GPU Accelerated Top-K Selection With Efficient Early Stopping

Vasileios Zois
University of California, Riverside
vzois001@ucr.edu

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

Walid A. Najjar
University of California, Riverside
najjar@cs.ucr.edu

ABSTRACT

Top-k selection retrieves the k highest ranking tuples from a given relation by utilizing a user-defined monotone function. Efficient query processing entails skipping evaluation of low ranking tuples by leveraging on early termination. Achieving this goal is possible using sophisticated data organization schemes combined with random access to resolve score ambiguity. Although these practices have proven to be successful for CPU based systems operating on disk-resident data, they have yet to be tested on modern in-memory systems utilizing GPU based processing. This problem is hard to tackle because random accesses are necessary for enabling high algorithmic efficiency (i.e. low number of object evaluations), while at the same time being inherently detrimental for GPU based processing. Existing solutions that rely on data re-ordering support sequential access at the expense of higher object evaluations. In our work, we investigate the effects of data preordering when combined with intelligent partitioning to enable efficient early termination on GPUs. We concentrate on evaluating the proposed solutions when data reside either in device or host memory. Our experimental results demonstrate the high potential of our methods for a variety of query parameters and data distributions. We showcase between $2\times$ to $200\times$ better query latency (executing on device or host memory respectively) when compared against state-of-the-art solutions that necessitate evaluation of all tuples in a given relation.

1. INTRODUCTION

Identifying interesting objects from a large database collection is a fundamental problem in multi-criteria decision making. Top-k queries present a widely accepted solution to this problem, indicated by their prevalence in the areas of information retrieval [9] and database systems [14]. Top-k selection involves retrieval of the k highest ranking tuples from a relation R using a user-defined monotone function F. Typically, this problem implies ranking tuples considering only a subset of their attributes. With the advent of cyber-physical systems [18] and IoT [21], Top-k selection on high-dimensional data has become increasingly important for supporting applications related to anomaly detection [22], data exploration [21] and visualization [17].

A straightforward approach for answering Top-k queries involves two steps: (1) calculating the score of each tuple by summing their weighted attributes (also known as tuple score aggregation), (2) utilizing sorting or k-selection algorithms to identify those tuples having the k highest scores/rankings. The most expensive part of Top-k query evaluation is score aggregation because during that phase data movement dominates the total execution time. This observation motivated the development of different methods, designed to avoid evaluating the complete data collection. Such methods often rely on threshold-based early termination [8, 16, 19] combined with data preordering [10] or layering [12] strategies.

Increasing memory capacity and decreasing memory costs motivated the development of in-memory database systems. Although the process of migrating in-memory has created several opportunities for improved query latency, their potential has been severely limited by the growing gap between processor and main memory speed. Further improvements in processing throughput and query latency can be obtained utilizing multi-core [2] processing or hardware acceleration [1, 15]. Related work has demonstrated the immense potential of GPU accelerated processing for filtering [24], and complex selection [4] operators. This body of work has revealed that caring about practices geared towards high throughput (i.e. coalesced memory access, minimal thread divergence) is as important as designing algorithmically efficient solutions.

GPU accelerated Top-k selection with support for early stopping has not been studied in previous work. It is a very challenging problem to tackle for two reasons: (1) Traditionally, Top-k query processing methods leverage on indexed based random access to resolve score ambiguity during tuple evaluation [3, 19], a practice that is inherently incompatible with GPU processing, (2) the immense compute capabilities of GPUs make it hard to justify the additional work that is required for enabling early termination. The latter point is concerned with avoiding intricate query evaluation strategies which might lead to higher query latency, despite enabling less tuple evaluations. Unless a satisfactory trade-off can be obtained there is no motivation to avoid evaluating the complete relation. Data preordering [10, 11] and layering [12, 16] are popular methods geared towards efficient sequential access. Despite being cheap to implement, their

10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'19), August 26, 2019, Los Angeles, California, CA, USA.

pruning abilities are severely affected for queries on relations with high number of attributes.

In this work, we investigate the suitability of data preordering and intelligent partitioning in order to enable efficient GPU based Top-k selection with support for early termination. We examine Top-k selection queries that involve high number of attributes and focus on techniques that utilize clustered indices to enable early termination. These techniques involve a single initialization step to build the underlying index (preordering step), after which multiple sub-queries can be efficiently executed on top of it. As established from previous work, such indices can function in a dynamic environment enabling low cost insertions/updates [12, 16]. Our ultimate goal is to improve query latency by developing solutions suitable for massively parallel architectures. The main contributions of this work are summarized below:

- We develop the skeleton of a parallel threshold algorithm (see Section 3.2) that is designed to enable efficient GPU Top-k selection with support for early termination.
- We consider two different data partitioning strategies and evaluate their effectiveness when combined with data preordering (see Section 3.3).
- We study the performance characteristics of GPU-based Top-k selection and evaluate our proposed solutions for a variety of parameters, including result size, attribute number, and variable preference vectors.

The rest of the paper is organized as follows: In Section 2, we discuss the GPU architecture and the details of previous work on GPUs, while Section 3 contains a thorough discussion of the proposed framework. Section 4 describes the experimental evaluation and Section 5 concludes the paper.

2. BACKGROUND

This section is devoted towards formally defining the Top-k problem, reviewing GPU architectural characteristics and programming practices, and presenting the details of a state-of-the-art GPU based k-selection algorithm. Our analysis concentrates on showcasing the shortcomings of that algorithm when used to solve the Top-k selection problem.

2.1 Problem Description

Let R be a relation consisting of N tuples with D attributes each ($t = \{a_0, a_1, ...a_{d-1}\}$) in range $(0, L]$. A user-defined ranking function $F(t)$, also known as a preference vector, maps the objects in R to values in the range $(-\infty, \infty)$. In related work, it is common to assume that the given function is monotone [7, 13, 26, 28] and linear [6, 7, 13]. A linear function is defined formally as:

$$F(t) = \sum_{i=0}^{m} (w_i \cdot a_i) \tag{1}$$

The variable w_i is the corresponding weight for each one of the m attributes of the user subquery. A *monotone* ranking function satisfies the:

$$\begin{gathered} if\ t_u(a_i) \geq t_v(a_i), \forall i \in [0, d-1] \\ then\ F(t_u) \geq F(t_v) \end{gathered} \tag{2}$$

Hence, our goal is to discover a collection S of tuples $[t_1, t_2, ...t_k]$ such that $\forall j \in [1, k]$ and $\forall t_i \in (R - S)$, $F(t_j) \geq F(t_i)$. Following the majority of previous work [14], our work is applicable for every possible monotone function.

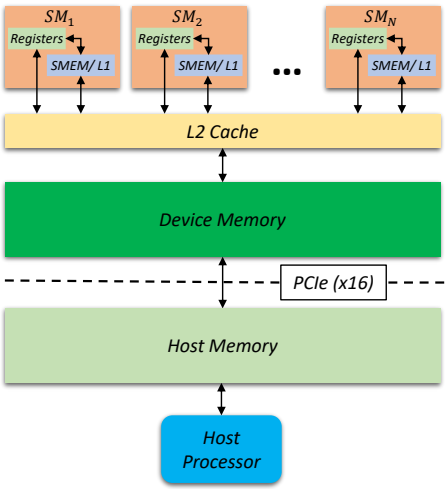

Figure 1: A simplified view of the GPU architecture showcasing the hierarchy of different memory levels accessible to the associated multi-processors.

