Efficient Top-K Query Processing on Massively Parallel Hardware

Anil Shanbhag
MIT
anil@csail.mit.edu

Holger Pirk
Imperial College London
pirk@imperial.ac.uk

Samuel Madden
MIT
madden@csail.mit.edu

ABSTRACT

A common operation in many data analytics workloads is to find the top-k items, i.e., the largest or smallest operations according to some sort order (implemented via LIMIT or ORDER BY expressions in SQL). A naive implementation of top-k is to sort all of the items and then return the first k, but this does much more work than needed. Although efficient implementations for top-k have been explored on traditional multi-core processors, there has been no prior systematic study of top-k implementations on GPUs, despite open requests for such implementations in GPU-based frameworks like TensorFlow[1] and ArrayFire[2]. In this work, we present several top-k algorithms for GPUs, including a new algorithm based on bitonic sort called bitonic top-k. The bitonic top-k algorithm is up to a factor of 15x faster than sort and 4x faster than a variety of other possible implementations for values of k up to 256. We also develop a cost model to predict the performance of several of our algorithms, and show that it accurately predicts actual performance on modern GPUs.

CCS CONCEPTS

• **Information systems → Query operators**; • **Theory of computation → Massively parallel algorithms**; • **Computer systems organization → Heterogeneous (hybrid) systems**;

KEYWORDS

Top-K Algorithms for GPU; Bitonic Top-K

ACM Reference format:
Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient Top-K Query Processing on Massively Parallel Hardware. In *Proceedings of 2018 International Conference on Management of Data, Houston, TX, USA, June 10–15, 2018 (SIGMOD'18),* 14 pages.
https://doi.org/10.1145/3183713.3183735

1 INTRODUCTION

A common type of analytical SQL query involves running a *top-k*, i.e., finding the highest (or lowest) k of n tuples given a ranking function. Examples of top-k queries include asking for the most expensive products on an e-commerce site, the best-rated restaurants in a review site, or the worst performing queries in a query log. Top-k is a well studied problem in computer science in general and data management in particular since top-k calculation (order-by/limit clauses) is supported by virtually every data analytics system. There are many instances of the problem and a diversity of efficient solutions (see [13] for a survey).

Figure 1: The Duality of Top-K and Sorting

A naïve method for finding the top-k elements is to sort them and return the first k. However, sorting does more work than necessary, as there is no need to sort the elements beyond the top-k. A better approach is to maintain a priority-queue (a.k.a. max-heap) of size k and inserting greater elements while removing lesser ones. The runtime of this approach is in the order of $n \log(k)$. This algorithm can be parallelized across m processors by logically partitioning the data, having each processor compute a per-partition top-k and computing the global top-k from the m per-partition heaps. While this method can be efficiently implemented on multi-core processors (see Section 6.7), it is not suited to the Single-Instruction-Multiple-Threads execution model of massively parallel systems[3]. With the recent interest in GPU-based query processing [3, 12, 14, 16, 19, 23], there is an obvious need for a efficient, massively parallel algorithm to solve the top-k problem. In fact, we found that two of the most mainstream GPU programming frameworks (Tensorflow and Arrayfire) [1, 2] have open feature requests to add a top-k operator.

One way to develop an intuition for the existence and even the characteristics of a solution to this problem is to consider the duality of top-k and sorting algorithms. We illustrate this duality in Figure 1: the corresponding sort algorithm to priority queues is heapsort. In fact, one may view heapsort as the construction of a priority queue with $k = n$ and the subsequent extraction of the elements in sorted order. This, of course, hides many implementations details but helps to form an intuition. When thinking about sorting and top-k in the context of massively parallel architectures, one finds that the textbook massively parallel sorting algorithm is bitonic sorting. Yet, there is no known corresponding top-k algorithm to bitonic-sort. We can, however, hypothesize that, like bitonic sort, it

[1] https://github.com/tensorflow/tensorflow/issues/5719
[2] https://groups.google.com/d/topic/arrayfire-users/oDtQcI7afZQ

[3] the unpredictable execution flow leads to high branch divergence overhead

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183735

is likely to be based on bitonic merges and needs to incorporate a number of low-level optimizations to make it compute- as well as bandwidth-efficient.

In this work, we systematically develop this intuition into a working algorithm by extensively studying existing top-k solutions on GPUs and developing a novel solution targeted towards massively parallel architectures. We found that it is in fact based on bitonic merges and called it *bitonic top-k*. We investigate the characteristics of a number of other potential top-k algorithms for GPUs, including sorting and heap-based algorithms, as well as radix-based algorithms that use the high-order bits to find the top items. In the end, we find that bitonic top-k is up to 4 times faster than other top-k approaches and upto 15x faster than sorting for k up to 256.

These new algorithms have the potential to directly impact the performance of modern GPU-based database systems: all of the systems we are aware of (PG Strom, Ocelot, and MapD) currently use sort or transfer the entire dataset to the CPU for top-k calculation. They could, thus, directly obtain the benefits of our approach by integrating our algorithm. While we do not explicitly study means of mitigating the PCI-E bottleneck, having an efficient GPU-based top-k operator will allow these system to transfer less data through the PCI-E bus and, thus, achieve higher performance.

In summary, we make the following contributions:

- We study the performance characteristics of a variety of different top-k algorithms on a variety benchmarks, varying the data-set size, the value of k, the type of data (ints vs floats), and the initial distribution of data.
- We develop a novel, massively parallel, algorithm for the efficient evaluation of top-k queries.
- We devise a number of optimizations (in part based on known techniques, in part entirely novel) and show that our new bitonic top-k algorithm generally outperforms all other algorithms, often by a factor of $4x$ or more, for values of k up to 256. Furthermore, we demonstrate its robustness against skewed input data distributions.
- We demonstrate that the algorithm is able to be integrated into existing systems (specifically, MapD.)
- Finally, we develop detailed cost models for our bitonic top-k as well as other algorithms, and show that these cost models can accurately predict runtimes, which is valuable when a query planner needs to choose a top-k implementation.

Before describing the details of our algorithms, optimizations, and experiments, we begin with a discussion of GPU performance, and existing sorting and top-k algorithms on them.

2 BACKGROUND

2.1 GPU Data Access

Many database operations written on the GPU are still performance bound by the memory subsystem (shared or global memory) [23]. In order to characterize the performance of different algorithms on the GPU, it is, thus, critical to properly understand its memory hierarchy.

Figure 2 shows a simplified hierarchy of a modern GPU. The lowest and largest memory in the hierarchy is the global memory. The global memory is off-chip and has a memory bandwidth of 150-920 GBps on modern GPUs. Each GPU has a number of compute

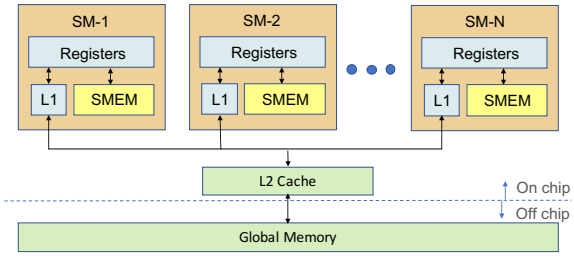


Figure 2: GPU Memory Hierarchy

units called Streaming Multiprocessors(SMs). Each SM has a number of cores and a fixed set of registers. Each SM also has a shared memory which can be accessed by all the cores and an L1 cache to cache requests to global memory. Accesses to global memory from the SM are cached in the L2 cache. The L2 cache is shared across all streaming multiprocessors(SM) and is on-chip.

Work on the GPU is done by large number of threads organised into thread blocks (each run by one SM). Thread blocks are further divided into warps (usually 32 threads). The threads of a warp execute in a Single Instruction Multiple Threads (SIMT) model. The device *coalesces* global memory loads and stores from a single warp. Maximum bandwidth can be achieved when warps coalesce access to global memory, resulting in neighboring locations being accessed.

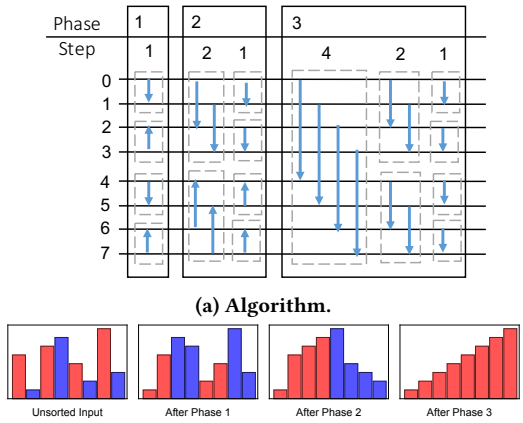
The programming model allows users to explicitly allocate global memory and shared memory in each thread block. Shared memory is an order of magnitude faster than global memory. On the Nvidia Titan X Maxwell GPU, the global memory bandwidth is around 250 GBps while the shared memory bandwidth is around 3 TBps. At the same time, GPUs only have a few MBs of shared memory spread across the SMs compared to GBs of global memory. To maximize performance, shared memory is organized into 32 banks, so that threads in a warp can access different memory banks in parallel. However, if two threads in a warp access the same memory bank, a *bank conflict* occurs, and accesses to the bank are serialized.

