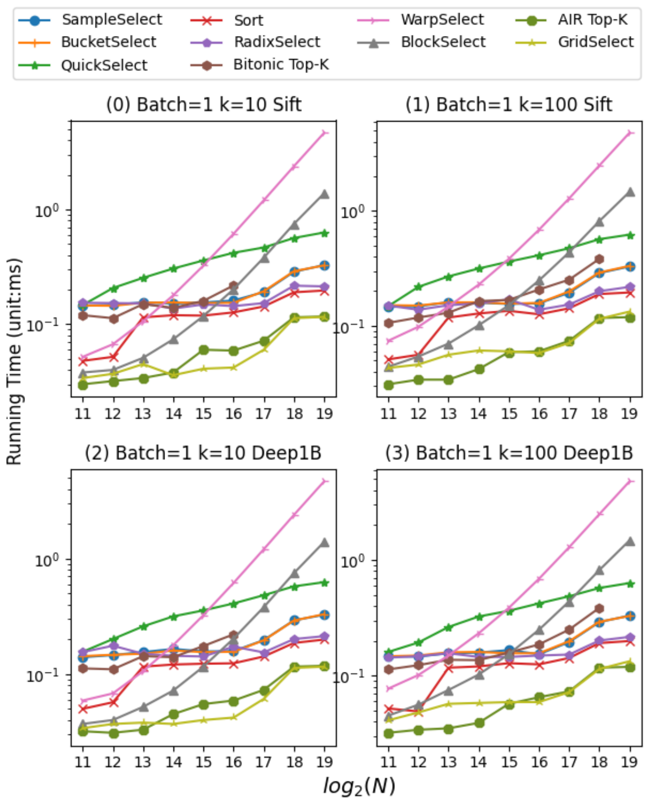


Figure 13: Performance comparison for $K = 1$ and $K = 100$ with real-world datasets DEEP1B and SIFT

and is widely used in recommender systems, information retrieval, computer vision, etc. It is used to retrieve similar vectors in a vector database containing a large number of candidate vectors to a query vector. One key step in ANN is top-K calculation which chooses K candidate vectors of the smallest distances to the query vector.

Two commonly used datasets in ANN algorithm research, DEEP1B [3] and SIFT [15], are used in this experiment to compare different top-K algorithms. Vectors of DEEP1B are extracted from the last layers of convolutional neural networks, and vectors of SIFT are local descriptors of images computed by scale-invariant feature transform. We use the corresponding datasets prepared in the widely used ANN Benchmarks [2]: DEEP1B contains $9,990,000$ vectors in 96 dimensions, and SIFT contains $1,000,000$ vectors in 128 dimensions. We calculate the distances between a query vector and all candidate vectors to get a distance array used as the input of top-K procedures. For each dataset, 1000 queries are used and the running time for each combination of N and K is the average of 1000 runs.

In our experiments, K is set to 10 and 100, the two typical values used in ANN Benchmarks [2]. N varies from 2^{11} to 2^{19} because usually only a subset of candidate vectors are chosen in an ANN algorithm to avoid exhaustive distance computation.

Fig. 13 shows the performance results for these two real-world datasets. When K is as small as 10, GridSelect shows an advantage over AIR Top-K for many N values. When K is 100, AIR Top-K is faster, especially for small N values. Most importantly, the performance for these two datasets is consistent with the results for synthetic datasets. AIR Top-K and GridSelect are always faster than other methods, and the gap between our method and other methods gets larger as N increases.

6 CONCLUSION

With high arithmetic throughput and memory bandwidth, GPUs are widely deployed in various deep learning, machine learning, and HPC applications. As a fundamental problem, Top-K demands a widely applicable and high-efficiency parallel solution.