2.2 GPU Architecture & Organization

A simplified depiction of the GPU architecture and memory hierarchy is shown in Fig. 1. GPUs consist of multi-core processing units known as Streaming Multiprocessors (SMs), each one containing their own set of registers, L1 cache and a software programmable cache (i.e. shared memory). In addition, each SM has direct access to a shared L2 cache and a dedicated RAM often designated as global or device memory. Programs execute on the GPU in the form of kernels. Each kernel utilizes thousand of active threads, typically grouped into thread blocks. Thread blocks share access to L1 cache and shared memory, while each thread within a block has private access to their own set of registers. Blocks are split further into warps which take turns executing in lock-step using any available SM. The threads within a warp should access data stored in global memory sequentially to ensure maximum bandwidth utilization. In addition, a sufficient number of active warps is necessary to effectively mask the latency associated with instruction dependencies (i.e. data access, synchronization).

Although the device memory offers high bandwidth its capacity is limited to only few GBs (i.e. $12 - 24$ GBs). For this reason, GPUs may rely on the host memory for storage, retrieving (across PCIe) the necessary data on-demand during processing. In modern GPUs, this is made possible through the use of a unified virtual memory space that is managed seamlessly either by the GPU driver or the programmer. The GPU driver facilitates data exchange across PCIe utilizing two types of memory declarations: (1) *Zero copy* memory initiates data transfers each time a GPU kernel is executed, (2) *Managed* memory utilizes heuristics and hints during runtime to prefetch the necessary data into device memory. The latter method works also as a caching mechanism being able to retain data and re-use it in future kernel calls.

2.3 Bitonic Top-k Selection

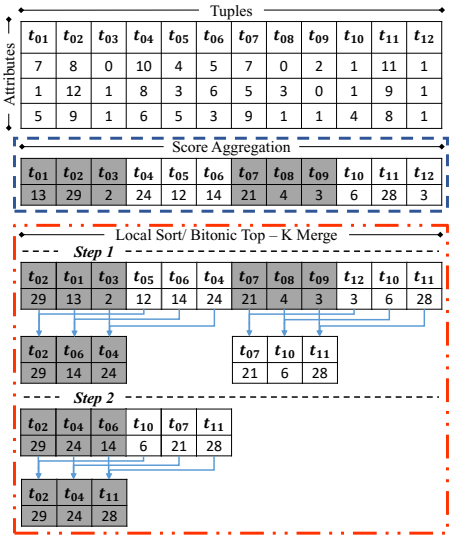

Figure 2: Different stages associated with Top-3 selection using Bitonic Top-k.

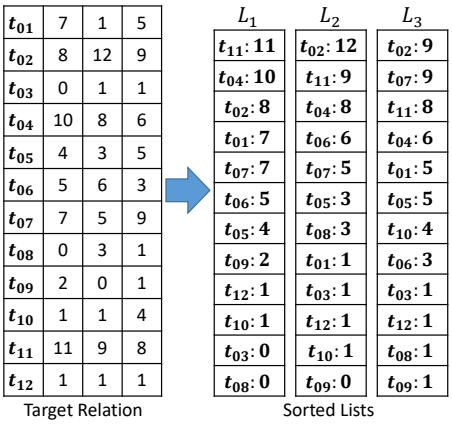

Figure 3: An example of mapping a base relation (left) to a collection of per attribute sorted-lists (right).

Typically, GPU enabled algorithms operate on data that reside in device memory. In this environment, it is important to take advantage of the immense GPU memory bandwidth by enabling coalesced data accesses when reading from and writing to the device memory. When the associated data are used multiple times during computation, it is commonplace to avoid unnecessary memory transactions by storing and operating on them through registers or shared memory.

A simple implementation of Top-k selection on GPUs, requires first aggregating the scores of all tuples in a given relation using the user-defined monotone function, and then utilizing a k-selection algorithm to identify those tuples with the k-highest scores. Bitonic top-k [23] is the state-of-the-art k-selection algorithm for GPUs. Its main goal is to avoid completely sorting the key-value pairs that are generated after the aggregation step. In order to achieve this, it executes bitonic sort to create k-sized groups of data which are sorted in alternating order. Consecutive groups are combined using bitonic merge, and the process repeats until a single group with the k-highest scoring tuples is created. An example of this process to calculate the Top-3 query is shown in Fig. 2. The bitonic top-k algorithm operates on the key-value pairs generated by summing all attributes of the input relation. Bitonic sort (a.k.a local sort) and bitonic merge execute in sequence by repeatedly sorting and extracting the maximum values until the 3-highest scoring tuples remain.

Despite its higher complexity (i.e. $O(N \log^2 k)$, bitonic top-k performs better than sorting or previous k-selection algorithms based on radix-sort because it avoids expensive scatter operations while also considerably reducing the total amount of data being written back to global memory [23]. Nevertheless, the performance of Top-k selection based on bitonic top-k drops when the target relation contains a large number of attributes. In that scenario, the aggregation phase dominates the total execution time because processing is limited by how fast the data can be read from device memory. In addition, for very large relations that cannot fit in device memory, the required attributes need to be fetched from host memory at the moment of query evaluation. In both, circumstances evaluating all the tuples is detrimental to query latency. Hence improving performance is connected to reducing the overall number of tuples being evaluated.

In relational databases the proven way to achieve this is to utilize a threshold based indexing scheme [3,8,10,12,19] able to support sub-queries and variable preference vectors. Typically, such solutions require an initialization step where the index is build, after which many queries can be executed on top of the existing data structure. Such methods also support data schemes which can be easily updated in a dynamic environment. However, these early termination solutions are not directly applicable to the GPU environment because they are known to incur too many random accesses [3, 8]. Methods optimized for sequential access [10,12,19] exist but operate at the expense of higher number of tuple evaluations. In the next section, we review TA, a threshold based early termination solution, and describe a generic framework for developing efficient threshold based algorithms for the GPU using data preordering and layering.

3. GPU THRESHOLD ALGORITHMS

List-based algorithms utilize a collection of per attribute sorted lists to enable efficient Top-k query processing. These lists present a simple data abstraction that is resilient to different query parameters (e.g. variable preference vectors, result size, attribute number) and can be easily realized in a dynamic environment using self-balancing trees (i.e. B/B+trees). An example of those sorted lists build upon a toy relation is shown in Figure 3. The majority of list-based methods follow a similar execution model to the one that was established by the Threshold Algorithm (TA) [8]. The main idea is to iterate over the sorted lists in round robin order and calculate the score of each seen tuple through random access to every other list. The seen tuples with highest scores are maintained using a priority queue that is updated periodically as new tuple-score pairs are generated. Algorithm 1 summarizes the steps associated with TA's execution. We start by initializing an empty priority queue (Line 1) and set the threshold value to zero (Line 3). As stated before, we iterate through all lists in round robin order (Line 4) retrieving one tuple (i.e. tuple-id, attribute value) at a time from each sorted list (Line 5).

We use the retrieved key-value pairs to update the current threshold value (Line 6). Unless the tuple was evaluated in the past (Line 7), we continue by inserting the tuple-id into a hash-table (to keep track of evaluated tuples) and initialize the score of the associated tuple equal to the value of the retrieved attribute (Line 11). An index is used to retrieve the remaining attributes of the given tuple from every other list (Line 13) which are then aggregated to the total score of that tuple (Line 14). We update the priority queue with the new tuple if its score is greater than the minimum or less than k tuples have been discovered (Lines 16 - 23). Query processing continues until k items have been discovered and the minimum scoring item has a ranking greater than or equal to the threshold of the current list level (Line 25).