Finally, registers are the fastest layer of the memory hierarchy. If a thread block needs more registers than available, register values spill to thread-local memory. Despite its name, local memory only means it is only accessible by the thread – it is stored off the SM in slow global memory.

2.2 Sorting on the GPU

Many sorting algorithms have been proposed over the years. The early implementations were often based on bitonic sort [6, 10, 17]. Later, radix-based sort algorithms were proposed which perform better than bitonic sort [15, 21, 22].

Bitonic Sort Bitonic sort is based on bitonic sequences, i.e. concatenations of two subsequences sorted in opposite directions. Given a bitonic sequence S with length $l = 2^r$, S can be sorted ascending (or descending) in r steps. In the first step the pairs of elements $(S[0], S[l/2])$, $(S[1], S[l/2+1])$, ..., $(S[l/2-1], S[l-1])$ are compared and exchanged if the second element is smaller than the first element. This results in two bitonic sequences, $(S[0], ..S[l/2-1])$ and $(S[l/2], ...S[l-1])$ where all the elements in the first subsequence are smaller than any element in the second subsequence.



(a) Algorithm.

(b) Example Data (red bars sorted increasing, blue decreasing)

Figure 3: Bitonic Sorting Network

In the second step, the same procedure is applied to both the subsequences, resulting in four bitonic sequences. All elements in the first subsequence are smaller than any element in the second, all elements in the second subsequence are smaller than any element in the third and all elements in the third subsequence are smaller than any element in the fourth subsequence. The third, fourth, ..., r-th step follow similarly. Processing the r-th step results in 2^r subsequences of length 1, thus the sequence S is sorted.

Let A be the input array to sort and let $n = 2^k$ be the length of A. The process of sorting A consists of k phases. The subsequences of length 2 $(A[0], A[1])$, $(A[2], A[3])$, ..., $(A[n-2], A[n-1])$ are bitonic sequences by definition. In the first phase these subsequences are sorted (as described above) alternating ascending and descending. This creates bitonic subsequences of length 4, $(A[0], A[1], A[2], A[3])$, ..., $(A[n-4], A[n-3], A[n-2], A[n-1])$. In the second phase these subsequences of length 4 are sorted alternating ascending and descending, resulting in subsequences of length 8 being bitonic sequences. In the i-th phase of bitonic sort the total number of subsequences being sorted is 2^{k-i} and the length of each of these subsequences is 2^i, thus the i-th phase consists of i steps. After the (k-1)-th phase the array A is a bitonic sequence. A is sorted in the last phase k.

In every step $n/2$ compare/exchange operations are processed. There are $logn$ phases, with the i-th phase having i steps. Thus, the number of comparisons is $O(nlog^2n)$. Hence, bitonic sort is slower than other $O(nlogn)$ sort algorithms on a serial CPU. The advantage of bitonic sort is that it can be easily parallelized on SIMT and SIMD architectures and requires less inter-process communication. Figure 3a shows the bitonic sorting network for an arbitrary sequence of size 8. There $log_2 8 = 3$ phases, where phase i has i steps. Every step consists of $8/2 = 4$ comparisons. Sorting in Figure 3a(a) follows the process described above: in phase 1, elements 0 and 1 are compared and sorted in asscending order; elements 2 and 3 are sorted descending; elements 4 and 5 ascending, and so on. Each of these comparisons can be done in parallel on separate threads. At the end phase 1, there are 4 sorted sequences of length 2. In phase 2, with step size 2, first elements 0 and 2 and 1 and 3 are compared and sorted descending, while 4 and 6 and 5 and 7 are sorted ascending. These comparisons can also be done in parallel. Then, phase 2 with step size 1 is executed, such that elements 0 and 1 and 2 and 3 are sorted descending, and 4 and 5 and 6 and 7 are sorted ascending. Again these comparisons are parallelized. At the end of phase 2, we are left with two length 4 sorted lists. Finally, phase 3 merges these two lists using decreasing step sizes from 3 to 1.

The fastest implementation of bitonic sort is the one proposed by Peters et al.[17]. The bitonic top-k algorithm discussed later re-uses some of the ideas from their paper.

Radix Sort Radix sorting is based on the reinterpretation of a k-bit key as a sequence of d-bit digits, which are considered one a time. The basic idea is, that splitting the k-bit digits into smaller d-bit digits results in a small enough radix r = 2^d, such that keys can be partitioned into r distinct buckets. As sorting of each digit can be done with an effort that is linear in the number of keys n, the whole sorting process has a time complexity of $O(\lceil k/d \rceil n)$. Iterating over the keys' digits can be performed from the most-significant to the least significant digit (MSD radix sort [22]) or vice versa (LSD radix sort [15, 21]).

In either case, the first step is to compute a histogram of the input values in a sequential scan. As the histogram reflects the number of keys that shall be put into each of the r buckets, computing the exclusive prefix-sum over these counts yields the memory offsets for each of the buckets. Finally, the keys are scattered into the buckets according to their digit value. Recursively repeating these steps on subsequent digits for the resulting buckets ultimately yields the sorted sequence. The best performing sort algorithm today is based on MSD radix sort [22].

2.3 K-Selection

The k-selection problem asks one to find the k-th largest value in a list of n elements. Having a solution to the k-selection problem, one can easily find the top-k elements by possibly making one additional pass over the data. Alabi et.al [5] studied this problem extensively. Apart from the sort and choose the k-th element, they studied two other algorithms: Radix Select and Bucket Select.

Radix Select: Radix select follows from the MSD radix sort algorithm. Like the MSD radix sort, it operates as a sequence of steps, each of which processes a d-bit digit. It performs the same histogram and prefix sum steps. However, instead of writing out all the entries partitioned into buckets, radix select uses the histogram to find the bucket B containing the k^{th}-largest entry. It then writes out only the entries of B and continues to examine the next d-bit digit of the elements in *only* the matched bucket.

Bucket Select: Instead of creating the buckets based on radix bits, bucket select tries to be more robust by computing the buckets based on the min-max values. The algorithm makes an explicit first pass over the dataset to calculate the min and max values. Subsequently we execute a series of passes. Each pass is three step: create multiple buckets equally spaced out between min and max and, compute the number of entries in each bucket per thread. Second, do a prefix sum and find the bucket with the k^{th} largest element. Finally, read the input and write out elements of the matched bucket. We run the next pass on the entries of the matched bucket.

Algorithm 1: Per-Thread Top-K

Input : List L of length n; const int k
Output: List O of the top-k elements per thread

```
1 int t ← getGlobalThreadId();
2 int nt ← numThreads();
3 MinHeap heap;
4 for i ← t; i < n; i += nt do
5     T xi = L[i]; // T is the key type
6     if xi > heap.min() then
7         heap.pop(); heap.push(xi);
8 for j ← 0; j < k; j++ do
9     O[t + j*nt] = heap.pop();
```

3 ALGORITHMS

Based on the discussion so far, we have 3 algorithms to find top-k:

- Sort and Choose: Use radix sort to sort the entire vector and select the top-k elements from it.
- Using Radix Select: We can use the radix-based selection algorithm to get the k^{th} largest element and use that to find the top-k by making one additional pass over the input array.
- Using Bucket Select: We can do the same as above, this time using Bucket Select instead of Radix Select.

In this section, we describe two new algorithms for finding the top-k elements. In the first algorithm, each thread independently maintains the top-k elements it has seen so-far and finds the global top-k amongst the local (per thread) top-ks. Second, we present the bitonic top-k algorithm which is based on bitonic sorting. For ease of presentation, our description assumes tuples consisting only of a key. Of course, real applications may need to perform top-k on other settings, including (key,value) pairs, multiple keys, and different data types and distributions; our evaluation shows that our algorithms cover all of these cases (Section 6).

3.1 Per-Thread Top-K

A single-threaded version of top-k would maintain the top-k elements in a min-heap and update it for every new element seen. The natural way to parallelize is to partition the input, calculate the top-k per partition and calculate the global top-k from those as a final reduction step. Algorithm 1 shows the pseudocode that would run in parallel in each thread (`nt` threads are run in parallel). We use a min-heap per thread to maintain the top-k elements seen by that thread so far. After initializing the heap, we iterate over the elements starting from t in steps of number of threads. This (coalesced) memory access pattern has been shown to benefit memory access on the GPUs [11]. We check if the current element is larger than the minimum value among the top-k seen. If so, we pop the minimum and add the current element. Finally, we write out the top-k values to O in a coalesced manner. This approach is efficient in terms of memory usage. It makes one full read pass over the global memory and writes significantly less data. However, it suffers from thread divergence and occupancy issues, discussed in greater detail in Section 4.1.

Algorithm 2: Bitonic Top-K *Local Sort*

Input : List L of length n
Output: L with sorted sequences of length k

```
1 int t = getGlobalThreadId();
2 for len ← 1; len < k; len ← len ≪ 1 do
3     dir ← len ≪ 1;
4     for inc ← len; inc > 0; inc ← inc ≫ 1 do
5         int low ← t & (inc − 1);
6         int i ← (t ≪ 1) − low;
7         bool reverse ← ((dir & i) == 0);
8         x0, x1 ← L[i], L[i + inc];
9         bool swap ← reverse ⊕ (x0 < x1) ;
10        if swap: x0, x1 ← x1, x0;
11        L[i], L[i + inc] ← x0, x1;
```

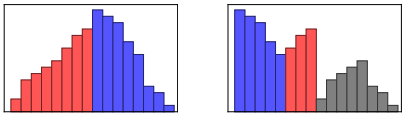
3.2 Bitonic Top-K

While a full bitonic sort is a solution to the top-k problem, it performs a significant amount of unnecessary work in sorting the entire input, just as heap sort is much less efficient than using a priority queue to select the top-k.

