Proceedings of CONF-MLA 2024 Workshop: Neural Computing and Applications
DOI: 10.54254/2755-2721/105/2024TJ0059

Optimizing Top-K query processing on GPU using data compression and pre-filtering

Yiqing Yang[1,2,†], Yan Fu[1,3,†], Shuyong Liu[1,4,*], Zhixiang Zhao[1,5], Guoyin Zhang[1,6]

[1]Department of Computer Science, Harbin Engineering University, Harbin, China

[2]yangyiqing@hrbeu.edu.cn
[3]fuyan@hrbeu.edu.cn
[4]liushuyong@hrbeu.edu.cn
[5]641zzx@hrbeu.edu.cn
[6]zhangguoyin@hrbeu.edu.cn
*corresponding author
[†]Yiqing Yang and Yan Fu contributed equally to this work.

Abstract. In large-scale data analysis, efficient Top-K query processing is critical for numerous applications in science, industry, and society. Traditional approaches often involve substantial data transfer and computational overhead, making it difficult to meet the scalability and efficiency demands of modern datasets. This paper proposes a GPU-accelerated Top-K query processing method that integrates data compression and pre-filtering techniques to address these challenges. By partitioning and compressing data on the host side, it alleviates common PCIe bottlenecks in heterogeneous computing environments. A metadata-driven pre-filtering technique further reduces the data volume processed on the GPU, significantly improving query performance, particularly when handling anti-correlated datasets. Experimental results demonstrate that this method markedly reduces data transfer and processing time, confirming its effectiveness in enhancing the efficiency and scalability of Top-K query processing compared to existing methods.

Keywords: Top-K query, CPU-GPU heterogeneous architecture, big data analysis.

1. Introduction

Data-driven decision-making is becoming increasingly vital across various domains[1]. Top-K queries, which retrieve the top k tuples from a dataset using a user-defined function, offer a solution to identifying interesting objects from large databases. These queries are widely applied in information retrieval, high-dimensional data processing, and anomaly detection [2,3]. However, processing massive datasets presents significant challenges. To address this, systems are continuously adapted for modern hardware [4], including GPUs, which excel at parallel processing and can enhance throughput and query latency [5,6]. Previous studies have shown the potential of GPUs in accelerating filtering [7] and selection operations [8], with key considerations such as coalescing memory access and minimizing thread divergence playing critical roles in optimization efforts.

The typical approach to GPU-accelerated Top-K queries involves three steps [9]: transferring data to the GPU, applying sorting or k-selection algorithms, and returning results to the CPU. Score aggregation dominates execution time, and as data grows, the cost of data movement—especially over PCIe—becomes a major bottleneck [10]. Various methods, such as early termination and hierarchical strategies, aim to reduce data transfers [11,12], but traditional techniques like random access indexing are inefficient on GPUs [10]. While these methods can reduce query latency, they often increase tuple evaluation times and struggle with complex queries involving numerous attributes [10].

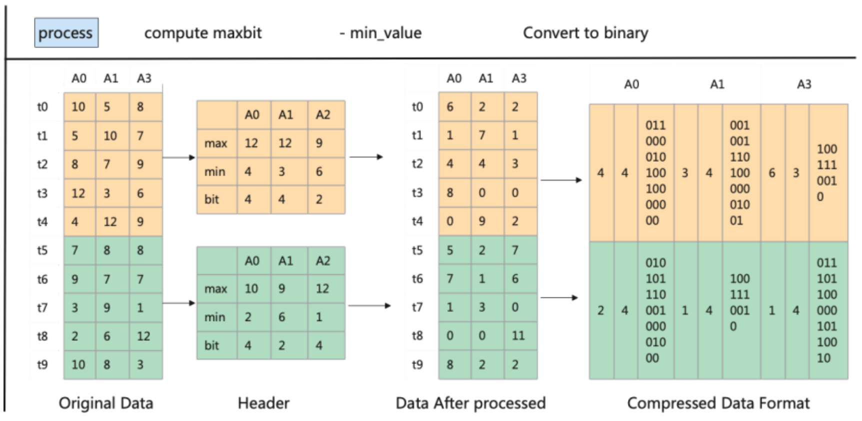
To overcome these limitations, this paper introduces a compression-based Top-K query processing algorithm. By physically partitioning and compressing data before transferring it to the GPU, the algorithm reduces data transfer time. A pre-filtering step uses block-level statistics to filter irrelevant data, improving query efficiency, especially for anti-correlated datasets [10].

