A wait-free output data structure for GPU-based streaming query processing

Discussion paper

Claudio Silvestri[1], Francesco Lettich[1], Salvatore Orlando[1]
Christian S. Jensen[2]

[1] Università Ca' Foscari Venezia, Italy
{lettich,orlando,silvestri}@dais.unive.it
[2] Aalborg University, Denmark
csj@cs.aau.dk

Abstract. The performance of GPU-based algorithms can be reduced significantly by contention among memory accesses and by locking. We focus on high-volume output in GPU-based algorithms for streaming query processing: a very large number of cores process input streams and simultaneously produce a sustained output stream whose volume is sometimes orders of magnitude larger than that of the input streams. In this context, several cores can produce results simultaneously that must be written in the output buffer according to some order and without conflicts with other writers. To enable this behavior, we propose a wait-free bitmap-based data structure and a usage pattern that combine to obviate the use of locks and atomic operations. In our experiments, where the GPU-based algorithm considered is otherwise unchanged, the introduction of the new wait-free data structure entails a performance improvement of one order of magnitude.

1 Introduction

The efficiency of parallel algorithms, and in particular those tailored for massively parallel computing architectures, depends on the degree of parallelism that can be attained, which depends on how the available resources are used. Contention in memory accesses results in computing cores being under-utilized because they wait for access to resources. Thus, many optimization efforts in parallel computing aim to avoid contention in memory access. This is particularly important in the setting of massively parallel GPUs [1], where several thousands of cores may compete for a write lock and where many of them may need to write at the same time due to data parallelism.

In this paper, we focus on high-volume output in stream processing: a very large number of cores process input streams and simultaneously produce a sustained output stream whose volume is sometimes orders of magnitude larger than that of the input streams. In this context, one source of contention is related to the construction of the output buffer: several cores may produce results that must be written in a particular order in the output buffer. Since the amount of simultaneous writes is unknown, write locations for each core cannot be predetermined. Even if atomic accesses and locks can be used to avoid race conditions and result loss, their use imply that some threads must wait for their turn to write. Having some thousands of cores and many more concurrent threads yields a high probability of contention and a large number of threads that are waiting for locks. A more efficient solution is to let every core write in a different position and then use stream compaction to keep only the relevant output. Unfortunately, this approach is particularly memory demanding, since space is reserved both for threads that produce output and threads that do not write to memory. Further, in case more results are produced for each query received from the input stream, we could be interested in having all the related output represented together, for logical reasons or to avoid redundant information. Since it is possible, and quite likely, that threads do not write their results in the expected order, a final sorting of the buffer would be needed.

We observe that, in case query results are subsets of a finite domain, the stream to be compacted can be represented as a sequence of Boolean values, corresponding to the presence or absence of a particular element in a query results. Moreover, this finite set may have a limited cardinality, either originally or due to indexing or domain restrictions.

Guided by these thoughts, we use a bitmap-based data structure for the compaction of Boolean streams that capture set membership. From those streams that represent results of filtering predicate evaluations, we produce a stream of item identifiers, grouped by set identifiers, containing only the references to those elements that are contained in the corresponding set. This data structure can be used to fill the output buffer in streaming query processing in a wait-free manner.

In experiments [6], we applied the proposed data structure for streaming range queries over moving objects. Without changing any other aspect of the GPU-based algorithm, the use of the new wait-free data structure gives a performance improvement of one order of magnitude compared to using a more straightforward approach for output production. Due to constraints on the length of the paper, we limited the paper's coverage of experiments to a single use case. In future works, we will evaluate the benefit of the proposed design pattern in different scenarios such as streaming queries over streams of social media updates and streaming similarity queries over real-time financial data.

The remainder of the paper is organized as follows. Section 2 summarizes some GPU related terminology, describes the problem, and presents a straightforward solution; Section 3 describes the proposed data structure and usage pattern; Section 4 applies the proposed data structure to streaming range queries over moving objects and discusses experimental results. Finally, Section 5 summarize the paper.

2 Preliminaries

2.1 GPU Terminology

The GPU terminology used in the paper refers to NVIDIA CUDA [4], although different programming frameworks and architectures adopt similar solutions with slightly different names. A GPU consists of an array of n_{SM} *multithreaded streaming multiprocessors* (SMs), each with n_{core} cores, yielding a total number of $n_{SM} \cdot n_{core}$ cores. Each SM is able to run *blocks* of *threads*, namely *data-parallel tasks*, with the threads in a block running concurrently on the cores of the SM. Since a block typically has much more threads than the cores available in a single SM, only a subset of threads, called *warps*, can run in parallel at a given time instant. Each warp consists of sz_{warp} *synchronous, data parallel threads*, executed by an SM according to a SIMD-like paradigm.