Algorithm 1 Threshold Algorithm

L = Sorted list collection.
W = Preference vector.
k = Result size.

```
1:  Q = {}                                  ▷ Initialize empty priority queue.
2:  do
3:      T = 0                               ▷ Initialize threshold value.
4:      for i ∈ [1, d] do
5:          (T_id, A_i) = getNextObjectFromList(L_i)
6:          T = T + W_i · A_i               ▷ Update threshold.
7:          if T_id ∈ M then                ▷ Check if object seen before.
8:              continue
9:          end if
10:         M.push(T_id)
11:         S = W_i · A_i
12:         for j ∈ [1, d] AND j ≠ i do
13:             A_j = getValueByKeyFromList(T_id, L_j)
14:             S = S + W_j · V_j
15:         end for
16:         if Q.size() < k then
17:             Q.push(T_id, S)
18:         else
19:             if Q.top() < S then
20:                 Q.popMin()
21:                 Q.push(T_id, S)
22:             end if
23:         end if
24:     end for
25: while Q.size() < k AND Q.top() < T
```

Ignoring memory accesses associated with updating the hashtable (in order to avoid duplicate tuple evaluations), for every tuple evaluation, TA performs 1 sequential access (Line 5) to the corresponding sorted list and $d-1$ random accesses (Line 13) to find the remaining attributes from every other list. The number of tuple evaluations increase rapidly with respect to increasing query attributes [10], a behavior that affects proportionally the total number of random accesses. Random accesses are not compatible with GPU based processing which often relies on coalescing to fully utilize the available memory bandwidth. Data layering [12, 16] or tuple preordering strategies [10] are used to eliminate random accesses at the expense of higher tuple evaluations. In fact, their performance degrades rapidly for high dimensional relations with extreme data variability (i.e. correlated, independent, and anti-correlated data distributions [5]). As opposed to data re-ordering, data layering incurs a higher initialization cost, therefore the former method is a better candidate for GPU based processing.

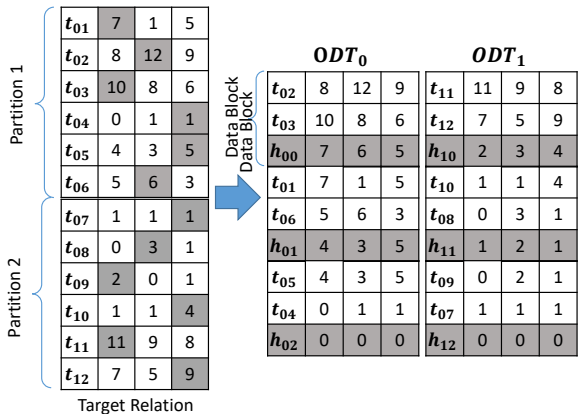

Figure 4: An example of mapping a base relation (left) to multiple ODT tables (right) by preordering tuples based on the maximum attribute value (indicated in gray).

In order to resolve the above issues, we present a simplistic data layout scheme based on data preordering that can be adopted to enable efficient GPU based processing. In addition, we describe the major components of a generic framework for developing efficient GPU Threshold Algorithms (GTA). Utilizing this framework, we concentrate on developing and evaluating two algorithms which use different partitioning schemes when assigning work to distinct thread blocks, namely, GTA with random partitioning (GTA-RP) and GTA with angle space partioning (GTA-ASP). We show empirically and experimentally that intelligent partitioning contributes towards high algorithmic efficiency.

3.1 Ordered Data-Threshold Table

Designing GPU-friendly threshold algorithms necessitates preordering the tuples of a relation to enable coalesced data access. This practice is often detrimental for queries that execute only on a subset of all available attributes (i.e. subqueries). Queries on skewed distributions (i.e. anti-correlated data) or those evaluated on relations with high number of attributes become very challenging to process. Such behavior is related to the number of possible tuple orderings which grow exponentially to the number of query attributes. It is possible to overcome the aforementioned issues by restricting the value range of the associated attributes for a collection of tuples, through intelligent partitioning. Creating these range boundaries limits the number of possible tuple orderings within a partition, thus enabling a better total ordering that is beneficial for early termination.

Investigating this hypothesis requires first describing the central component of our broader data organization scheme, henceforth referred to as Ordered Data-Threshold (ODT) table. ODT tables are formulated by rearranging/layering the tuples of a target relation so as to ensure early evaluation of those with the greatest likelihood to score high. Different preordering strategies are possible including those based on skyline layering [12] or first seen position using list-based ordering [10]. In order to simplify discussion and construction of ODT tables, we order the tuples according to their

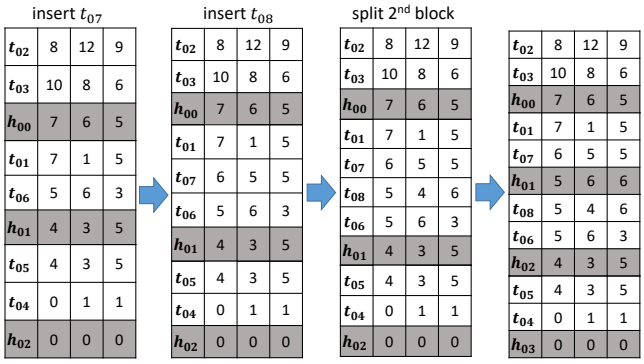

Figure 5: Example depicting insertion of a new tuples in a given ODT table.

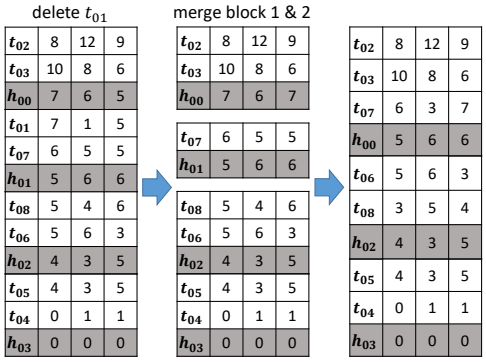

Figure 6: Example depicting deletion of an object from a given ODT table.

largest attribute as shown in Fig. 4. In that example before creating each ODT table, we partition the data into two distinct collections. Although different partitioning strategies are possible (see Section 3.3), we group tuples based on their insertion order just for demonstration purposes. An ODT table is logically split into *ordered* data blocks, each one containing several tuples from the original relation plus one extra tuple (depicted in grey), known as the the threshold tuple. In the general case, where a relation is divided into p partitions, and b data blocks per ODT table, a single data block contains a set of relation tuples (C_{ij}) and a *threshold* tuple (H_{ij}), where $i \in [0, p-1], j \in [0, b-1]$. Let $C_{ij}[n, d]$ be the d-th attribute of the n-th tuple in C_{ij} then the threshold tuple H_{ij} is calculated as follows:

$$H_{ij} = \{a_m | a_m \geq \underset{r \in [j+1, b-1]}{\arg\max} C_{ir}[n, m]\} \quad (3)$$

The threshold tuple (H_{ij}) contains the maximum attribute values among those in the tuples of any subsequent data block. It is useful for determining when to safely stop query processing because it provides information about the maximum possible score of the tuples which are yet to be processed. When constructing the associated ODT tables, the threshold tuples are computed inexpensively by keeping track of the maximum attribute values during the assignment of every tuple to its corresponding data block. Consider the example of Fig. 4, for a Top-2 query on all attributes, in ODT_0 tuples $t_2 = 29$ and $t_3 = 24$ will be evaluated and processing will stop at the first data block because the threshold tuple $h_{00} = 16$ guarantees that no tuples exist with higher score. Note that the chosen block size is independent of the query result size k. In a real life application, our method would operate on hundreds of partitions containing thousands of blocks each with variable query parameters including varying query attributes (m), result size (k), and tuple number (n). Note that the threshold attributes can be updated iteratively by examining only the attribute values of neighboring blocks to those where new tuples are being added or existing ones deleted (see Construction & Maintenance).