2. Compression Top-K query algorithm (CTK)

The CTK algorithm efficiently handles Top-K queries on large datasets by combining data compression and GPU acceleration. Data is first partitioned and compressed on the CPU, with key statistics calculated. Compressed data and metadata are transferred to the GPU, where metadata guides the filtering process to reduce unnecessary data transfers. On the GPU, filtered blocks are decompressed, and query conditions are applied to extract relevant tuples. The Top-K result set is dynamically updated, and the final results are output. By integrating compression, GPU acceleration, and prefiltering, CTK minimizes overhead and significantly improves query efficiency.

2.1. Compression Strategy on CPU

On the CPU side, the data is processed in blocks, with each block containing multiple entries from the dataset, each consisting of several attributes (e.g., $A_1, A_2, \ldots, A_n$). For each block, the algorithm calculates key statistics for each attribute, such as the maximum and minimum values, which will be used in the prefiltering stage. The blocks are then compressed, and both the compressed data and metadata (statistical information) are stored for quick access and transmission. Figure 1 illustrates the compression process, where the minimum value is used as a reference for efficient compression.



	A0	A1	A3
t0	10	5	8
t1	5	10	7
t2	8	7	9
t3	12	3	6
t4	4	12	9
t5	7	8	8
t6	9	7	7
t7	3	9	1
t8	2	6	12
t9	10	8	3

	A0	A1	A2
max	12	12	9
min	4	3	6
bit	4	4	2

	A0	A1	A2
max	10	9	12
min	2	6	1
bit	4	2	4

	A0	A1	A3
t0	6	2	2
t1	1	7	1
t2	4	4	3
t3	8	0	0
t4	0	9	2
t5	5	2	7
t6	7	1	6
t7	1	3	0
t8	0	0	11
t9	8	2	2

Figure 1. Data compression process with minimum value reference

Assuming the tuples have three attributes and are divided into two blocks, the algorithm calculates the maximum and minimum values for each attribute. It then determines the number of bits required to represent the values within the range (e.g., a range of 8 requires 4 bits). Each value is adjusted by subtracting the minimum to reduce the range and then converted into binary based on the number of bits needed. The binary data is then merged and stored in a compressed format, reducing storage space and facilitating efficient transmission and processing.

To enhance CPU-side data compression, this paper uses multithreading and memory optimization. Data is divided into 256-sized blocks to align with GPU thread blocks, optimizing data transfer and computation. Block-based processing reduces transmission latency and balances workload. Multiple

CPU threads handle each block in parallel, calculating the maximum and minimum attribute values. The minimum value is used as a reference for compression, reducing attribute ranges and storage needs.

2.2. Pre-filtering And Decompression Strategy

On the GPU, each compressed data block first undergoes prefiltering, which significantly reduces the amount of data that needs to be decompressed and processed. As shown in Figure 2, the prefiltering stage uses statistical information stored in the metadata (e.g., the minimum value and the number of bits representing the maximum difference for each attribute) to determine whether the current data block may contain entries that meet the query criteria. If the metadata indicates that a data block may contain qualifying entries (such as B1 in Figure 2), the block is further decompressed and processed. This prefiltering step effectively reduces the number of fully decompressed data blocks, saving valuable computational resources and processing time.Asynchronous processing decouples computation from data transfer, allowing parallel execution. CUDA's asynchronous data transfer supports simultaneous data transfer and task execution without blocking. After prefiltered blocks are decompressed on the GPU, they are processed based on query conditions to determine the Top-K results. This pipelined approach avoids high-latency memory transactions and global updates by using shared memory, reducing performance overhead.