In this work, we introduce two parallel top-K algorithms, AIR Top-K and GridSelect. For AIR Top-K, we first propose an iteration-fused design, which does not require CPU engagement except for kernel launch, minimizes device-wide synchronization, and significantly reduces device memory access. Then we design an adaptive strategy to automatically determine whether to store the candidates so that the device memory traffic is minimized regardless of the data distribution. GridSelect is based on WarpSelect. We propose a single shared queue with parallel two-step insertion to reduce costly operations and register pressure. With these efforts, our algorithms outperform previous ones by a large margin. We have open-sourced a carefully crafted implementation of AIR Top-K in RAFT [28]. We believe our work can facilitate the acceleration of a huge number of applications on GPU.

7 ACKNOWLEDGMENTS

We would like to thank Artem Chirkin and Tamas Feher for integrating our code into RAFT library. We also thank Peng Wang, Guang Ji, and Guofeng Zhou for providing valuable feedback on the manuscript.

REFERENCES

[1] 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.
[2] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.
[3] Artem Babenko and Victor S. Lempitsky. 2016. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016.* IEEE Computer Society, Las Vegas, 2055–2063.

[4] Ricardo Barrientos, J.I. Gomez, Christian Tenllado, and Manuel Prieto Matias. 2010. Heap based k-nearest neighbor search on GPUs. *Congreso Espanol de Informática (CEDI)* 1, 7 (01 2010), 559–566.

[5] Ricardo J Barrientos, Fabricio Millaguir, José L Sánchez, and Enrique Arias. 2017. GPU-based exhaustive algorithms processing kNN queries. *The Journal of Supercomputing* 73, 10 (2017), 4611–4634.

[6] Nathan Bell and Jared Hoberock. 2012. Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 359–371.

[7] 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* (Melbourne VIC, Australia) *(WSDM '19)*. Association for Computing Machinery, New York, NY, USA, 456–464.

[8] Jack Choquette and Wish Gandhi. 2020. NVIDIA A100 GPU Performance & Innovation for GPU Computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, IEEE, Palo Alto, CA, 1–43.

[9] 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.

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

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

[12] 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 39, 14 pages.

[13] 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.

[14] Bonan Huang, Jinlan Gao, and Xiaoming Li. 2009. An empirically optimized radix sort for GPU. In *2009 IEEE international symposium on parallel and distributed processing with applications*. IEEE, IEEE, Chengdu, 234–241.

[15] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (Jan. 2011), 117–128. https://inria.hal.science/inria-00514462

[16] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data* 7, 3 (2021), 535–547.

[17] Ivan Komarov, Ali Dashti, and Roshan M D'Souza. 2014. Fast k-NNG construction with GPU-based quick multi-select. *PloS one* 9, 5 (2014), e92409.

[18] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, Vancouver, BC, Canada, 14 pages. https://openreview.net/forum?id=SkhQHMW0W

[19] Abdelhamid Malki, Sidi-Mohamed Benslimane, Mimoun Malki, Mahmoud Barhamgi, and Djamal Benslimane. 2020. Top-k query optimization over data services. *Future Generation Computer Systems* 113 (2020), 1–12.

[20] Duane Merrill. 2015. Cub. https://on-demand.gputechconf.com/gtc/2015/presentation/S5617-Duane-Merrill.pdf

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

[22] Nvidia. 2021. NVIDIA A10 Tensor Core Gpu. https://www.nvidia.com/en-us/data-center/products/a10-gpu/

[23] Nvidia. 2022. CUDA 12.0 Release Notes. https://docs.nvidia.com/cuda/archive/12.0.0/cuda-toolkit-release-notes/index.html

[24] NVIDIA. 2023. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[25] Nvidia. 2023. NVIDIA H100 Tensor Core Gpu. https://www.nvidia.com/en-us/data-center/h100/

[26] NVIDIA. 2023. NVIDIA Kernel Profiling Guide. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

[27] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-efficient and practical parallel in-place radix sorting. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. IEEE, Phoenix AZ USA, 213–224.