2.2 Problem Description

The paper presents a design pattern that allows a class of streaming algorithms for GPUs to avoid the use of locks and atomic memory accesses when writing to their output stream buffer. In the following, we therefore describe a generic problem set that is common to all of the algorithms in the class.

The algorithms in this class:

- process an input stream of queries, represented by a sequence $IS = q_1, \ldots, q_i, \ldots$ where $q_i \in Q$, the input query domain;
- compute a set of results $r(q_i) = \{o | \theta(q_i, o), o \in O\}$ for each query q_i, where θ is a logical predicate and O is the output domain;
- produce a stream of results $OS = (q_1, r(q_1)), \ldots, (q_i, r(q_i)), \ldots$ where $q_i \in IS$.

Moreover, the input stream is partitioned into batches $B = b_1, \ldots, b_i, \ldots$, where $b_i = \{q_j, q_{j+1}, \ldots\}$, whose queries are processed in parallel. In general, θ could be time dependent and yield, for the same query, different result sets in different batches. Further, computation may depend on system state and contextual information (as in the use case described in Section 4). To simplify the notation, however, we omit those dependencies.

Batching is common in GPU algorithms, since high parallelism on GPUs can only be attained when an adequate amount of data is available for processing. Even if batching can introduce some latency in stream processing, for a fixed batch size, the duration of each batch decreases as the stream rate increases. Thus, this kind of latency is not an issue for important applications requiring high rate stream processing.

In Section 4, we instantiate this generic problem and consider a more specific one for the experiments: streaming spatial range queries. In this case, input queries are rectangles, and the predicate θ represents containment of points in rectangles.

2.3 A Solution Based on Filtering and Synchronized Output

It is relatively easy to devise a simple strategy to write output directly to a GPU's global memory during query computation. Each GPU thread is in charge of a distinct query, and all threads concurrently enqueue the output to contiguous memory locations, as soon as a new element that satisfies the predicate θ is detected. The query results thus constitute an un-ordered list of pairs $(query_{ID}, o_{ID})$, where $query_{ID}$ and o_{ID} identify a query q and an output element o such that $\theta(q, o)$ is true.

This approach has two main drawbacks: the need for synchronization among threads (in each block) in order to flush the results to global memory in a coherent manner, and the format of the output, which is not compressed, thus increasing the I/O costs.

3 Design Pattern

To address the output requirements of high rate streaming query processing algorithms, as described in Section 2.2, we adopt a *multi-step* strategy that is a common choice in the GPGPU field [3, 5]. We split the result production in two phases: during the first one (*filtering*), we compute the results and enter them into an intermediate, wait-free, data structure; then, during the second one (*decoding*), the intermediate data structure is decoded to produce the final result.

3.1 Output Data Structure

The purpose of the intermediate data structure is to contain the logical values of the predicate θ for each pair composed of a query from the input stream ($q_i \in I$) and a possible output value from the output domain ($o \in O$).

We apply three criteria to select a data structure and access pattern to use:

wait-freedom: to ensure that the output of each thread is independent, both true and false values are represented. Thus, each thread can write to a predetermined location, according to the query it is currently processing, and exclusive access is granted without locks or atomic operations.

locality: to achieve memory access locality, the results produced by different threads are interlaced. In particular, to benefit from coalescing, consecutive threads handle consecutive queries and write to consecutive memory locations, thanks to an interlaced data layout.

bitwise representation: to reduce the memory footprint, each Boolean value $\theta(q_i, o)$ is represented by a single bit.

In the proposed approach (see the next subsections), each input stream query q_i is processed by a different thread to evaluate θ for all possible $o \in O$ (i.e., to process the query q_i). According to the above design criteria, the resulting bitmap depicted in Figure 1 is written line by line, where columns correspond to the output of each thread (results for query q_i). Since it would be inefficient to write individual bits to memory immediately, the elements in each cell are integers that represent the results for 32 consecutive output domain elements. Thus, each line represents the results for all the queries in a batch limitedly to 32 elements of the output domain.

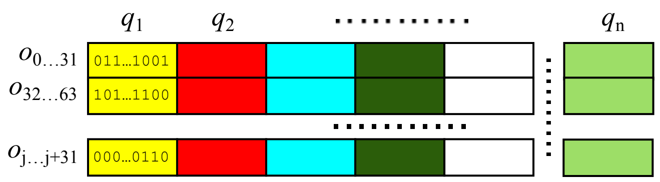

Fig. 1: Bitmap-based output data structure.

From memory footprint perspective, this representation becomes advantageous when the selectivity of queries exceeds $1/32$ with respect to the output domain. In case indexes are used, the same consideration holds, but the selectivity is computed with respect to the number of distinct elements in each index leaf (see Section 3.4).