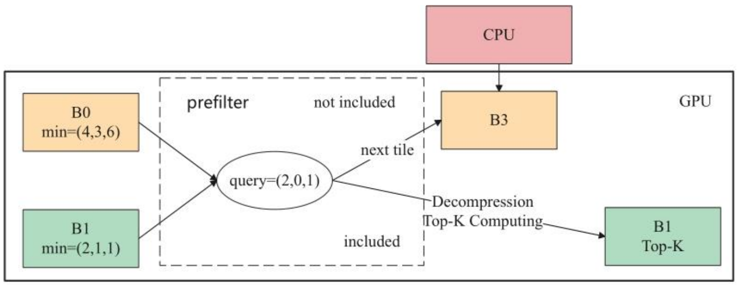


Figure 2. Prefilter Process

2.3. Top-K Query on GPU

The GPU-based Top-K query computation is designed to efficiently determine the final Top-K results from prefiltered and decompressed data blocks. The process is divided into multiple stages. First, GPU threads perform parallel score calculations for each entry in the decompressed data blocks, based on the query conditions, such as attribute ranges or composite functions. Each thread processes different entries independently, significantly speeding up score computation. Next, using the GPU's shared memory, each thread block calculates the local Top-K elements within its block using an optimized Bitonic Top-K algorithm [13]. This algorithm leverages fast shared memory access, reducing synchronization overhead and improving local computation efficiency. Each block independently determines its Top-K entries, minimizing the complexity of global sorting. Finally, the local Top-K results from different blocks are merged into the GPU's global memory. This merging is done in parallel, with tasks distributed across thread blocks to maximize GPU utilization. After several iterations, the system efficiently produces the global Top-K results, ensuring high efficiency in query processing.

3. Experimental Evaluation

In the experimental evaluation, the state-of-the-art GPU-accelerated Top-K query algorithm, GTA, combined with preprocessing techniques, was used as the baseline algorithm [10]. The hardware platform includes a single NVIDIA TU102 GPU and a system with an Intel Xeon Silver 4208 CPU (Skylake architecture, 8 cores, supporting 2 hardware threads). All algorithms were implemented and executed using C++ and the CUDA 11.0 environment.

The experimental design follows methods from related studies [12,14,15], using synthetic datasets with three data distribution types (correlated, independent, and anti-correlated) generated by a standard dataset generator. The dataset includes relational data with 8 attributes and 512 million tuples, totaling about 16 GB. The experiments tested the impact of varying the number of query attributes ($d \in [2,8]$) and Top-K size ($k \in [4,8,16,32,64,128,256]$) on algorithm performance. Additionally, for the independent distribution dataset, execution time was measured in detail for different preference vectors, with results shown in Table 1.

Table 1. Query weight values based on preference vector.

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)

3.1. *Preprocess Comparison*

Figure 3 shows the initialization phase cost, focusing on data preprocessing time. The results indicate that CTK incurs 1.6 to 1.9 times higher initialization overhead compared to GTA. However, despite this, CTK delivers about 10 times faster query latency than GTA, as explored in later chapters.

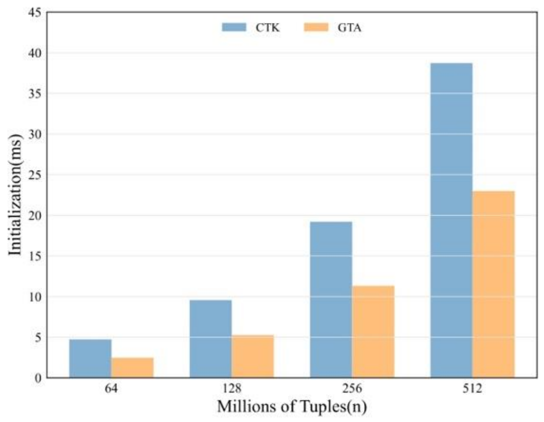


Figure 3. Latency of data preprocess to ready for to GPU

CTK's compression strategy reduces data transferred from the CPU to the GPU by about 95%, completing the transfer in a single PCIe pass. In contrast, GTA transfers all raw data to the GPU for preprocessing, sends it back to the CPU for filtering, and then returns it to the GPU for query processing, causing significant communication overhead and processing delays. While CTK's CPU preprocessing time is longer, it minimizes overall computation time by reducing PCIe transfers. For large-scale datasets, CTK's design effectively cuts transmission overhead and boosts processing performance, ensuring efficient data preprocessing and query response.

3.2. *Variable Preference Vectors*

Figure 4 shows the performance of CTK and GTA across varying attribute numbers. While CTK's query latency increases with more attributes, it remains stable, showing strong adaptability. For simpler queries (e.g., Q0, Q1, Q2), CTK is about 10 times faster than GTA, thanks to its prefiltering stage that quickly eliminates irrelevant data. On the other hand, CTK's prefiltering becomes less efficient due to more attribute conditions, leading to higher computational overhead and increased latency.