In bitonic sort, we start from an unsorted array which is equivalent to sorted sequences of length 1 and construct longer sorted sequences of length 2,4, ... up to n, at which point the entire list is sorted. Our basic approach is to develop an algorithm that performs as little unnecessary work as possible but maintains the massively parallel nature as bitonic sort. To achieve this, we decompose the complex bitonic sort operation into a series of parallel steps with different comparison distances. We carefully reassemble the steps into three operators that, in combination, allow the efficient fully parallel calculation of the top-k elements of a vector. These operators are *local (bitonic) sort*, *merge* and *rebuild*.

In *local sort*, we generate sorted sequences of size k using (partial) bitonic sort. In the *merge*, we bitonically merge two sorted sequences of size k, thus creating two bitonic sequences, where the first sequence contains the k greatest (w.l.o.g) and the second sequence contains the k least elements. In *rebuild* we sort the sequence containing the greatest (w.l.o.g.) elements; the second sequence containing the smaller k elements is discarded. While sorting, we exploit the fact that the output of the second operator already satisfies the bitonic property. At this point, we have effectively halved the problem size. We recursively apply the *merge* and *rebuild* operators to the (halved) sequence until we are left with only k elements which form the top-k. The resulting algorithm performs no unnecessary work and has the massive parallelism of bitonic sort. In the rest of this section, we describe the individual operators in more detail.

(1) Local Sort. The goal of this operator is to generate sorted runs of length k alternating between ascending and descending, starting from an unsorted array. Algorithm 2 shows the pseudocode. The unsorted sequence is equivalent to sorted sequence of length $len = 1$. Starting from $len = 1$, we generate sorted sequences of length $len = 2, 4 \ldots k$. When $len = k$, we are done. This is the outer loop on line 3. When $len = x$, two neighboring sorted sequences



(a) Before Top-K Merge (b) After Top-K Merge

Figure 4: Top-K Merge

Algorithm 3: Bitonic Top-K *Merge*

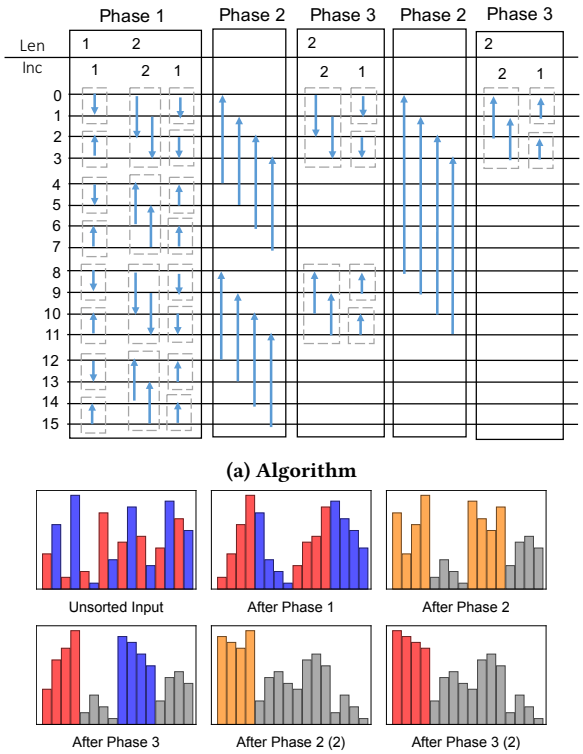
Input : List L with sorted sequences of length k
Output: List L_2 of size $|L|/2$ with bitonic sequences of length k

```
1 int t ← getGlobalThreadId();
2 int low ← t & (k-1);
3 int i ← (t ≪ 1) - low;
4 L2[t] ← max(L[i], L[i+k]);
```

of length x form a bitonic sequence of length $2x$ and can be sorted in $log(x) + 1$ steps. This is handled by the inner loop on line 4. In the first step, when $inc = len$, we compare pairs of elements $(L[0], L[len])$, $(L[1], L[1 + len])$, ... $(L[len - 1], L[2len - 1])$. This is done in parallel, and each thread compares one pair of elements. In general, thread t compares element $L[i]$ to $L[i + len]$ where the index i is calculated as a function of t and inc as shown in lines $5-6$. The elements are compared and exchanged (12-13) (if needed) and written back to the original array (14-15). The direction of exchange is determined by len. When $len = x$, we want to generate alternating ascending descending sorted sequences of length $2x$, i.e.: the direction changes every $dir = 2 * len$ elements (Line 3). The actual direction of comparison is determined by whether (i/dir) is odd or even (Line 7). *Phase 1* in Figure 5 illustrates the accesses of *Local Sort* operator to generate find the top-4 of 16 elements.

(2) Merge. At the end of the *local sort*, we have alternating ascending descending sorted (i.e., bitonic) sequences of length k. We compare neighboring sequences pair-wise and select the larger element in each pair. While we do not know how many elements of each of the sequences are selected, we know that the top-k elements were selected and that they form a bitonic sequence. This is *the key insight of our work*. To illustrate it, consider Figure 4 which illustrates the calculation of a top-8: in Figure 4b (after the merge step), all elements on the left are amongst the top-8 because they are greater than their comparison partner which implies that they are greater than all elements to the left (or right, respectively) of their comparison partner (due to the bitonic property). This step halves the top-k candidate set. Algorithm 3 shows the pseudocode.

(3) Rebuild. The input to *rebuild* is a list L with bitonic sequences of length k instead of an unsorted sequence in the *local sort* operator. As a result, we can generate sorted sequences of length k in $log(k)$ steps by applying the inner loop of the *local sort* starting with $len = k/2$. For completeness, Algorithm 4 in Appendix B shows the pseudocode. The flow is the same as in *local sort*. A combination of *merge* and *rebuild* reduces a list of length n with sorted sequences of length k to a list of length $n/2$ with sorted sequences of length k. *Merge* and *rebuild* are repeated till we have a list of length k.



(a) Algorithm

(b) Visualisation (gray: inactive, orange: candidates)

Figure 5: Bitonic Top-K (K=4)

Analysis. In *local sort*, every step does $n/2$ comparisons. There are $logk$ outer loop iterations, with the i-th one having i steps. In *merge*, we do $n/2$ comparisons. In *rebuild*, we have $logk$ steps of $n/2$ comparisons. Each time *merge* runs, the list size halves. *Merge* and *rebuild* run multiple times till we get a list of size k. The total number of comparisons are $O(nlog^2k)$. The runtime of bitonic top-k, like that of the bitonic sorting network is independent of the data distribution and depends only on $|L|$ and k.

4 OPTIMIZATION & IMPLEMENTATION

In this section, we describe a number of optimizations – at both logical and implementation levels – that we applied to the different methods to optimize performance. All the performance numbers in this section are from running algorithms on a dataset of 2^{29} floating point values generated from a uniform distribution $U(0, 1)$ on a Nvidia Titan X Maxwell GPU (see Section 6.1 for details about the hardware setup). Numbers on more diverse data are given in Section 6.

4.1 Per-Thread Top-K

To implement the per-thread top-k algorithm (Algorithm 1) efficiently, we use shared memory to store the heap. Each thread block allocates an array of size $k * wg$ in shared memory where wg is size of the thread block. Each thread maintains its own heap in shared memory using an array of size k. In order to avoid bank conflicts, we store the array striped, where thread t uses entries `sdata[t + wg*i]` where *sdata* is the shared memory array used by the thread

block, and, i varies from $0...k$. Since wg is always a multiple of 32, each thread's array maps to one shared memory bank and multiple threads in a warp updating their respective arrays does not cause shared memory bank conflicts.

The implementations suffer from two problems:

Thread Divergence: Heap updates are data dependent. On the GPU, a warp (32 threads) runs in a SIMT model. As a result, even if one thread wants to update its entries, all the other threads in the warp have to follow the same instruction path, leading to slowdown.

Occupancy: The shared memory used per thread increases with k. As the shared memory used by a block increases, the number of concurrent warps that can be run (occupancy) reduces. Beyond a point, the occupancy reduction leads to the GPU not having enough active warps to saturate the global memory bandwidth. For $k \geq 512$, even using the minimum thread block size of 32, we would need $64KB$ of shared memory, which is greater than $48KB$ available per thread block on our GPU.

We also implemented the per-thread top-k algorithm using registers and found its performance to be inferior. Appendix A contains a more detailed discussion and performance comparison of the register-based version.