3.2 Filtering

The goal of this phase is to produce a *bitmap* containing all the results of the θ predicate evaluation for a specific batch.

The query processing specifics are problem dependent. In general, however, when each input query q_i is processed, the predicate $\theta(q_i, o)$ is evaluated once for each output domain element $o \in O$ (Section 3.4 discusses on the use of indexes), and the corresponding bits are set or cleared.

All the evaluations of $\theta(q_i, o)$ predicates for the same q_i value are assigned to the same GPU thread. Due to the proposed data structure, each thread writes a 32-bit integer once every 32 evaluations of the θ predicate. Since this happens simultaneously for all threads in the same warp, and since queries IDs are assigned to threads according to the thread's IDs, consecutive threads will write to consecutive memory locations. This allows memory access coalescing: several memory accesses are combined in a single memory transaction (e.g., 128 bytes, $32 \cdot 32$ bits, on a K20 device [4]). The size of the bitmap for a batch b containing $|b|$ input elements is therefore $|b|\ |O|$, but this can be reduced in case indexes or domain partitioning techniques are adopted (Section 3.4).

3.3 Decoding

After filtering, the bitmap data structure contains the logical values of every θ predicate evaluation, and the results related to different queries are interlaced in groups of 32 output elements (the cells in Figure 1). To simplify the transformation of the bitmap to lists of query results, the interlaced bitmaps are first linearized: in the resulting layout, the bit-vectors associated with every query have their words arranged consecutively in memory, which enable read coalescing during the decoding phase. This transformation can be done through a fairly simple GPU kernel.

Then, for each query, its linearized bit-vector is decoded by a group of 32 threads always scheduled concurrently on the same SM (a warp). Due to their scheduling, such threads are able to access the same SM shared memory. They use this opportunity to read the input data in sequence to perform a collaborative count of the number of results preceding each output element that will be written (to determine writing positions) and, finally, to write the output element identifiers that constitute the query results. Coalesced memory access can be exploited both for reads and writes to global memory since they happen in sequence, with consecutive threads writing to consecutive memory locations. Finally, no locking is used since write positions are distinct for each thread and are predetermined before write time.

3.4 Optimizations

The abstract method described above can be improved in several ways. Some of them are commonplace in GPU computing [4], for example the overlap of memory transfers and computations can be used in the decoding phase to mask the cost of moving final results to CPU memory. Others are common in specific domains, corresponding to specific predicates θ. For example, in case we consider range queries over a set of points, input queries would be rectangles, and the predicate θ would represent containment of points in rectangles. Using some kind of spatial indexing is an obvious option, but the way it is used and computed on GPUs strongly affects the choice of which index to use [6].

In addition to reducing the number of predicate evaluations, indexes can be used to split the workload into independent partitions to be processed in parallel. When restricting the input domain to one of the partitions causes a restriction of the output domain, it is possible to process partitions independently and thus benefit from the use of smaller bitmaps thanks to the reduced cardinality of the output domain [6].

4 Use Case: Streaming Spatial Range Queries

4.1 Use case description

To assess the performance that can be achieved by using the proposed design pattern, we focus on a specific instance of the generic problem described in Section 2.2, where moving spatial point objects repeatedly issue range queries to find nearby objects [7]. This use case has applications in Massively Multiplayer Online Games and in simulations, where agents may affect the behaviors of other agents within a given range [2, 8]. Further, if we consider the particular case in which queries are static and only position updates are streaming, this example matches the Twitter streaming API, in which data collection applications may filter geo-tagged Tweets by defining a set of rectangles that represent regions of interest.

Processing of streams of spatial range queries and position updates is described in greater details in previous work [6], where several domain-specific techniques and optimizations are covered. For example, the spatial data is organized according to a grid index, used both for filtering and for workload partitioning. Further, bitmaps representing the query results in an index cell are optimized to represent only the points contained in that cell. This is an application of one of the optimizations described in Section 3.4, enabled by the output domain restriction induced by the index. Next, we focus our attention only on the effectiveness of the wait-free data structure and consider this particular problem just as one use case.

4.2 Experimental Setup and Findings

The experiments are conducted on a PC equipped with an Intel Core i3 550 CPU (at 3.2 GHz) with 4 GB of RAM and an NVIDIA GTX 560 GPU (Ubuntu 12.04, GCC 4.6.3, CUDA Toolkit 4.2). We exploit a publicly available framework [7] for both workload generation and testing. The reader can refer to our previous work [6] for a complete description of test parameters and additional experimental results.

We proceed to compare the optimized algorithm, denoted by $\mathsf{GPU_{Opt}}$, which exploits bitmaps during the filtering phase, with a baseline version, denoted by $\mathsf{GPU_{BLine}}$, which uses thread synchronization to correctly enqueue the list of results. Since the only difference between the two approaches is the use of the proposed data structure, their comparison can be used to assess the data structure's effectiveness.