Figure 5 shows the performance trend of CTK and GTA with varying query result sizes (K values). As K increases, query latency rises, but CTK maintains a smoother increase and is about 10 times faster than GTA for all K values. CTK's superior performance comes from its compression and prefiltering strategies, which reduce the data volume transferred and processed. By filtering irrelevant data early, CTK minimizes transmission and computational overhead, optimizing resource use, especially in large-scale queries. In contrast, GTA faces higher latency as K increases, making it less efficient for large datasets. CTK's consistent performance makes it ideal for tasks like recommendation systems and real-time data analysis.

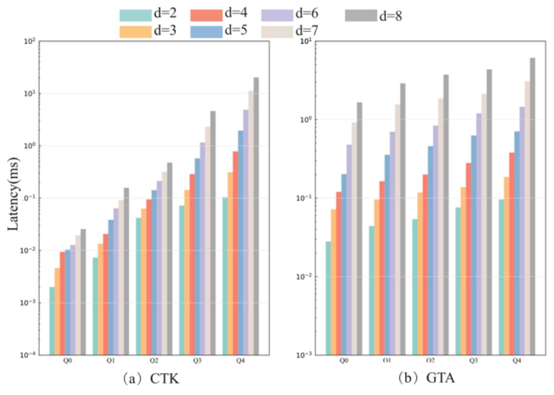


Figure 4. The impact of the number of attributes on query latency.

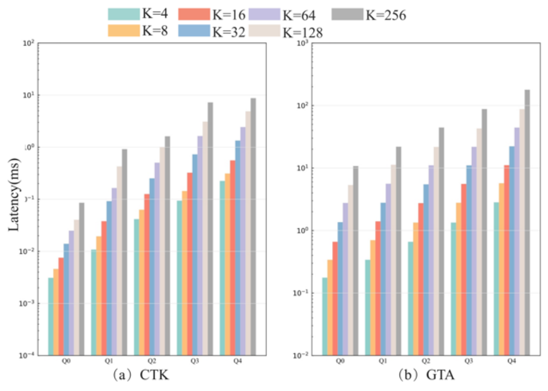


Figure 5. The impact of K value on query latency.

3.3. *Device Memory Query Processing*

Figure 6 shows the query latency of different algorithms in the GPU device memory for varying data distributions. This not only reflects the efficiency of each algorithm in utilizing GPU resources but also, through ablation experiments, demonstrates the stability and adaptability of the algorithms.

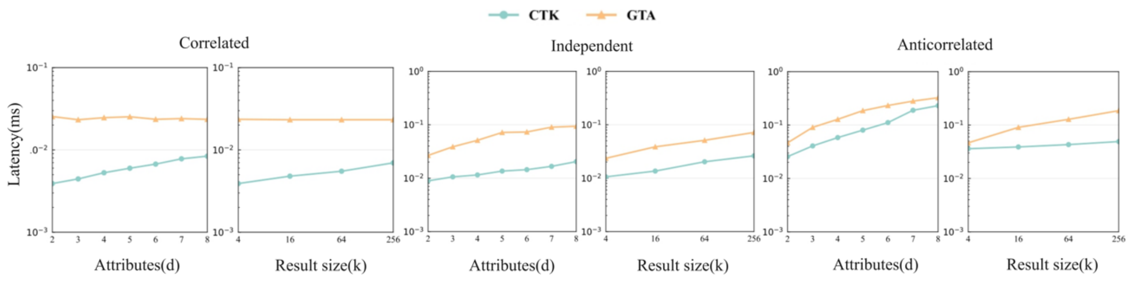


Figure 6. The impact of data distribution on device memory on query latency

When handling correlated and independent data, the CTK algorithm achieves exceptionally high efficiency, largely due to the sorting operation during compression, making it about 10 times faster than GTA in average processing speed. For these data types, CTK's compression and prefiltering mechanisms work efficiently to filter and process data, significantly reducing processing time. However, with anti-correlated data, CTK faces challenges. The large variation in attribute values lowers compression efficiency, and the prefiltering mechanism becomes less effective, requiring multiple filtering passes due to the significant differences in correlation. Despite these challenges, CTK remains approximately 1.28 times faster than GTA, demonstrating solid performance across diverse data distributions. CTK also shows high stability and adaptability when processing varying numbers of attributes and K values. As the K value increases, CTK's query latency grows more gradually, particularly in independent and anti-correlated data scenarios. This stability is attributed to the Bitonic Top-K algorithm, which helps CTK maintain efficiency even with varying K values. Overall, CTK not only excels with correlated and independent data but also shows strong adaptability in more complex, anti-correlated data scenarios.