4.2 Selection-based Top-K

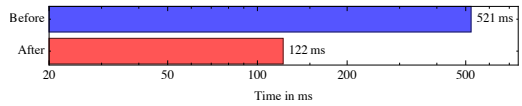
The radix select and bucket select implementations used come from the GGKS package [4]. We revised the implementation of radix select to use 8-bit digits (based on MSD radix sort [22]) instead of 4-bit digits in the original code. This results in 4 passes for 32 bit (int and float) keys. Each pass can reduce the data size. However, if after the prefix sum we see no data reduction, the clustering step is skipped and we simply re-use the input in the next pass. Bucket select also divides the data into 16 buckets at a time and selects one bucket containing the k-th element. The interested reader can refer to [5] for more implementation details. The radix select implementation would write out the entire input array after each pass and then update the array pointer to point to the bucket containing k^{th} element. We fixed this inefficiency to only write out the right bucket.

Given the k^{th} highest element X, we can make an additional pass over the data to find the top-k elements. However this is not necessary. Once we select the bucket containing X, when scanning the array the second time to write out the tuples that fall into the bucket, we can also write out the elements in the higher buckets to a separate *result* array. In the last pass, we copy over all the elements in the identified bucket with value less than X to *result* and pad *result* with X to make it of size k. This eliminates the last pass we previously had to find the top-k elements given X.

4.3 Optimizing Bitonic Top-K

In this section we discuss a number of optimizations we devised to achieve close-to-optimal performance with our new bitonic top-k algorithm. While some of these are inspired by similar optimizations for other algorithms, to the best of our knowledge, none of them are applied in the context of top-k calculation. However, since our optimizations may be applicable to other problems, we include a paragraph on novelty and applicability in the description of each optimization. To give an impression of the importance of each optimization, we end every subsection with a graph indicating the the effect of optimization on the runtime for the case of finding top-32 elements in the dataset described at the start of this section.

Operating in Shared Memory. The first optimization can be applied to each of the three operators individually: instead of reading/writing data after each massively parallel step to global memory, we do it once per operation. The data required is loaded into shared memory at the beginning of the operation. All the operation's intermediate steps happen in shared memory. At the end, the result is written back to global memory. For example, the *local sort* operation in Figure 5a has 3 intermediate steps. With this optimization, each threadblock would read the required data to shared memory, run the 3 steps within shared memory and then write back results at the end. Recall that the shared memory is an order of magnitude faster than global memory. This optimization shifts global memory reads/writes to shared memory reads/writes, thereby improving performance.



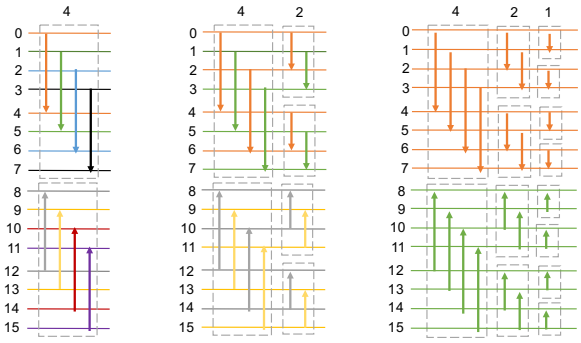
This results in a significant performance improvement from 521ms to 122ms. The *local sort* operator becomes shared memory bound while the *merge* and *rebuild* are still global memory bound.

Note that this optimization is contingent on k being less than or equal to 2∗max thread block size (= 2048 on modern GPUs). It also cannot be applied to all steps of a general bitonic sort algorithm with steps with *inc* up to $n/2$, because this would require loading the entire array into shared memory, but this is not a limitation in our bitonic top-k alogorithm as long as k is small enough. This optimization has been applied to bitonic sort to minimize accesses to the global memory [17].

Merging Operators. As discussed in Section 3.2, our bitonic top-k algorithm can be broken into three operations: (1) *local sort* to create sorted sequences of length k, (2) *merge* two sorted sequences of length k to create a bitonic sequence of length k and (3) *rebuild* a bitonic sequence of length k after a merge. While the *local sort* operation is only executed once in the beginning, the *merge* and *rebuild* phase are alternated until the result is found.

The naive implementation would run a kernel per operator. However, there are no cross thread block dependencies across each of the kernels. This leads to a significant optimization potential: multiple operators can be fused into a single kernel and shared memory can be used to communicate results between operators. In addition to reducing kernel invocation overhead, this optimization eliminates global memory traffic due to intermediate results.

Each *merge* halves the number of elements. In order to ensure that each thread in the last operation in the fused kernel has work to do, we need to ensure that number of data items per thread is atleast 2^x where x is the number of *merge* phases in the fused kernel. We found the optimal number of processed of data items per thread to be 8. Beyond that, doubling the number of elements per thread doubles the number of shared memory bank conflicts and yields no performance improvement. Since each *merge* halves the number of elements per thread, processing 8 elements per thread allows us to have three (i.e., $ld(8)$) *merge* phases per kernel. This leads to two separate kernels: the first performing *local sort* followed by two

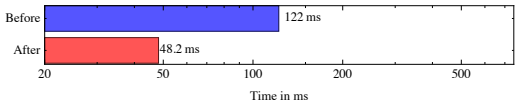


(a) Single step (b) Combining 2 Steps (c) Combining 3 Steps

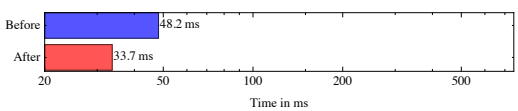
Figure 6: Combining Multiple Steps

merge-rebuild operators and a single *merge* (**SortReducer**). The second kernel performs three *rebuild-merge* operator sequences (**BitonicReducer**). To the best of our knowledge, this is a novel optimization.

This optimization reduces the runtime of top-32 from 122ms to 48.15ms. Both kernels (and as a result the entire application) are now shared memory bandwidth bound. The SortReducer kernel and BitonicReducer kernel achieve shared memory bandwidth of 2.75TBps and 2.7TBps respectively. This is greater than 90% utilization of the 2.9TBps peak bandwidth of shared memory observed on repeated read workload. We, therefore, shift our attention towards optimizing shared memory accesses.



Combining/Sequentializing Multiple Steps. For the next optimization, we rearrange the assignment of data items to threads to reduce the amount of memory traffic. Figure 6a shows the default assignment (threads are color-coded), each thread reads two values from shared memory, compares them and writes them back to shared memory. As each thread is responsible for 8 elements, it does the same for 3 other pairs. If, however, we process more than two values per thread per round, the read and write operations can be shared. In Figure 6b, e.g., the orange thread reads the values at positions 0, 2, 4 and 6 and performs two comparisons on each. This halves the shared memory traffic and can be generalized to more elements (see Figure 6c). While this (partially) serializes the processing (from three fully parallel steps with four operations each to 12 sequential operations) it does not increase the overall number of comparisons. This optimization is similar to optimization 1 which combines multiple steps that read and write to global memory to read and write to shared memory. Instead here, we combine multiple steps that read and write to shared memory to work in registers. This reduces the runtime to 33.7ms.



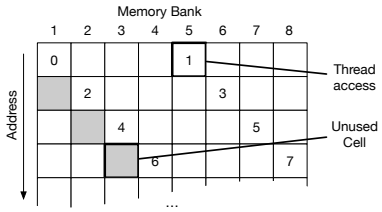
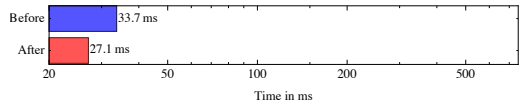


Figure 7: Avoiding Bank Conflicts with Padding (numbers in boxes designate the accessing thread's ID)

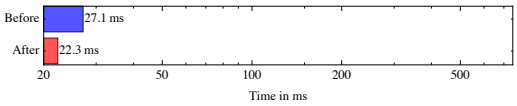
Do work before writing. Conventional wisdom is to copy a chunk of data from global to shared memory in a coalesced manner and processing data only in shared memory. However, by rejecting this common wisdom, we can reduce shared memory accesses. Each thread loads 8 consecutive elements from global memory into registers, perform all intermediate steps required to create local sorted sequence of length 8 without hitting shared memory and then write to shared memory. Note that as a result of this optimization, accesses to global memory are no longer coalesced because threads access data elements at a stride of 8. However, this does not lead to any noticeable performance difference on modern GPUs due to their data caches. This optimization is likely to be widely applicable. In our experiments, it yields an effective reduction of the runtime to 27.1ms.



Breaking Conflicts with Padding. In this and the following subsection, we introduce three optimizations that help avoiding memory bank conflicts. While most current GPUs have 32 shared memory banks and warps of 32 threads, illustrating the effects of our optimizations on 32 memory banks would unnecessarily inflate the size of our figures. For that reason, we assume 8 memory banks (and warps of size 8) for the illustrations (note that the experiments are conducted on a real GPU with 32 memory banks).

The first optimization is an instance of a widely known technique: padding arrays to avoid memory conflicts. A shared memory array of size n can be viewed as a 2D array of dimensions $[\frac{n}{8}, 8]$ (where 8 is the number of banks). The key idea is to allocate slightly more memory to create a larger array of dimensions $[\frac{n}{8}, 9]$. The extra column added does not store any elements, however, it helps break shared memory conflicts. Figure 7 shows the accesses performed by a combined step combining $inc = 2, 1$ at time step 0 after padding. The grayed out cells do not hold any values and are simply space overhead. Each thread wants to read 4 contiguous elements. Thread 0 wants to read entries 0-3, thread 2 wants to read 8-12. Without padding these two threads would conflict (0 and 8 are in the same bank). The figure illustrates how the padding prevents the conflicts (thread 0 and 2, access different memory banks after padding). This decreases the runtime of top-32 to 22.3ms. Note that padding does not help bitonic sorting due to its global memory bandwidth boundness. In contrast, the bitonic top-k is shared memory bandwidth bound.