Due to the space limitation, we discuss only one experiment: repeatedly using a uniformly distributed synthetic data stream (700k moving objects, every object sends a query per batch), we change the query range size, thus indirectly affecting the size of the result bitmaps.

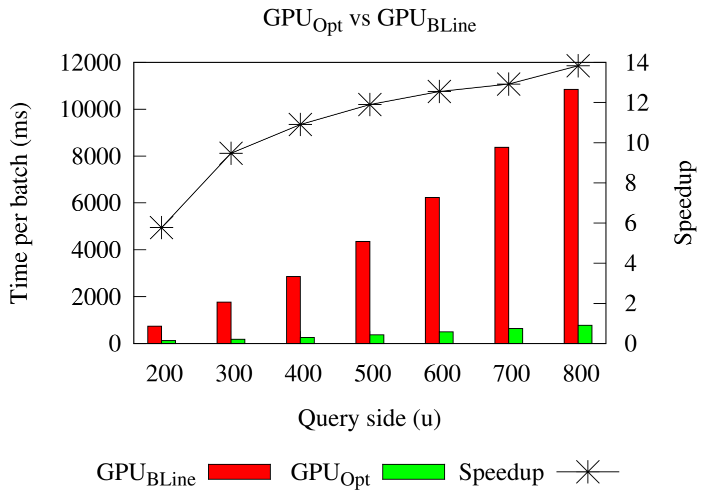

Fig. 2: Running time per batch: varying query area

Figure 2 shows how the execution time changes when varying the query range. We observe that for small ranges, $\mathsf{GPU}_{\mathsf{Opt}}$ is approximately 5 times faster than $\mathsf{GPU}_{\mathsf{BLine}}$. As the range increases also the number of results per query increases; consequently, the benefit of the wait-free approach is more evident, and $\mathsf{GPU}_{\mathsf{Opt}}$ is approximately 14 times faster than $\mathsf{GPU}_{\mathsf{BLine}}$. We highlight that an increase of the average query size corresponds, due to the peculiarities of the algorithm [6], also to an increase of the grid cell size, which in turn results in more containment check and in an increase of the bitmap size. Despite this fact, the use of the bitmap-based data structure is increasingly better than the output buffer management based on atomic operations when the query range increases. Other experiments, not reported in details, highlight that storing the results of the baseline approach (as pairs) requires more memory that storing both the intermediate bitmaps and the final results of the optimized approach (as lists of objects for each query identifier). Further, if storing the final result is not needed, the execution time per tick can be reduced by approximately 50%. Indeed, the execution times for the filtering and the decoding phases are almost the same [6].

5 Conclusions

We present a new perspective on an output production technique [6] based on a wait-free bitmap-based data structure, discussing its use in a more general context. This broader setting covers any stream processing scenario in which queries are processed at a sustained rate to produce a high-volume output stream, in which results for the same query are grouped together, and in which the output domain has limited cardinality (e.g., 10k elements), either globally or per leaf in case an index is used. We consider the original problem [6] as an instance of the general one, and we present a preliminary evaluation of the performance gain achievable by using the proposed data structure.

In future work, we plan to extend this evaluation to other scenarios to assess how the benefits given by the wait-free data structure depend on the characteristics of the output stream.

Acknowledgment

Research partially supported by the European Union under the ESF/COST COST Action IC0903: Knowledge Discovery from Moving Objects (MOVE, http://www.move-cost.info).

References

1. D. Cederman, B. Chatterjee, and P. Tsigas. Understanding the performance of concurrent data structures on graphics processors. In *Proc. International Conference on Parallel Processing*, pages 883–894, 2012.
2. Joshua M. Epstein. Agent-based computational models and generative social science. *Complexity*, 4(5):41–60, 1999.
3. D. Merrill and A.S. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.
4. NVIDIA. *CUDA C Programming guide 7.0.* 2015.
5. S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for GPU computing. In *Proc. of ACM SIGGRAPH Symposium on Graphics Hardware*, pages 97–106, 2007.
6. C. Silvestri, F. Lettich, S. Orlando, and C. S. Jensen. Gpu-based computing of repeated range queries over moving objects. In *Proc. Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 640–647, 2014.
7. B. Sowell, M. V. Salles, T. Cao, A. Demers, and J. Gehrke. An experimental analysis of iterated spatial joins in main memory. *Proc. VLDB Endow.*, 6(14):1882–1893, 2013.
8. F. Tauheed, T. Heinis, and A. Ailamaki. THERMAL-JOIN: A scalable spatial join for dynamic workloads. In *Proc. of ACM SIGMOD Conf.*, 2015.