ODT Construction & Maintenance. ODT tables possess similar properties to that of data layering strategies [10, 12]. However, they are simpler to build and inherently well structured for coalesced data access. They can be constructed in batch using simple highly parallel GPU primitives (i.e. sort, parallel reduction). In addition, ODT tables can be easily maintained in a dynamic environment by retrofitting them to support insert and delete operations using a self-balancing tree structure (i.e. B/B+ tree). In that environment the data blocks of a given ODT table, correspond to the leaf pages of the associated tree structure (think of a clustered index). These leaf pages are created by indexing the maximum attribute of each tuple and are augmented with a threshold tuple as described previously. Any insert, update or delete operation will be managed on the CPU side, allowing the GPU to operate on a read-only instance of the transformed relation. Efficient GPU based processing is contingent on enabling coalesced access within any given data block, thus linking randomly pages in main memory will not degrade performance.

Below we demonstrate the capability of ODT tables to support insert and delete operations using two examples indicating their behavior during such scenarios. For these examples, we assume a minimum and a maximum block size of 2 and 3 respectively. In Figure 5, we showcase how an ODT table is updated during several consecutive tuple insertions which lead to a block split operation. A new tuple $t_v = \{a_0, a_1, ..a_{d-1}\}$ is inserted into an ODT table by utilizing binary search to discover block B having a threshold tuple $h_{ij} = \{t_0, t_1 ... t_{d-1}\}$ such that $\exists a_m \in t_v$ where $a_m > t_m$ where $m \in [0, d-1]$. In our example, t_{07} will be inserted in the second block because at least one of its attributes is greater than the equivalent attributes of h_{01}. The same happens for tuple t_{08} after the insertion of which a split operation occurs due to the current block size being larger than the maximum (3). For a block that is being split into two new ones, we preorder the tuples according to their maximum attribute value and group them into blocks of size equal to the minimum. In the previous example, this will result in two groups, one containing t_{01} and t_{07} and the other t_{08} and t_{06}. For the first group, the threshold tuple is calculated by finding the maximum attributes of the subsequent block (third block). The second group retains the threshold tuple of the original block, as it was before the split, since no changes occurred below that block.

Figure 6 showcases what happens when a delete operation causes the merging of two blocks. There tuple t_{01} is deleted resulting in the second block having less tuples than the minimum allowed. When merging two ordered blocks B_i and B_j, where B_j comes after B_i, we create a new block with all their tuples combined and a threshold tuple equal to that of B_j. In our example, the first and second blocks

combined together utilizing the threshold tuple of the latter for that of the new block.

Algorithm 2 Aggregate-Heap Building (hbuild)

C = ODT collection.
S = Tuple-id, scores buffer.
Q = Tuple-id, scores heap.
W = Preference vector.
k = Result size.

```
1: for i ∈ [1, p] in parallel do
2:     for j ∈ [1, b] do
3:         for (t ∈ C_ij) & (h ∈ H_ij) in parallel do
4:             S_it = Σ_{m=1}^{d} (w_m · t_m)      ▷ Score aggregation.
5:             T_h = Σ_{m=1}^{d} (w_m · h_m)       ▷ Threshold value.
6:         end for
7:         __syncthreads()
8:         Q_i = hmerge({Q_i ∪ S_i}, k)            ▷ Bitonic merge.
9:         if Q_i.min() >= T_h then                ▷ Early stopping.
10:            return Q_i
11:        end if
12:    end for
13: end for
```

3.2 Heap Build & Reduction

GTA operates in two phases: (1) Aggregate-Build Heap phase (hbuild), (2) Heap Merge phase (hmerge). Each phase is implemented using a distinct GPU kernel. A simplified description of the first phase is shown in Algorithm 2. Every partition contains a single ODT table, assigned for processing to a single thread block (Line 1). Threads within a block are responsible for aggregating in parallel the scores of several tuples (Line 3-5) and calculating the threshold value (Line 5). At the end of every data block evaluation the current collection of <tuple-id,score> pairs are combined with the k highest scoring pairs identified from previous iterations (Line 6). This heap is stored in shared memory and is constructed using Bitonic Top-k. The code responsible for merging is similar to that of the hmerge kernel (Algorithm 3). At the end of the merge step, the minimum heap score is compared to the associated threshold (Line 8) to determine when the Top-k answer for a given partition becomes available. Unless this condition is false, we write the corresponding pairs in global memory and terminate processing. The hmerge operates on the individual heaps created by hbuild. A fixed collection of ¡tuple-id, score¿ pairs is assigned to distinct thread blocks for processing. Each thread block reduces their input set to k pairs having the highest score by combining bitonic sort (only the first k iterations [23] Line 2) and parallel reduction (Line 4). Several rounds of sort-reduce operations are executed in sequence until only k pairs remain.

3.3 Data Partitioning Strategies

Data partitioning is crucial for achieving efficient GPU based processing. The rationale is that good partitioning facilitates workload balance which is pivotal for masking data access latency and maximizing throughput by using an adequate number of operating threads. Likewise, in Top-k selection data partitioning influences algorithmic efficiency since it restricts access to tuples that can effectively prune the search space. Achieving a balance between these extreme cases is possible using an intelligent partitioning scheme.

Algorithm 3 Heap Merge (hmerge)

S = Tuple-id, scores collection.
k = Result size.
n = Score buffer size.

```
1: while n > k in parallel do
2:     bitonic_sort(S, k, n)                  ▷ Sort up to k.
3:     S_i = max(S_i, S_{i+k})                ▷ Bitonic merge.
4:     n = n/2
5: end while
6: return S_{0:k}   ▷ Return k highest tuple-id, score pairs.
```

Random partitioning (RP) groups together tuples according to their relative position within a target relation; an example of RP is shown in Fig. 7 (left). This method of partitioning is beneficial for two reasons: (1) it incurs almost zero initialization cost, (2) it constructs partitions having approximately the same size, which supports workload balance during processing. However, this practice might contribute to the creation of partitions that consist primarily of anti-correlated data. In this case, algorithmic efficiency and in turn query latency are adversely impacted by the fact that an optimal query evaluation depends on different tuple re-orderings dictated by variable query parameters (i.e. attribute number, preference vector, result size).

$$
\begin{aligned}
tan(\phi_1) &= \frac{\sqrt{(\tilde{A}_d)^2 + (\tilde{A}_{d-1})^2 ... + (\tilde{A}_2)^2}}{\tilde{A}_1} \\
&... \\
tan(\phi_{d-2}) &= \frac{\sqrt{(\tilde{A}_d)^2 + (\tilde{A}_{d-1})^2}}{\tilde{A}_{d-2}} \\
tan(\phi_{d-1}) &= \frac{\tilde{A}_d}{\tilde{A}_{d-1}}
\end{aligned}
\tag{4}
$$

Angle space partitioning (ASP) formulates multiple data collections by enabling grid partitioning on the polar coordinates (Eq. 4) of every tuple in the target relation. Considering geometric symmetry and the fact that we are interested in the highest ranked tuples, we compute the polar coordinates of $\tilde{A}_i = (\alpha - A_i)$ where A_i is the i-th attribute of each tuple having values in range $(0, \alpha]$. The number of resulting partitions is determined by the number of split points for each angular dimension. Assuming s split points, the resulting number of partitions is s^{d-1}, where d is the number of attributes in the based table. Alternatively, there exist solutions enabling equi-volume partitioning using ASP [27]. In our experiments, we concentrate on regular grid partitioning since it incurs lower initialization cost without noticeable difference in query latency.