Padding also has a second benefit: it allows us to merge more operators into a kernel. Recall that processing more than 8 elements



caused conflicts (discussed with operator merging earlier in this section) and that this effect limited us to merge only three operators.

With padding, this is no longer true which allows us to merge four or even more steps (processing 16 or more elements per thread). However, beyond 16, the number of allocated registers forces the compiler to reduce occupancy leading to a performance penalty: Figure 8 shows the performance when varying the number of processed elements (B). There is virtually no benefit when increasing B from 16 to 32 and a detriment when increasing B to 64. We, thus, fixed B to 16.

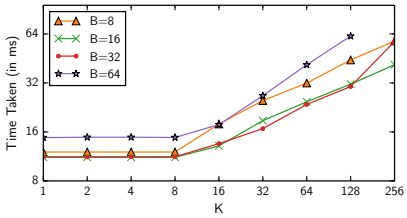


Figure 8: Varying the number of elements per thread

Chunk Permutation. Figure 9 illustrates the shared memory access pattern of the local sort operation after applying the optimizations discussed so far. Here, each outlined shape represents an operation with no accesses to shared memory (shared memory access is performed at the edges of each shape), the axes represent iterations of sequential loops within the kernel, and the numbers the distance in the input array of the compared elements. While most of the kernels are bank-conflict-free, we observe that, when the comparison distance is four, the memory accesses cause bank conflicts. To illustrate this, consider Figure 10a: it illustrates the comparisons that are performed in the red box in Figure 9 (a pairwise comparison of elements with a distance of four). The figure indicates the memory accesses of the threads in a warp: each arrow represents the comparison performed in one thread with the colors indicating the time (and thus the order) of the accesses. We observe that, despite padding, the memory accesses at clock time 0 overlap with respect to their memory bank. We can avoid this by changing the memory locations each thread reads from (and writes to). We call this optimization *Chunk Permutation* and illustrate it in Figure 10b: instead of reading from conflicting banks at clock 0, each thread accesses a different memory bank. One may notice that there is still overlap between the accessed values. However, these accesses are performed at different times. As is obvious the figure, there are no conflicts by observing that there are no two identically colored boxes in a column.

While we illustrated the chunk permutation optimization using the example of the last step in the local sort, the problem occurs whenever the comparison distance of a combined step is greater than 1. This makes it widely applicable for our case and even more broadly (e.g., for bitonic sort). For our application it removes all the remaining memory bank conflicts in the *local sort* operator for all $k \leq 256$ and improves the performance of top-32 from 17.8ms to 16ms. The effect is more pronounced at higher k, e.g., improving top-128 performance by roughly 20 percent. This optimization is novel.

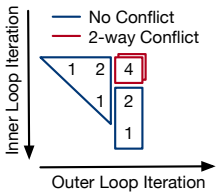
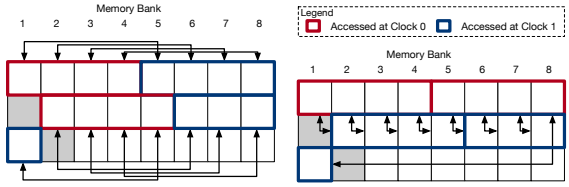


Figure 9: Comparison distance for local Sort $k = 8, x = 4$



(a) W/o Chunk Permutation **(b) W/ Chunk Permutation**

Figure 10: Bank-conflicts when comparing elements

The broader idea of re-arranging chunks to avoid bank conflicts could be applied to other algorithms that suffer from shared memory bank conflicts.

Reassigning Partitions. The last optimization we developed is targets the assignment of data items to threads after the first reduction: since the reduction halves the number of elements but the number of threads remains the same, there is less work per thread. This leads to fewer steps being merged because the number of steps that can be merged is the logarithm of the number of input data items per thread. To maintain the same number of input data items per thread after the reduction, we have half the threads perform all the work. While this leaves half of the threads without work, the reduction in shared memory traffic due to larger combined steps outweighs that cost. This optimization further improves the performance to 15.4ms. This optimization is novel and maybe applicable to kernels that reduce input data in phases.

Discussion:

Memory Usage. Memory usage is of critical importance for GPU-based data management systems. For a dataset of size n, out-of-place bitonic top-k uses one additional buffer of size $n/8$. This is significantly less than sort and selection-based methods which require an additional buffer of size n.

Data larger than GPU memory. When data is larger than can fit in GPU memory, data needs to moved to the GPU via the PCI bus. There is a significant amount of research on reducing pressure on that bottleneck using asynchronous transfers [10, 22], approximation [18], compression [9, 20] and cost-aware device selection [7]. While we do not explicitly address the PCI-bottleneck in this paper, the reductive nature of top-k queries makes it trivial to process the data in memory-size chunks and overlap computation with transfer (similar what is done for sorting [22]).

Bitonic Top-K on CPU. The bitonic top-k algorithm described can also be implemented on the CPU. We describe our implementation in Appendix C. Bitonic top-k after applying all the optimization described in this section comes close to being compute bound on the GPU. When the same algorithm is run on the CPU, it is strictly compute bound due to lower compute to bandwidth ratio of CPU.

5 DATABASE INTEGRATION

Having developed a highly optimized massively parallel top-k implementation, we were naturally interested in its usability in a full system. As a proof of concept, we integrated the bitonic top-k kernel into MapD, an open source GPU database [16]. In this section, we discuss two optimization opportunities that can we used in the context of database analytics to improve performance.

Fusing with filter. A common query template is to find the top-k items in a subset of the data satisfying a selection predicate. The easy way to execute this is to have a seperate kernel execute the filter and have the subsequent top-k kernel use the output to find the top-k items. GPU-based databases end up doing this currently as they treat the top-k kernel(done using sort) as a blackbox. We can optimize this by fusing the select into the bitonic top-k routine.

Each thread block running the SortReducer kernel reads in $16nt$ elements and writes out nt elements where nt is the number of threads in the thread block. One way to fuse the kernels is to read in $16nt$ elements, apply the filter predicate and run the SortReducer on the matched elements. However, the SortReducer kernel is then effectively running on $s * 16nt$ where s is the selectivity. As shown in previous section, having 16 elements per thread is crucial to the performance of SortReducer as it enables it to run combined steps. The FusedSortReducer instead uses the selection step as a buffer filler. It reads in nt elements at a time into shared memory, applies the filter predicate to find the number of matches elements, computes a prefix sum and then writes it out into a shared memory buffer of size $16nt$. It then reads in the next batch of nt elements till we have more than $15nt$ elements matched. The rest of the entries are padded with min/max value so that they never show up in the top-k results. The SortReducer then works on the buffer of $16nt$ elements and writes out nt elements contain the top-k.

Custom Ranking Function A custom ranking function is an order by clause of the form $f(A_1, A_2, A_3..)$ where f is any function and $A_1, A_2, ..$ are columns of A. The ranking function can be evaluated at the start of SortReducer kernel instead of running it as a separate project step which outputs the value of the function.

6 EVALUATION

In this section, we compare the performance of the five different algorithms we presented in Section 3:

(1) `Sort`: Sorting to find top-k
(2) `PerThread TopK`: Using a heap per thread to find top-k
(3) `Radix Select`: Adapting radix select to find top-k
(4) `Bucket Select`: Adapting bucket select to find top-k
(5) `Bitonic TopK`: Using the bitonic top-k algorithm

after applying the optimizations in Section 4 varying the following parameters: (1) the value of K (2) the key data type (3) the data distribution (4) the data size (5) the number of key and value columns and finally (6) the device (CPU vs. GPU). After that, we show the performance achieved by integrating `BitonicTopK` in the MapD database by evaluating top-k queries on a twitter dataset.

6.1 Setup

All the results are averages of 3 runs on a single socket Intel i7-6900 @ 3.20GHz (Skylake with 8 Cores, 16 hardware threads) with Nvidia GTX Titan X Maxwell GPU running on Ubuntu 15.10 (Kernel 4.2.0-30) and CUDA 8.0.

6.2 Performance with Varying K

We generate 2^{29} random uniformly distributed ($U(0, 1)$) floats and observe the performance of the different algorithms with K varying from 1 to 1024 in powers of 2. Figure 11a shows the results.

`Memory Bandwidth` shows the time taken to read the entire data from global memory. Since all of the data needs to be read at-least once, this constitutes a lower bound on the runtime of any algorithm. In reality, most algorithms would write/read intermediate data and have other overheads. We observe that the runtime of the `Sort` method is virtually constant across k since it has to sort the entire input irrespective of K.

`Radix Select` and `Bucket Select` take almost the same time across K as expected. The latter does worse than the former due to the use of more expensive atomic operations. When $k = 1$, `Bucket Select` is fast as it terminates after finding the min-max of the array and directly returns it as the result.