4. Conclution

In modern scientific, industrial, and societal domains, data-driven decision-making is crucial. Top-K queries, widely used in information retrieval, anomaly detection, and visualization, help identify key objects in large databases. However, handling massive datasets presents challenges, including frequent PCIe transfers and inefficiencies with anti-correlated data. We propose the CTK algorithm, which processes Top-K queries efficiently by using data compression and GPU acceleration. Compressing data before GPU transmission reduces transfer volume by 95%, and a prefiltering step eliminates irrelevant data blocks, optimizing resource use. Experiments show CTK outperforms existing GPU-based algorithms by 1.28 to 10 times across various data distributions. While CTK offers major improvements, future work could enhance compression and prefiltering for complex datasets, and explore hybrid CPU-GPU methods to further boost performance. CTK represents a significant step forward in Top-K query processing, especially for large-scale data environments.

References

[1] ABADI D, AILAMAKI A, ANDERSEN D, et al. (2022). The Seattle report on database research. Communications of the ACM, 2022, 65(8): 72-79.
[2] NYCHIS G, SEKAR V, ANDERSEN D G, et al. (2008). An empirical evaluation of entropy-based traffic anomaly detection. Proceedings of the 8th ACM SIGCOMM conference on Internet measurement, 2008: 151-156.
[3] MIRANDA F, LINS L, KLOSOWSKI J T, et al. (2017). TOPKUBE: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data. IEEE Transactions on visualization and computer graphics, 24(3): 1394-1407.
[4] FANG J, MULDER Y T B, HIDDERS J, et al. (2020). In-memory database acceleration on FPGAs: a survey. The VLDB Journal, 29(1): 33-59.
[5] ABSALYAMOV I, BUDHKAR P, WINDH S, et al. (2016). FPGA-accelerated group-by aggregation using synchronizing caches. Proceedings of the 12th International Workshop on Data Management on New Hardware, 2016: 1-9.
[6] KALDEWEY T, LOHMAN G, MUELLER R, et al. (2012). GPU join processing revisited. Proceedings of the Eighth International Workshop on Data Management on New Hardware, 2012: 55-62.
[7] SITARIDI E A, ROSS K A. (2013). Optimizing select conditions on GPUs. Proceedings of the Ninth International Workshop on Data Management on New Hardware. 2013: 1-8.
[8] BØGH K S, CHESTER S, ASSENT I. (2015). Work-efficient parallel skyline computation for the GPU. Proceedings of the VLDB Endowment, 8(9): 962-973.
[9] ALABI T, BLANCHARD J D, GORDON B, et al. (2012). Fast k-selection algorithms for graphics processing units. Journal of Experimental Algorithmics (JEA), 17: 4-1.
[10] ZOIS V, TSOTRAS V J, NAJJAR W A. (2019). Efficient main-memory top-K selection for multicore architectures. Proceedings of the VLDB Endowment, 13(2): 114-127.
[11] HAN X, LI J, GAO H. (2016). Efficient Top-k Retrieval on Massive Data. 2016 IEEE 32nd International Conference on Data Engineering (ICDE), 2016: 1496-1497.
[12] HEO J S, CHO J, WHANG K Y. (2010). The Hybrid-Layer Index: A synergic approach to answering top-k queries in arbitrary subspaces. 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), 2010: 445-448.
[13] SHANBHAG A, PIRK H, MADDEN S. (2018). Efficient Top-K Query Processing on Massively Parallel Hardware. Proceedings of the 2018 International Conference on Management of Data, 2018: 1557-1570.
[14] ZOU L, CHEN L. (2011). Pareto-Based Dominant Graph: An Efficient Indexing Structure to Answer Top-K Queries. IEEE Transactions on Knowledge and Data Engineering, 23(5): 727-741.
[15] MAMOULIS N. (2007). Efficient Top-k Aggregation of Ranked Inputs. ACM Transactions on Database Systems, 32(3): 47.