A toy example indicating the different characteristics of ASP vs RP is shown in Figure 7. In that example, each partition consists of tuples indicated with similar point shape and color. Compared to RP, ASP is better alternative when partitioning the data for Top-k selection. This happens because the latter method creates partitions containing tuples with correlated attributes which are inherently easier to linearly order for any monotone function utilizing the ODT table concept. This conjecture is also applicable to relations with more than 2 attributes and has been showcased to be

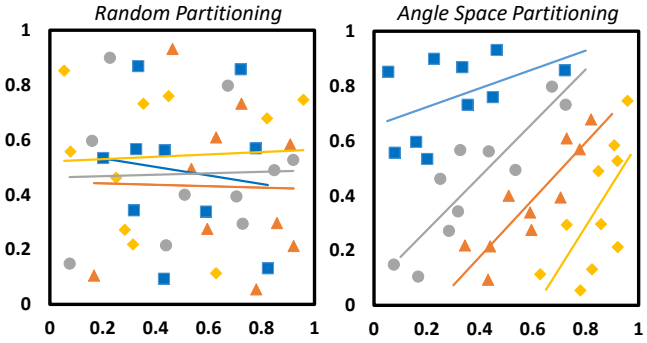

Figure 7: Example of partition formulation (indicated as points with same color and shape) when using Random Partitioning (RP) vs Angle Space Partitioning (ASP).

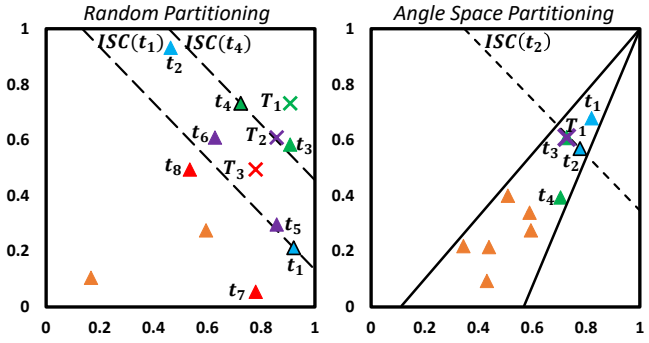

Figure 8: Order of evaluation (1:cyan, 2:green, 3:red) of the points in a single partition when utilizing Random Partitioning (RP) vs Angle Space Partitioning (ASP).

effective for skyline computation [27], where attaining a near optimal linear order is critical for improving algorithmic efficiency.

In order to demonstrate ASP's superiority in a more intuitive manner, we utilize the concept of "identical score curve" (ISC) as proposed in [25]. ISC is the line corresponding to equation $f(t) = v$, consisting of tuples (t) in the data space whose scores are equal to v. Let t_k be the minimum scoring tuple in the priority queue at some point during query evaluation, $ISC(t_k)$ the line defined by equation $f(t_k) = v_k$ and T_i the corresponding threshold tuple. Assuming the priority queue contains k tuples, query processing terminates if and only if $F(T_i) \leq F(t_k)$ has been satisfied. This indicates that T_i must be on or below $ISC(t_k)$ inside the half-space that is closer to the origin. In Figure 8, we concentrate on one partition from each partitioning method and plot the corresponding ISC, and threshold tuples at various points during Top-2 query processing. The example of Figure 8 derives from the partitions of the toy dataset depicted in Figure 7. Our goal with this example is to demonstrate how different partitioning strategies affect early termination by influencing $ISC(t_k)$ and T_i respectively. Note that because the tuples are reordered based on their maximum attribute value, the order in which they are visited during query processing is equivalent to performing in succession a plane sweep of each axis.

In the RP example (Figure 8 left), the first two tuples evaluated are $t_1 = (x_1, y_1)$ and $t_2 = (x_2, y_2)$. Query processing will terminate if the corresponding threshold tuple is below the halfspace defined by $ISC(t_1)$. In the worst case, because of random partitioning the threshold tuple will consist of attributes that are arbitrarily close to the maximum from those in at least one of t_1 and t_2. When this happens, it is very likely for the threshold tuple to reside in the halfspace above $ISC(t_1)$. In fact, in our example the threshold tuple T_1 contains x_3 from $t_3 = (x_3, y_3)$ which is very close to the value of x_1 from t_1. Thus, in the first iteration query processing does not terminate because the stopping condition is not satisfied. Because the tuples within the given partition are not restricted in any way, it takes several iterations and evaluation of tuples t_3, t_4, t_5, t_6, before Top-2 selection can safely terminate. This happens because T_2 resides above the halfspace defined by $ISC(t_4)$, and only after t_5, t_6 are evaluated, the new threshold T_3 is available for consideration, indicating that no tuples exist with better score than that of t_4.

ASP creates partitions with tuples having attributes restricted by the partition boundaries. This practice restricts the value range of the threshold attributes, subsequently reducing the threshold and enabling early termination with fewer tuple evaluations. In the ASP example (Figure 8 right), after evaluating $t_1 = (x_1, y_1)$ and $t_2 = (x_2, y_2)$, t_3 and t_4 are combined to create T_1 which is the current stopping threshold. At this point, because T_1 is in the halfspace below $ISC(t_2)$ processing can safely stop since we have discovered the two highest ranking tuples and satisfied the stopping condition. Therefore, for a single partition only 2 evaluations were required as opposed to 6 when using RP. When the partition angle is small, the partition boundaries restrict the threshold attributes, resulting in rapidly decreasing threshold score which contributes towards early with few tuple evaluations.

3.4 GTA Complexity

In this section, we analyze the complexity of GTA and the manner in which it is affected by the previously mentioned partitioning strategies (i.e. RP and ASP). Let L be the maximum attribute value for all n tuples and m be the number of query attributes. GTA evaluates each tuple within a given partition in-order of their maximum attribute. Considering

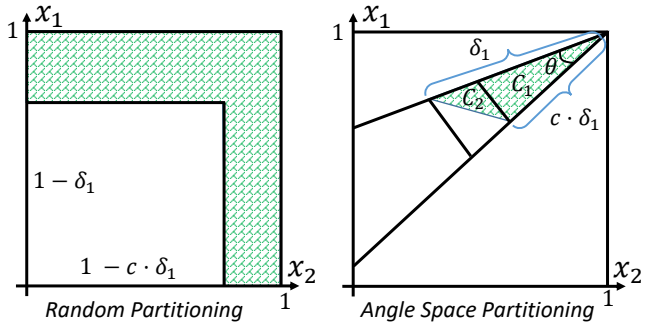

Figure 9: Processed area indicated in green for variable δ_i when utilizing different partitioning schemes (i.e. Random Partitioning (RP) or Angle Space Partitioning (ASP).

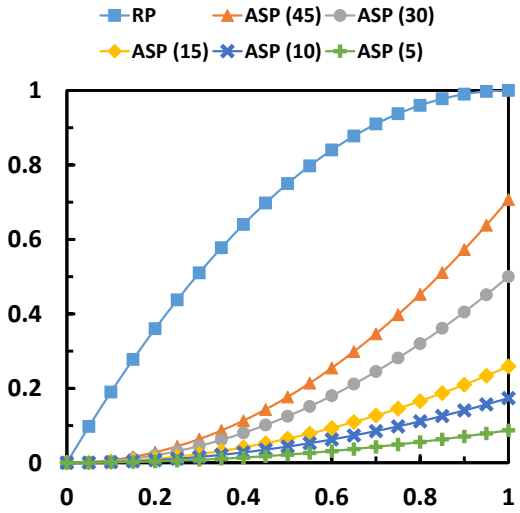

Figure 10: Expected processed area as function of δ_1 using RP and ASP(degrees).

that the tuples are mapped in multidimensional space, this order of processing is equivalent to a plane sweep of the axes that correspond to the actual tuple attributes.