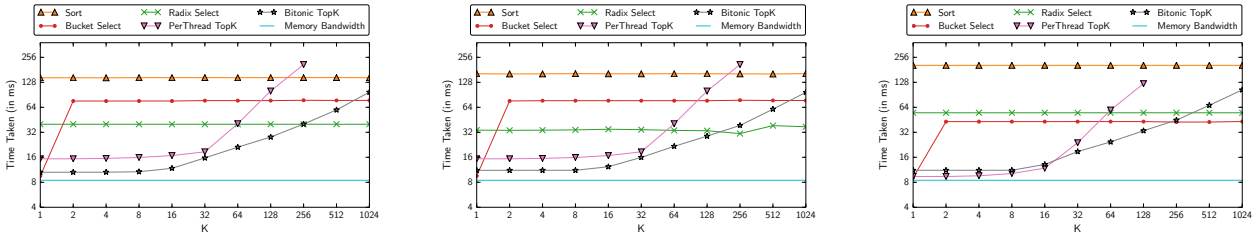
`PerThread TopK` line has steep slope rising from $k = 32$, this is due to reduced occupancy and thread divergence as explained earlier in Section 4.1. The approach fails for $K > 256$ due to the required amount of shared memory. For $K = 512$, even with the minimum thread block size 32, we need $512 * 32 * 4 = 64KB$ (each key is 4 bytes) which exceeds the available $48KB$ per thread block.

Finally, `Bitonic` does better than all the other algorithms for $K \leq 256$. For $K > 256$, the `Radix Select` method does better.

6.3 Dependence on Data Type

Next, we run the algorithms on a dataset with 2^{29} unsigned integers drawn from $U(0, 2^{31} - 1)$ (see Figure 11b). The time taken by all methods except `Radix Select` is virtualy identical to that observed with $float$ data type. `Radix Select` does better because with uniformly distributed data, the number of eliminated tuples per scan is maximal (a reduction of 256× assuming 8-bit radices).

Second, we run the algorithms on 2^{28} doubles drawn from $U(0, 1)$. The size of the data is the same, however the word size of each key has increased. Figure 11c shows the results. The `Sort`-based approach has to perform twice as many scans (since the number of digits has doubled) but scan fewer values. However, processing 64-bit values is significantly more expensive than 32-bit values on most GPUs which explains the cost increase. `Radix Select` has the same issue, however, this effect is less pronounced as the algorithm operates on a smaller number of elements in subsequent passes. `Bucket Select` ends up being slightly faster than with floats as the number of keys has reduced resulting in smaller number of atomic operations. The `PerThread TopK` line is similar to line seen with $float$ shifted to the left and slightly lower: this is natural since there less processing needs to be performed for every read byte. For each K, the method uses twice as much shared memory when processing doubles compared to processing floats. Thus, the approach fails earlier (for $K > 128$). Finally, `Bitonic TopK` remains largely unchanged as the data size is the same and the cost are dominated by the memory bandwidth.



(a) With float keys (b) With unsigned integers keys (c) With double keys

Figure 11: Time taken with different k

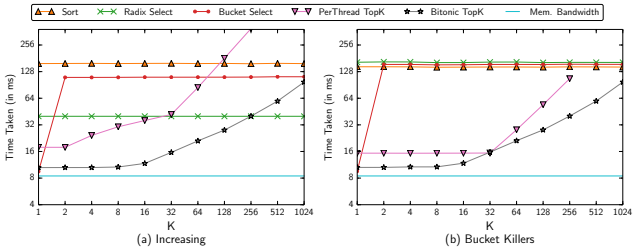


Figure 12: Performance across different distribution

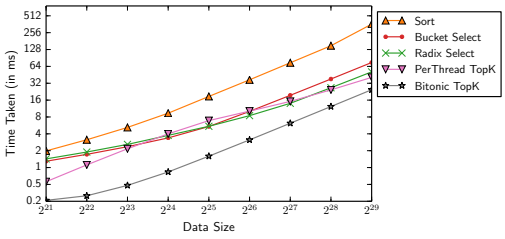


Figure 13: Performance with varying size

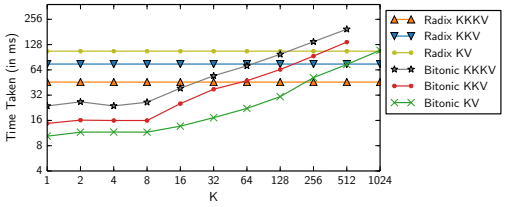


Figure 14: Different number of keys

6.4 Dependence on Data Distribution

Keeping the data size fixed at 2^{29}, we examine the performance of algorithms with varying k on 2 distributions:

- *Increasing*: Sorted floating point numbers from $U(0, 1)$
- *Bucket Killer*: Contains all 1s(floats) except 4 numbers, each of which differ from 1.0 in one 8-bit digit. This minimizes the reduction achieved in a single radix-scan.

Figure 12 shows the results. The only algorithms that do not change based on the distribution of elements are `Sort` and `Bitonic TopK`. Both perform precisely the same operations.

The *increasing* (figure 12(a)) distribution leads to `PerThread TopK` performing up to $3x$ worse while the other algorithms see no change. This is because `PerThread TopK`'s performance is dependent on number of heap inserts. With increasing distribution, each element causes a heap insert making it a near worst case for the algorithm.

For most selection algorithms, it is relatively easy to identify distributions which will cause worst case behaviour for the algorithms. *Bucket killer* is the adversarial distribution for `Radix Select`. With *bucket killer* (figure 12(b)), `Radix Select` ends up taking the same time as `Sort` because each radix pass leads to only one number being removed from consideration (the one which differs from 1 at that 8-bit digit). Each pass ends up reading and writing the entire dataset like in `Sort`. `Bucket Select` also experiences a $2x$ slowdown due to less data reduction in the intermediate steps. Note that, due to the predictable pattern of the bitonic merges, there is no adversarial input distribution for the `Bitonic TopK` approach making it a very robust option.

6.5 Dependence on Data Size

To show the performance of the algorithms across different data sizes, we run them with a fixed $k = 64$ and choose a data set of random floats drawn from $U(0, 1)$ with varying data sizes ranging from 2^{21} to 2^{29}. Figure 13 shows the results. `Bitonic TopK` and `Sort` grow linearly with input size. `PerThread TopK` maintains top-k per thread and runs a fixed number of threads to keep all the GPU cores busy. With larger data sizes, the number of elements processed by each thread increases. Also, for uniform distribution the probability of a heap insert decreases as more and more data is seen. This results in the initial outward bulge. `Radix Select` and `Bucket Select` grow linearly for larger data sizes. At data sizes below 2^{24}, the time taken by prefix sum (which is a constant across data sizes) becomes significant leading to flattening of the lines.

6.6 Key(s)+Value

So far, we used tuples with just a key. However, many applications would require key+value or multiple keys+value. In this section, we show the performance of `Radix Select` and `Bitonic TopK` with key + value (KV), two keys + value (KKV) and, three keys + value (KKKV). Each key is a float drawn from $U(0, 1)$ and value is a 4 byte integer. Size of the elements in the dataset is 2^{28}. Figure 14 shows the results. Both the methods show a linear increase in the runtime due to increased data sizes as we go from KV to KKKV. The cut-off point remains the same across the different key counts. We do not show the results for the other methods for readability.

We do not show experiments with larger value payloads as it is always better to pass around the tuple id and construct the full tuple at the end of top-k. For example, consider a dataset with 10 million tuples of 4 byte key, 12 byte payload. Running top-k on (key,id) instead of (key,payload) halves the data size moving around. Assembling the result at the end takes virtually no time.

6.7 Comparison against CPU

In this section, we compare the performance of CPU-based top-k to the GPU-based top-k. For CPU-based top-k, we have two heap-based methods: one using C++ STL priority queue as a min-heap (`STL PQ`) and, second a hand-optimized min-heap (`Hand PQ`). For each element, we check it against the heap minimum by comparing with the root of the heap. If its greater, we pop the root (the minimum) and insert the new element. We also show the CPU version of bitonic top-k. For GPU-based top-k, we show `Bitonic TopK` and `Radix Select`.

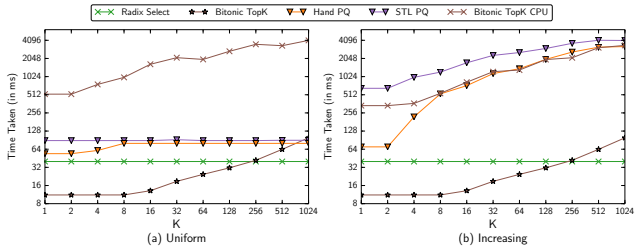


Figure 15: Comparing against CPU Top-K

First, we compare them on a dataset of 2^{29} floats drawn from uniform distribution $U(0, 1)$. Figure 15(a) shows the results. As the data is uniformly distributed, most of the elements get discarded when checked against the heap minimum and very few trigger a heap insertion. To illustrate this note that, for this dataset, with $k = 32$, each core looks at $671k$ elements and ends up doing about 500 insertions (including the first 32 elements that always get inserted). The performance is, thus, likely to be memory bound. `Bitonic TopK` does $3x$ better than `Hand PQ` when $k = 32$. Bitonic top-k on the CPU does signficantly worse than heap-based methods as it does significantly more computation than heap-based methods which do just 500 insertions.