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

[29] NVIDIA Research. 2022. cub::DeviceRadixSort. https://nvlabs.github.io/cub/structcub_1_1_device_radix_sort.html

[30] Tobias Ribizel. 2020. gpu selection. https://github.com/upsj/gpu_selection

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

[32] 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*. Association for Computing Machinery, Houston TX USA, 1557–1570.

[33] Xiaoxin Tang, Zhiyi Huang, David Eyers, Steven Mills, and Minyi Guo. 2015. Efficient selection algorithm for fast k-nn search on gpus. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, IEEE, Hyderabad, India, 397–406.

[34] Polychronis Velentzas, Panagiotis Moutafis, and George Mavrommatis. 2021. An Improved GPU-Based Algorithmfor Processing the k Nearest Neighbor Query. In *24th Pan-Hellenic Conference on Informatics* (Athens, Greece) *(PCI 2020)*. Association for Computing Machinery, New York, NY, USA, 372–375.

[35] Wikipedia. 2022. Selection algorithm. https://en.wikipedia.org/wiki/Selection_algorithm

[36] 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.

[37] Hamed Zamani, Susan Dumais, Nick Craswell, Paul Bennett, and Gord Lueck. 2020. Generating Clarifying Questions for Information Retrieval. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 418–428.

[38] Vasileios Zois, Vassilis J Tsotras, and Walid A Najjar. 2019. GPU accelerated top-k selection with efficient early stopping.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

This paper proposes two parallel top-K algorithms for GPU: AIR Top-K (Adaptive and Iteration-fused Radix Top-K) and GridSelect. And we provide a comprehensive benchmark to compare them with previous open-source GPU implementations.

Our computational artifact is a detailed benchmark (https://github.com/ZhangJingrong/gpu_topK_benchmark), which compares AIR Top-K and GridSelect with Sort, WarpSelect, BlockSelect, Bitonic Top-K, QuickSelect, BucketSelect, SampleSelect, and RadixSelect under three different kinds of data distributions: uniform, normal and radix-adversarial. Our AIR Top-K has already been open-sourced, and the code can be found in RAFT library (https://github.com/rapidsai/raft/blob/branch-23.04/cpp/include/raft/matrix/detail/select_radix.cuh).

Fig.6, Fig.7, and Table 2 in our work can be reproduced using this benchmark. Fig.6 and Fig.7 show the running time of our and previous algorithms for different combinations of N and K under the three distributions. Table 2 summarizes the speedup of our two algorithms over baseline methods. From this table and two figures, we should observe similar speedups mentioned in our paper.

REPRODUCIBILITY OF EXPERIMENTS

Our experiments are conducted with NVIDIA CUDA 12.0 on NVIDIA A100 GPU.

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) Download and compile required libraries: Run the "download.sh" in the directory "thrid_party". Follow the description in README.md to compile these libraries.

3) Generate the binary for the benchmark: In the directory "benchmark", run "make". A binary file named "benchmark" is generated.

4) Run the experiments and collect data for the table and figures: In the directory "script", run "bash run-k.sh" to collect data for Fig.6 and "bash run-n.sh" to collect data for Fig.7.

5) Use Python code to generate the table and figures: In the directory "script", run "bash exp.sh".

Running "bash run-k.sh" and "bash run-n.sh" takes about 8.38 hours and 8.65 hours, respectively.

After running "bash run-k.sh", we can get a file named "k-as-x.out.a100", which contains the running time for batch size 1, N equaling 2^15,2^20,2^25 and 2^30, and K varying from 2^3 to 2^20. After running "bash run-n.sh", we can acquire a file named "n-as-x.out.a100", which contains the running time for batch sizes 1 and 100, K equaling 2^5, 2^8, and 2^15, and N varying from 2^11 to 2^30.

With "exp.sh", we can get two figures and a CSV file in the directory "script": "k-as-x-a100.png" corresponds to Fig.6; "n-as-x-a100.png" corresponds to Fig.7; "speedup.csv" corresponds to Table 2.