Let $[L, L-\delta_j]$ $(i \in [1,m])$, be the region processed by GTA for some axis x_i, at some point during query evaluation. Due to the order in which tuples are visited during processing, GTA would have evaluated some tuple t_i $(i \in [1,n])$, if and only if t_i contains at least one attribute $a_j \in [L, L-\delta_j]$. In this case, it is possible to realize GTA's complexity as a function of δ_i by calculating the volume (or area in 2D) of the polytope defined by hypercube $[0,L]^m$ and the intersection of every "swept" region $[L, L-\delta_j]$ $(i \in [1,m])$. Our analysis is applicable for uniformly distributed points (tuples) in space and assume the existence of a single data partition. However, it is a scenario that could arise in the worst case with many partitions when RP is used because there is no restriction on the attribute range when tuples are assigned to distinct partitions.

In order to simplify the discussion and without loss of generality, we continue our analysis by concentrating on the $2D$ case where $L = 1$. Figure 9 (left) provides an illustration (depicted in green) of the area that has been processed after regions $[1, 1-\delta_1]$ and $[1, 1-c\cdot\delta_1]$ have been "swept" by GTA. We correlate $delta_2$ to δ_1 through $c \in [0,1]$ in order to emulate the different query parameters (e.g. preference vector, result size) and data distributions that could possibly affect how they evolve during processing. The expected processed area when using RP as function of δ_1 and c equals:

$$E_{RP} = 1 - (1-\delta_1)\cdot(1-c\cdot\delta_1) \tag{5}$$

Following the same process, we showcase in Figure 9 (right) the area of a partition defined by angle θ that has been processed after some point during query evaluation. The partition boundaries dictate the value of δ_i because no tuple within the partition will have attributes outside that range. In this case, we can represent the processed areas as a function of δ_1 and c using the following equation:

$$E_{ASP}\delta_i^2 \cdot sin(\theta) \cdot \frac{(3+c^2)}{4} \tag{6}$$

The previous equation was calculated by finding the combined area of triangles C_1 and C_2. Equation 6 indicates that a smaller partition angle contributes towards the reduction of the total area that needs to be processed for a given δ_1 value. However, we cannot decrease the partition angle indefinitely because it may result in evaluating more tuples than necessary since from each partition at least k tuple will be evaluated. This can be realized in Figure 10 where we plot the corresponding area values for increasing δ_1, $c = 1$ and varying angles. In this we observe that the processed area value decreases less rapidly after 15 degrees. Therefore, there is little benefit in having too many small partitions. We discovered experimentally that utilizing ASP by having 512 to 2048 partitions is enough to attain a good trade-off between having enough work for the GPU to operate efficiently and reducing the number of tuple evaluations.

4. EXPERIMENTAL EVALUATION

In our experimental evaluation, we consider two GTA algorithms: (1) GTA-RP which utilizes random partitioning, (2) GTA-ASP which utilizes angle space partitioning. Due to lack of existing solutions that support early termination on GPU, we compare against Bitonic Top-k [23], henceforth denoted as GFTE (GPU Full Table Evaluation).

In order to demonstrate the importance of early termination, we explore two different experimental paradigms (1) where the data reside in device memory and are directly accessible by the GPU Streaming Multiprocessors (SMs), (2) where the data reside in host memory and are managed explicitly by NVIDIA's unified memory driver. The latter form of data management has two modes of execution, one where the driver retrieves the data during the kernel's first call (Zero Copy Mode), and another where it receives a hint to prefetch the necessary data before query evaluation (Prefetching Mode).

In addition to our comparison with state-of-the-art GPU based algorithms, we implemented and evaluated the performance of CTA-ASP, an equivalent solution to GTA-ASP, that is optimized for CPU based processing using multithreading and AVX instructions. We provide discussion comparing CTA-ASP and GTA-ASP against their corresponding full table evaluation algorithms which are optimized for CPU (CFTE) and GPU (GFTE) processing, respectively.

Q_0	$(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)$
Q_1	$(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8)$
Q_2	$(0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1)$
Q_3	$(0.1, 0.2, 0.3, 0.4, 0.4, 0.3, 0.2, 0.1)$
Q_4	$(0.4, 0.3, 0.2, 0.1, 0.1, 0.2, 0.3, 0.4)$

Table 1: Query weight values for the given preference vector configuration.

Our experiments were conducted using a single 12 GB NVIDIA Titan V GPU attached to a single socket Intel Xeon E5-1650 processor @ 3.5 GHz with 32 GB of RAM. All algorithms were implemented using standard C++, CUDA 10.0 and NVIDIA's CUB Library [20]. The CPU implementations utilize distinct priority queues for every thread, which are combined towards the end of query evaluation. Our code is publicly available in Github [29].

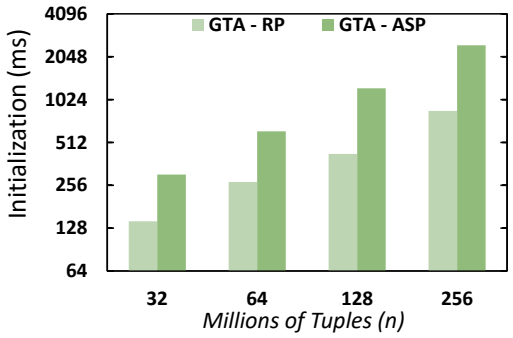

Figure 11: Measured latency to enable data partitioning and tuple preordering.

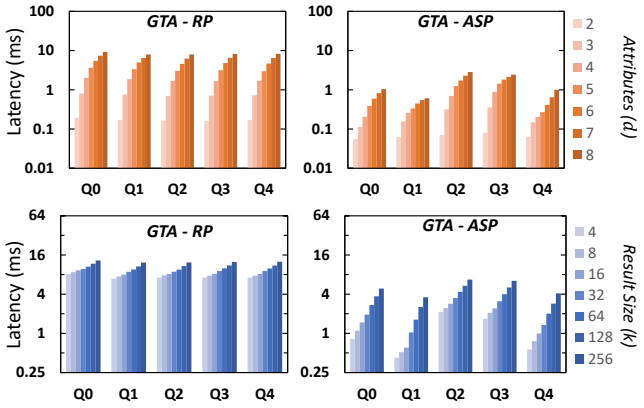

Figure 12: Measured query latency when utilizing variable weights in the preference vector.

4.1 Dataset, Queries & Metrics

Following the example of previous work [12, 19, 30], we conducted experiments using synthetic data and three types of distributions, mainly correlated, independent and anti-correlated, generated using the readily available data-set generator [5]. Our experiments concentrate on measuring query latency for variable attribute number ($d \in [2, 8]$) and result size ($k \in [4, 8, 16, 32, 64, 128, 256]$) for a relation containing 256 million tuples with 8 attributes each (8 GB of raw data). For experiments with data that reside in host memory, we generated a relation with 8 attributes and 512 million tuples (16 GB of raw data). Finally, we measured the execution time on independent data utilizing variable preference vectors as summarized in Table 1.