Next, we consider the same dataset but sorted in increasing order. Figure 15(b) shows the results. Since the data is sorted, each element causes a heap pop/insert. This is close to the worst case. `Bitonic TopK` and `Radix Select` take the same time while the CPU algorithms do significantly worse. `Bitonic TopK` does $60x$ better than `Hand PQ` and $120x$ better than `STL PQ` when $k = 32$. Time taken by bitonic top-k on the CPU is close to that of `Hand PQ` despite doing more comparisons. This is due to the use of SIMD instructions.

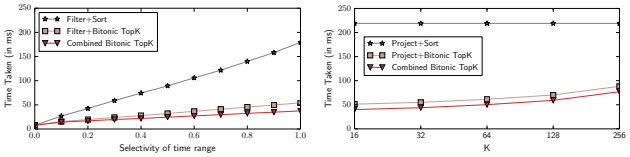
As empirically demonstrated in this section, `Bitonic TopK` is the best performing approach for smaller K ($K \leq 256$) and `Radix Select` for larger K. To provide an analytical argument in support of these findings and to predict the performance on different hardware, we develop a hardware-conscious cost model in Section 7.

6.8 MapD Integration

To evaluate the performance improvement got from `Bitonic TopK` in a real world setting, we evaluate the system on a twitter dataset consisting of 250 million tweets from May 2017. We evaluate four queries:

1) `SELECT id FROM tweets WHERE tweet_time < X ORDER BY retweet_count DESC LIMIT 50`

The query finds the top 50 most retweeted tweets in a specified time



(a) Get top-k most retweeted tweets in a time range **(b) Find top-k most popular tweets**

Figure 16: MapD Experiments

range. We vary the time range to have selectivity from 0 to 1 in steps of 0.1. MapD by default runs the filter on the time range followed by sort on the GPU. It then copies the top-k tweet ids and assembles the tweet (`Filter+Sort`). We evaluate two alternatives: 1) replace the sort by bitonic top-k (`Filter+Bitonic TopK`), 2) combined kernel that runs filter and bitonic top-k together (`Combined Bitonic TopK`). Figure 16a shows the results. Bitonic top-k based methods out-perform the existing methods. The filter fusion optimization saves the time to write out to and read in from global memory of the filtered id,retweet count entries. At selectivity 1, the filter fusion optimization reduces the total kernel runtime (time spent on GPU) by 30% and the end-to-end runtime by 23%.

2) `SELECT id FROM tweets ORDER BY retweet_count + 0.5 * likes_count DESC LIMIT K`

The query finds the most popular tweets based on a complex ranking function. MapD by default runs a projection step that computes the value of the ranking function followed by a sort step (`Project+Sort`). We evaluate two alternatives: 1) replace sort with bitonic top-k (`Project+Bitonic TopK`), 2) a combined kernel that computes the value of the ranking function inside the SortReducer (`Combined Bitonic TopK`). Figure 16b shows the results. The combined kernel saves on having to having to write out and read in the projected rank value. This reduces the runtime of the combined method by 10ms compared to `Project+Bitonic TopK`.

3) `SELECT id FROM tweets WHERE lang='en' OR lang='es' ORDER BY retweet_count DESC LIMIT K`

The query finds the top K tweets by retweet count in english or spanish language. We evaluate the same 3 methods used in query (1). The filter has a set selectivity of around 80%. We see the same trend as in the previous query. The combined kernel saves on having to read/write filtered id,retweet count entries. This reduces the runtime by 16ms compared to `Filter+Bitonic TopK` across all K.

4) `SELECT uid, COUNT() AS num_tweets FROM tweets GROUP BY uid ORDER BY num_tweets DESC LIMIT 50` The query finds the top 50 users by tweet count. There are about 57 million unique users in the dataset. By default in MapD, the query execution takes 97ms of which the sort step takes 44ms. Using bitonic top-k reduces the runtime by 39% as it reduces the time taken by the sort step by 38ms. A query which finds say the 50 most popular hash tags would not benefit as much from bitonic top-k as the most of the time is spent in the group by step.

7 COST MODEL

Due to space constraints, we limit our modeling efforts to the two best-performing algorithms (see last section): `Radix Select` and

`Bitonic TopK`. We model them using hardware parameters we determined empirically using benchmarks provided by Nvidia. The parameters are (1) the global memory bandwidth (B_G), (2) the shared memory bandwidth (B_S), (3) the key size in bytes (w), (4) the the input data size in bytes (D) and (5) the total number of threads (n_t).

7.1 Radix-based Top-K

Radix-based top-k (`Radix Select`) operates as a series of passes, each pass looking at one digit of 8 bits. Each pass reduces the data size and the total number of passes is at most $w/8$. Pass i involves:

- Read the input for the pass from global memory to write out the number of entries per digit value per thread (total: 16 integers per thread). D_{iI} is the input size for the pass in bytes. $D_{iI} = D$ for the first pass.
$$T_{i1} = \frac{D_{iI}}{B_G} + \frac{16 * 4 * n_t}{B_G}$$
- Calculate the prefix sum to find the digit value d containing the k-th value.
$$T_{i2} = \frac{2 * 16 * 4 * n_t}{B_G}$$
- Scan the input and write out entries with digit value d to another array in global memory. Let η_i be the fraction of entries with digit value d. Note that this step is skipped if $\eta_i = 1$.
$$T_{i3} = \frac{D_{iI}}{B_G} + \eta_i \frac{D_{iI}}{B_G}$$

The total time of pass i is $T_i = T_{i1} + T_{i2} + T_{i3}$. The total cost is the sum of the time taken by the individual passes.

7.2 Bitonic Top-K

Bitonic top-k runs a sequence of kernels: first the `SortReducer` kernel, followed by a series of `BitonicReducer` kernels. Let x be the number of elements per thread. Each kernel reduces the problem size by a factor of x. For every kernel, there are two components that can dominate performance depending on K: global memory access or shared memory access. Due to the high parallelism and the low overhead of context switches, the GPU will effectively hide the cost of the less expensive of these two behind the more expensive. The cost is, thus, the maximum of the two.

We start with the global memory access cost of the `SortReducer` kernel. The kernel makes one scan of the input from global memory and writes out $1/x$ of the input back (to global memory). The global memory data access time thus straight forward to model:

$$T_g = \frac{D}{B_G} + \frac{1}{x}\frac{D}{B_G}$$

The shared memory data access time is harder to estimate: in addition to the number of accesses, we need to take the number of shared memory bank conflicts into account. Since bank conflicts occur whenever two values on the same bank are accessed, we need to take the specific addresses of memory accesses into account.

The time taken if the kernel is bound by shared memory bandwidth is the sum of the time taken by each combined step:

$$T_s = \Sigma_i \delta_i \frac{D_{Ii} + D_{Oi}}{B_s}$$

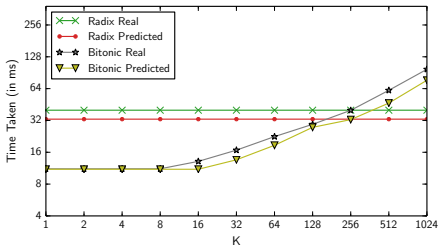


Figure 17: Estimated vs real runtimes

where δ_i is the number of shared memory bank conflicts for one warp and, D_{Ii} and D_{Oi} are size of data read and written by the phase respectively. Applying this to find T_s for SortReducer finding the top-32, we get $T_s = 17.5D/B_s$.

The estimated time taken by the *SortReducer* kernel is $max(T_g, T_s)$. For the Titan X Maxwell, $B_S = 2.9TBps$ and $B_G = 251GBps$. The estimated total time is $max(8.96ms, 12.1ms) = 12.1ms$ which is close to the actual runtime of $14.2ms$. The cost for `BitonicReducer` can be estimated in a very similar way except that it directly starts with $len = k/2$.

Figure 17 compares the actual time of the methods versus the predicted time based on the models for finding top-k on a dataset with 2^{29} floating point numbers drawn from $U(0, 1)$ with varying K. The predicted times show the same trends as the observed times and the cutoff point remains the same. Both the models underestimate the time taken. This is because a kernel bound by global or shared memory may not achieve the maximum possible bandwidth. For example, the first kernel of radix-based top-k should take $8.6ms$ based on model while in reality it takes $9.8ms$ and, the effective shared memory bandwidth used by the *SortReducer* kernel for $k = 32$ is around $2.5TBps$ versus the maximum $2.9TBps$.

As demonstrated in this section, bitonic top-k is not only experimentally faster but also theoretically more efficient than the best alternative we evaluated.

8 CONCLUSION

Data analytics on GPUs is increasingly common, and a frequently analytics task is to rank a set of data items according to some attribute and extract the top-k values. In this paper, we presented many algorithms to efficiently compute top-k on GPUs, including a new algorithm based on bitonic sort. Through an extensive performance evaluation of a number of different algorithms, we showed that our bitonic-top-k algorithm is an order of magnitude faster than the fastest algorithms based on fully sorting a list of elements, and, depending on the value of k, several times faster than several other algorithms for efficiently computing top-k. We also presented a cost model that accurately predicts the performance of several algorithms with respect to k, allowing a query optimizer to choose the best top-k implementation for a particular query.

We believe there is still room for innovation in this space. While we have intensively studied existing and proposed novel algorithms, we have consciously excluded hybrid and adaptive solutions from the scope of this paper. Such hybrid solutions could either involve multiple devices (CPUs and GPUs) as well as hybrids of the presented algorithms.

REFERENCES

[1] [n. d.]. Arrayfire discussion on top-k. http://bit.ly/2lLuFS1. ([n. d.]).
[2] [n. d.]. Issue to add gpu verion of top-k to tensorflow. https://github.com/tensorflow/tensorflow/issues/5719. ([n. d.]).
[3] Martín Abadi et al. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*.
[4] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. 2010. GGKS: Grinnell GPU k-selection. http://code.google.com/p/ggks/. (2010).
[5] Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. 2012. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)* (2012).
[6] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the spring joint computer conference*.
[7] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *SIGMOD*. ACM.
[8] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB* (2008).
[9] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *PVLDB* (2010).
[10] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*.
[11] Mark Harris. 2007. Optimizing cuda. *SC07: High Performance Computing With CUDA* (2007).
[12] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* (2013).
[13] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *CSUR* (2008).
[14] James Malcolm et al. 2012. ArrayFire: a GPU acceleration platform. In *SPIE*.
[15] Duane Merrill and Andrew Grimshaw. 2011. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters* (2011).
[16] Todd Mostak. 2013. An overview of MapD (massively parallel database). *White paper, Massachusetts Institute of Technology* (2013).
[17] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. 2010. Fast in-place sorting with cuda based on bitonic sort. *Parallel Processing and Applied Mathematics* (2010), 403–410.
[18] Holger Pirk, Stefan Manegold, and Martin Kersten. 2014. Waste notâĂę Efficient co-processing of relational data. In *ICDE*. IEEE.
[19] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo-a vector algebra for portable database performance on modern hardware. *PVLDB* (2016).
[20] Eyal Rozenberg and Peter Boncz. 2017. Faster across the PCIe bus: a GPU library for lightweight decompression: including support for patched compression schemes. In *DaMoN*. ACM.
[21] Nadathur Satish, Mark Harris, and Michael Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–10.
[22] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *SIGMOD*. ACM.
[23] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *PVLDB* (2013).

A PER-THREAD TOP-K USING REGISTERS

Registers are the fastest layer of the memory hierarchy. However, as noted in Section 2.1, current generation GPUs do not have thread-local memory. A thread-local array can be made to use registers only if all its accesses are statically known. Without it, the compiler is forced to allocate the array in local memory which is off-chip and has a severe negative impact on performance. This prevents us from implementing a heap using registers as array accesses made during heap updates cannot be statically determined. We found that we can still maintain top-k per thread in registers by maintaining a list of top-k seen so far as a list and, keeping the index and value of the minimum value.

```
T buf[k];
T minValue; int minIndex;
```

If the element seen is greater than *minValue*, we update *minIndex* and find the new *minIndex*, *minValue* as follows:

```
minValue = xi
for j in range(0,k):
  if j == minIndex: buf[j] = xi
  if buf[j] < minValue:
    minIndex, minValue = j, buf[j]
```

While iterating over the elements of the buffer array creates overhead in the order of k, it allows the compiler to place the elements of *buf* in registers. The faster data accesses counteract the overhead for low values of k. For high values of k, the limited number of available registers forces the compiler to allocate some of the entries of *buf* in local memory even if the access is implemented in the manner described.

Figure 18 compares the time taken by the register-based version to the shared memory-based version to find the top-k from 2^{29} floating point numbers with varying k. We vary the distribution: (a) *Uniform*: numbers drawn from a uniform distribution $U(0, 1)$, (b) *Increasing*: numbers from $U(0, 1)$ sorted increasing and, (c) *Decreasing*: numbers from $U(0, 1)$ sorted decreasing.

The register-based top-k is slower than the equivalent shared-memory based top-k method for larger k because the register-based method starts spilling registers to local memory, which leads to significant slowdown. This is evident in the sharp slope going from $k = 32$ to $k = 64$ in the graph. Comparing the *increasing* and *decreasing* distribution, we see that the gap between the methods widens. This is because *increasing* has every number updating the top-k. Updates are more expensive in the list compared to the heap. In *decreasing*, there are no heap updates after inserting the first k elements.

B REBUILD-ALGORITHM

Algorithm 4 shows the pseudocode for the bitonic top-k rebuild operation.

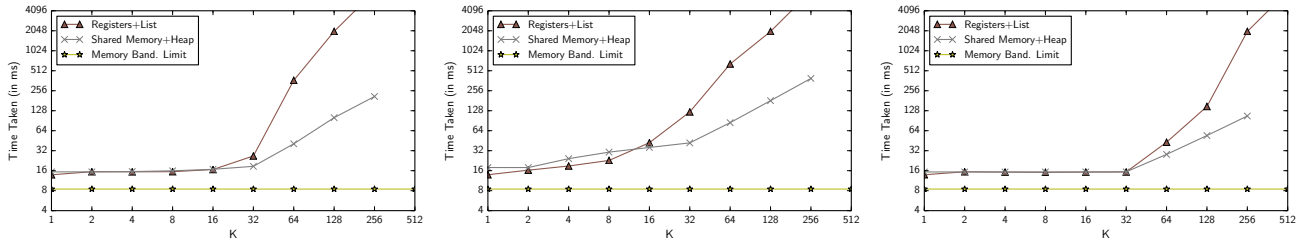
Algorithm 4: Bitonic Top-K *Rebuild*

Input : List L with bitonic sequences of length k
Output: L with sorted sequences of length k

```
1  int t = getGlobalThreadId();
2  int len ← k ≫ 1;
3  int dir ← len ≪ 1;
4  for inc ← len; inc > 0; inc ← inc ≫ 1 do
5      int low ← t & (inc − 1);
6      int i ← (t ≪ 1) − low;
7      bool reverse ← ((dir & i) == 0);
8      x0, x1 ← L[i], L[i + inc];
9      bool swap ← reverse ⊕ (x0 < x1) ;
10     if swap: x0, x1 ← x1, x0;
11     L[i], L[i + inc] ← x0, x1;
```

C BITONIC TOP-K ON CPU

The bitonic top-k algorithm presented in the paper can be adapted to run on the CPU as well. The bitonic top-k algorithm is reductive, it reduces an array of size n to an array of size k containing the



(a) Uniform **(b) Increasing** **(c) Decreasing**

Figure 18: Different Per-Thread Top-K Approaches

Algorithm 5: CPU Bitonic Top-K Thread

Input : Input Parititon S of length n; int k
Output: List O of the top-k elements per thread

```
1  int numElements ← n;
2  int numVectors ← numElements / vectorSize;
3  int temp[2][n/16];
4  int current ← 0;
5  for i ← 0; i < numVectors; i += 1 do
6      SortReducer(S, temp[current], i, k)
7  numElements ← numElements / 16;
8  while numElements >= vectorSize do
9      for i ← 0; i < numVectors; i += 1 do
10         BitonicReducer(temp[current], temp[1-current], i, k);
11     numElements ← numElements / 16;
12     numVectors ← numElements / vectorSize;
13     current ← 1 - current;
14 O ← sort(temp[current], numElements);
```

top-k elements. To make use of all the cores available, we partition the input array into equal sized partitions and let each core independently process the partition to emit the top-k. The top-k elements emitted by the individual core are combined in a final global step to find the global top-k.

On each core, we further break down the input partition into vectors of fixed size (in the implementation we use 2048 elements as the vector size). We process the input partition in phases. The first phase does the function of the SortReducer. It reads in the unsorted input paritition, one vector at a time and outputs $(1/16)^{th}$ of the input containing bitonic sequences of length k. The subsequent phases do the function of BitonicReducer. They read in the input containing bitonic sequences of length k, one vector at a time, and outputs $(1/16)^{th}$ of the input containing bitonic sequences of length k. Algorithm 5 shows the pseudocode.

On the GPU, each vector is processed in parallel by a thread block. Each thread of the thread block reads in 16 elements from shared memory and runs a combined step and outputs it back to shared memory. However, on the CPU, on each core, we process the vector in a single threaded fashion. The thread reads in 16 elements at a time from main memory and runs a combined step and outputs it back to shared memory. The reason we process small sized vectors (here of size 2048) is so that the data is cached in L1 cache. This allows random accesses in the vector to not incur latency of main memory read.

Modern CPUs also have support for Single Input Multiple Data(SIMD) instructions. The bitonic sorting network used to process a combined step can be implemented using SIMD instructions for improved performance. In the implementation we use 128-bit SSE-based implementation of [8]. Also, some of the optimzations details in Section 4.3 are not needed on the CPU. In particular, padding and chunk permutation are not useful on the CPU as there is no notion of bank conflict.

The bitonic top-k algorithm is not work-efficient. It does $O(n(logk)^2)$ number of comparisons as shown in Section 3.2. This is strictly worse than heap-based methods which do $O(nlogk)$ number of comparisons. However, bitonic top-k can leverage SIMD instructions to improve performance. Overall, in the case when lots of heap insertions occur (e.g.: when the input data is sorted) , the performance of bitonic top-k is close to that of heap-based methods despite the larger number of comparisons. Further, bitonic top-k could be better on platforms with wider vector instruction support like AVX-512 in Intel Knights Landing processors. We plan to explore this in the future.