4.2 Initial Cost of Indexing

Figure 11 indicates the total cost of initialization which includes partitioning and the host-to-device communication when building the ODT tables. ASP incurs at most twice the initialization overhead of RP while being 2× to 200× faster in terms of query latency, as it will be come clear in the following sections. Initialization occurs only once before any queries are executed and is commonplace for all list-based early stopping algorithms [8,12,16,19]. Note that regardless of the chosen partitioning strategy, ODT tables can be constructed utilizing a "bulk loading" algorithm and maintained using insert/delete operations as described in Section 3.1. In addition, our methods are generic in that they can be applied on an arbitrary number of attributes and support queries only for a subset of them. This property is demonstrated throughout our experimental evaluation where we present the query latency for sub-queries on 2 to 8 attributes for a target totaling 8 attributes per tuple.

4.3 Variable Preference Vectors

As indicated in Fig. 12, there is no difference in execution time when using random partitioning for every tested preference vector and query parameters. On the other hand, ASP experiences somewhat noticeable change in execution time across different queries. This behavior is associated with the way query weights influence the associated processing workload of each partition. ASP operates on all attributes, though it correlates them through angular coordinate calculations which consider fewer of them in ascending attribute order (see Eq. 4). This ordering lessens the effects of other attributes in the corresponding angular coordinate calculations for those appearing at the end.

Hence, preference vectors that favor these attributes using higher weights (see Q_1, Q_4) will naturally result in less work to process the corresponding edge (those closer to the axes in high-dimensional space) partitions. In contrast, symmetrically opposite preference vectors (see Q_2, Q_3) are responsible for higher execution time since the angle coordinates

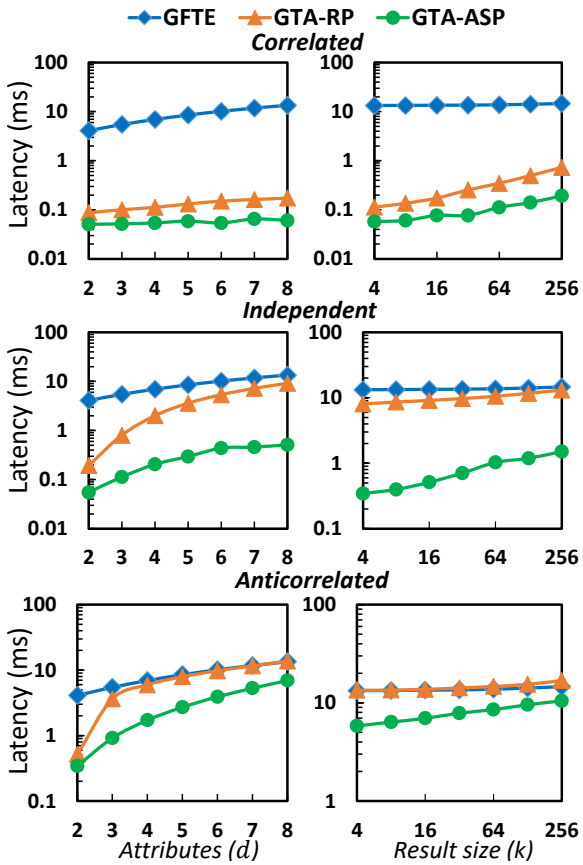

Figure 13: Measured query latency when data are accessed directly from device memory.

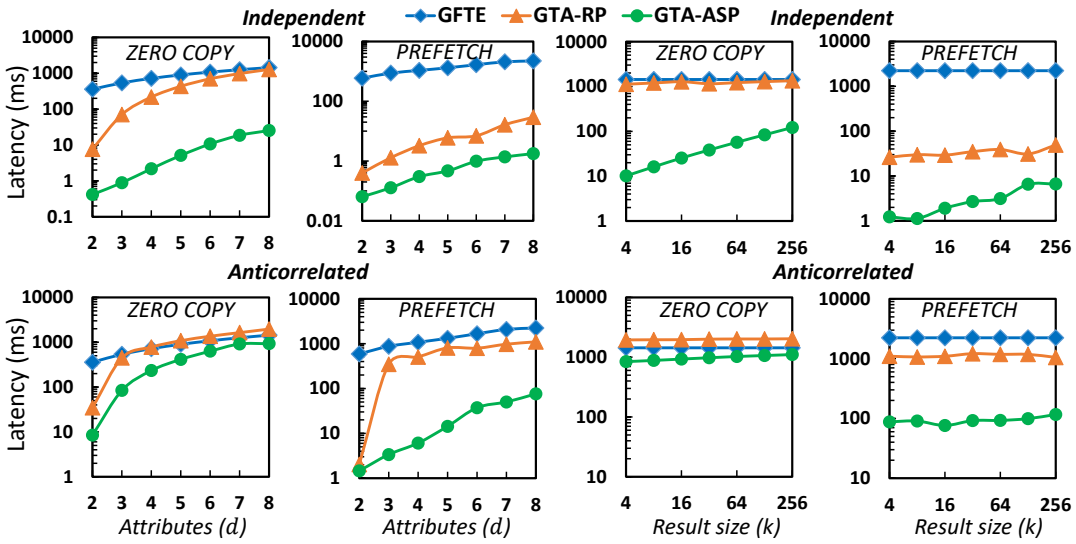

Figure 14: Measured query latency when data are accessed through host memory using different data management policies.

are diluted more with irrelevant information from other attributes. Although this behavior is inherent to ASP, its effect on execution time are minuscule, as indicated by our measurements. Henceforth, in our remaining experiments we present results using Q_0.

4.4 Device Memory Query Processing

In Fig. 13, we present the query latency for all developed algorithms on different data distributions when the base relation resides in device memory. Early stopping solutions are very efficient when processing highly correlated data because it is possible to order the tuples linearly without any ambiguity. For this reason, both GTA-RP and GTA-ASP are respectively 60× and 150× on average faster than GFTE. Independent data are somewhat more difficult to process. This happens because the likelihood of a tuple with one high scoring attribute appearing early on during processing increases dramatically. For this reason, GTA-RP ends up evaluating almost 50% of the raw data, despite them being irrelevant to the Top-k answer. This contributes to higher query latency because of the additional synchronization cost associated with the heap merge phase. GTA-ASP remains work-efficient as it relies on ASP to restrict the answer search space (i.e. variance of tuple attribute values) within a partition. This technique enables stopping earlier, reducing the associated processing workload and query latency. Anti-correlated data are the most difficult to process. Similar to before, the high variance in attribute values results in GTA-RP processing much more data that necessary (up 90% from our measurements). Hence, its query latency is almost comparable to that of GFTE for every experimental parameter. For the same reason as before, GTA-ASP is able to adapt well exhibiting 1.7× to 4× better query latency compared to GFTE.

4.5 Host Memory Query Processing

In Fig. 14, we depict the query latency measurements for all developed algorithms on different data distributions when the base relation resides in host memory. Due to lack of space, we concentrate on the independent and anti-correlated distributions since they are the most challenging to process. For independent data, data prefetching as opposed to accessing them during evaluation is beneficial for both GTA variants. However, it is occasionally worse when accessing all the tuples (i.e. GFTE). We traced this behavior back to an excessive number of GPU page faults. In fact, we observed occasionally 2× the amount of the necessary data (i.e. 16GB) being transferred across PCIe. We estimate that the GPU driver is unsuccessful in detecting temporal locality during query processing, so it resorts into moving data back and forth from the host memory. On anti-correlated data, we observe similar behavior with prefetching being the best option to speed-up processing. GTA-RP requires processing more data blocks to find the Top-k answer and outperforms GFTE occasionally. GTA-ASP performs on average 22× to 40× better than GTA-RP. The reason is because it requires fetching less data from host memory while most of it is already cached in GPU memory due to prefetching.

4.6 CPU Performance Comparison

In this section, we compare the performance of CPU Top-k selection against GPU Top-k selection. We developed two CPU methods; one that evaluates the score of all tuple for the target relation (CFTE), and another that relies on ASP and data re-ordering to enable early termination during query processing (CPU-ASP). Both methods utilize multi-threading and AVX instructions to improve processing throughput. Every CPU thread keeps track of the k-highest ranking tuples in a private priority queue, merging them only towards the end of query evaluation.

In Figure 15, we summarize the results attained for varying query parameters and data distributions. Overall, early termination methods using ASP and data re-ordering on CPU and GPU outperform solutions that rely on full evaluation because they require less tuple evaluations to compute the query answer. In fact, early termination on CPU is able to outperform the full table evaluation algorithm on GPU despite the latter being heavily optimized and while having

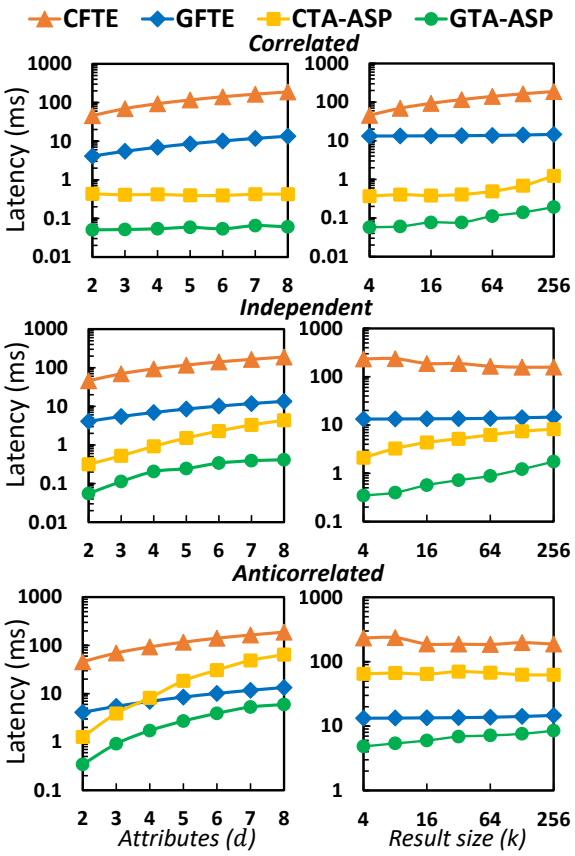

Figure 15: Comparison of measured query latency (when data are stored in device memory) against CPU-based implementations.

higher bandwidth and compute capabilities. This behavior is consistent for our experiments with correlated and independent data, showcasing improve query latency by a factor of at least 100× and 30× respectively. On anticorrelated data, it is more challenging to enable early termination because every tuple contains at least one relatively high ranking attribute. In this case, the performance of early termination drops noticeably for both CPU and GPU implementations because about 50% of all tuples are evaluated. Despite this behavior, early termination improves query latency at least 2× for both CPU and GPU solutions compared to full evaluation. In fact, subqueries referring to 2 or 3 attributes experience up to 10× lower query latency in either architecture.

5. CONCLUSIONS

In this paper, we developed the skeleton of parallel threshold algorithms that were optimized to enable GPU accelerated Top-k selection with support for early termination. We considered two different data partitioning strategies, evaluating their effectiveness on various data distributions and query parameters. Our empirical results showcased that data preordering when combined with angle space partitioning is superior in terms of tuple evaluations compared against random partitioning. Experiments with queries that were evaluated on device memory resident data showcased 2× to 100× better query latency against the state-of-the-art solution that relied on evaluating the complete relation. In addition, our experiments on queries that were evaluated on host memory resident data showcased that our methods are very effective when combined with prefetching and related caching strategies. For these experiments, we showcased 10× to 1000× better query latency as opposed to a full table evaluation algorithm. Finally, we implemented our methods on multi-core CPUs and demonstrated proportional performance improvements compared to the corresponding state-of-the-art full table evaluation solution utilizing priority queues.

6. REFERENCES

[1] I. Absalyamov, P. Budhkar, S. Windh, R. J. Halstead, W. A. Najjar, and V. J. Tsotras. FPGA-accelerated group-by aggregation using synchronizing caches. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 11. ACM, 2016.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th International Conference on Data Engineering*, pages 362–373. IEEE, 2013.

[3] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *Proceedings of the 32nd International Conference on Very Large Databases*, pages 475–486. VLDB Endowment, 2006.

[4] K. S. Bøgh, S. Chester, and I. Assent. Work-efficient parallel skyline computation for the GPU. *Proceedings of the VLDB Endowment*, 8(9):962–973, 2015.

[5] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430. IEEE, 2001.

[6] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *ACM SIGMOD Record*, volume 29, pages 391–402. ACM, 2000.

[7] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd International Conference on Very Large Databases*, pages 451–462. VLDB Endowment, 2006.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[9] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *Proceedings of the VLDB Endowment*, 4(12):1213–1224, 2011.

[10] X. Han, J. Li, and H. Gao. Efficient top-k retrieval on massive data. *IEEE Transactions on Knowledge and Data Engineering*, 27(10):2687–2699, 2015.

[11] X. Han, X. Liu, J. Li, and H. Gao. Tkap: Efficiently processing top-k query on massive data by adaptive pruning. *Knowledge and Information Systems*, 47(2):301–328, 2016.

[12] J.-S. Heo, J. Cho, and K.-Y. Whang. The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces. In *Proceedings of the 26th International Conference on Data Engineering*, pages 445–448. IEEE, 2010.

[13] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *ACM SIGMOD Record*, volume 30, pages 259–270. ACM, 2001.

[14] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11, 2008.

[15] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *Proceedings of the 8th International Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.

[16] J. Lee, H. Cho, S. Lee, and S.-w. Hwang. Toward scalable indexing for top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3103–3116, 2014.

[17] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, 2013.

[18] X. Liu, L. Golab, W. Golab, I. F. Ilyas, and S. Jin. Smart meter data analytics: systems, algorithms, and benchmarking. *ACM Transactions on Database Systems (TODS)*, 42(1):2, 2017.

[19] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-k aggregation of ranked inputs. *ACM Transactions on Database Systems (TODS)*, 32(3):19, 2007.

[20] D. Merrill. Cub, 2016. `https://nvlabs.github.io/cub/`.

[21] F. Miranda, L. Lins, J. T. Klosowski, and C. T. Silva. Topkube: A rank-aware data cube for real-time exploration of spatiotemporal data. *IEEE Transactions on Visualization and Computer Graphics*, 24(3):1394–1407, 2018.

[22] G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, pages 151–156. ACM, 2008.

[23] A. Shanbhag, H. Pirk, and S. Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1557–1570. ACM, 2018.

[24] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on gpus. In *Proceedings of the 9th International Workshop on Data Management on New Hardware*, page 4. ACM, 2013.

[25] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Information Systems*, 32(3):424–445, 2007.

[26] Y. Tao, X. Xiao, and J. Pei. Efficient skyline and top-k retrieval in subspaces. *IEEE Transactions on Knowledge and Data Engineering*, 19(8):1072–1088, 2007.

[27] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the 2008 International Conference on Management of Data*, pages 227–238. ACM, 2008.

[28] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *Proceedings of the 32nd International Conference on Very Large databases*, pages 235–246. VLDB Endowment, 2006.

[29] V. Zois. Top-k selection. `https://github.com/vzois/TopK`.

[30] L. Zou and L. Chen. Pareto-based dominant graph: An efficient indexing structure to answer top-k queries. *IEEE Transactions on Knowledge and Data Engineering*, 23(5):727–741, 2011.