GPU-Accelerated Quantification Filters for Analytical Queries in Multidimensional Databases[*]

Peter Tim Strohm, Steffen Wittmer, Alexander Haberstroh, and Tobias Lauer

Jedox AG
Bismarckallee 7a
D-79106 Freiburg, Germany
{peter.strohm,steffen.wittmer,alexander.haberstroh,
tobias.lauer}@jedox.com

Abstract. In online analytical processing (OLAP), filtering elements of a given dimensional attribute according to the value of a measure attribute is an essential operation, for example in top-*k* evaluation. Such filters can involve extremely large amounts of data to be processed, in particular when the filter condition includes "quantification" such as ANY or ALL, where large slices of an OLAP cube have to be computed and inspected. Due to the sparsity of OLAP cubes, the slices serving as input to the filter are usually sparse as well, presenting a challenge for GPU approaches which need to work with a limited amount of memory for holding intermediate results. Our CUDA solution involves a hashing scheme specifically designed for frequent and parallel updates, including several optimizations exploiting architectural features of Nvidia's Fermi and Kepler GPUs.

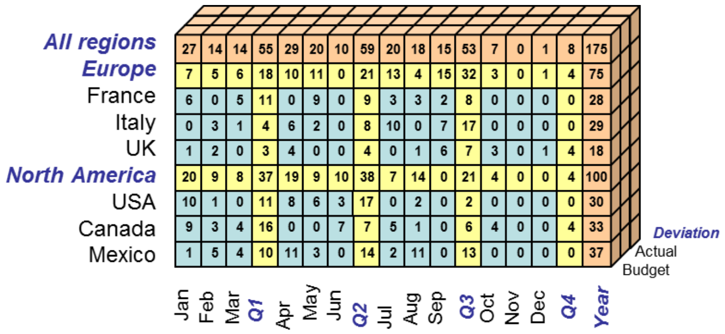
1 Introduction

Multidimensional databases allow users to analyze data from different perspectives by applying OLAP operations such as roll up, drill down, or slice and dice. Handling very large dimensions with hundreds of thousands or even millions of elements requires filter methods in order to show users only those data that are relevant to their analytical needs. Apart from simple filters based on characteristics of the elements themselves (e.g. element names, hierarchy levels, etc.), more advanced methods can filter elements based on conditions regarding the values of the (numeric) measures associated with the elements and stored in the OLAP cube.

An example from the sales analytics domain would be to show only those products of which at least *k* units were sold in *any* store (or, alternatively: *all* stores) during a certain time. Filters involving such an ANY or ALL requirement are called *quantification filters* and are the central topic of this paper. Other filters might require that an aggregated value such as SUM, MIN, MAX, or AVG fulfills a condition; those are called aggregation filters.

[*] Parts of the research described in this paper were presented by the authors at Nvidia's GPU Technology Conference in San Jose, CA (USA) in March 2014.

The calculation of all such filters involves the processing of very large amounts of data. Essentially, for each of the input elements to be filtered, an (n-1)-dimensional slice of an n-dimensional cube must be scanned with respect to the given condition(s). The slices may consist of aggregated values, which have to be computed first (cf. Fig. 1). While research on traditional relational approaches (ROLAP) has often concentrated on the partial pre-computation of aggregate values and the question which subset of aggregates should be pre-calculated to be stored in a given amount of available memory, in recent years a multitude of in-memory database systems have been emerging, many of them non-relational but using specialized multidimensional data structures [11, 12, 13]. Those systems store only base data (Fig. 2) and compute all calculated values on demand, i.e. at query time. This obviates the need for pre-computation but requires algorithms and computational resources for extremely fast aggregation in order to deliver results to users in (near) real time to support interactive operations such as slicing, dicing, roll-up and drill-down.



	Jan	Feb	Mar	Q1	Apr	May	Jun	Q2	Jul	Aug	Sep	Q3	Oct	Nov	Dec	Q4	Year
All regions	27	14	14	55	29	20	10	59	20	18	15	53	7	0	1	8	175
Europe	7	5	6	18	10	11	0	21	13	4	15	32	3	0	1	4	75
France	6	0	5	11	0	9	0	9	3	3	2	8	0	0	0	0	28
Italy	0	3	1	4	6	2	0	8	10	0	7	17	0	0	0	0	29
UK	1	2	0	3	4	0	0	4	0	1	6	7	3	0	1	4	18
North America	20	9	8	37	19	9	10	38	7	14	0	21	4	0	0	4	100
USA	10	1	0	11	8	6	3	17	0	2	0	2	0	0	0	0	30
Canada	9	3	4	16	0	0	7	7	5	1	0	6	4	0	0	4	33
Mexico	1	5	4	10	11	3	0	14	2	11	0	13	0	0	0	0	37

Fig. 1. OLAP cube with base cells (light blue) and aggregated cells (yellow and orange)

Hence, it is essential that aggregates along the relevant dimensions of a data cube can be calculated as efficiently as possible.

General-purpose computing on graphics processing units (GPGPU) is a trend used in many computing domains with the potential for tremendous speedups through the massively data-parallel computation available on such devices. During the past decade, a growing number of database operations and applications have been examined in this respect, with relatively mixed results [4, 5, 8]. One reason for this diversity lies in the nature of database operations, which are more often bandwidth-bound rather than computation-bound. In addition, if GPUs are used as pure co-processors, the bottleneck is typically the transfer of the data to and from GPU memory, rather than the computation itself [4]. As the vast majority of research contributions try to accelerate the computation of individual relational operators [2, 4, 5, 8], this problem remains inherent. Only in few approaches – such as [7], [10] and our own system – the graphics processors form an integral part of the database system design and GPU memory is also explicitly used for data storage.

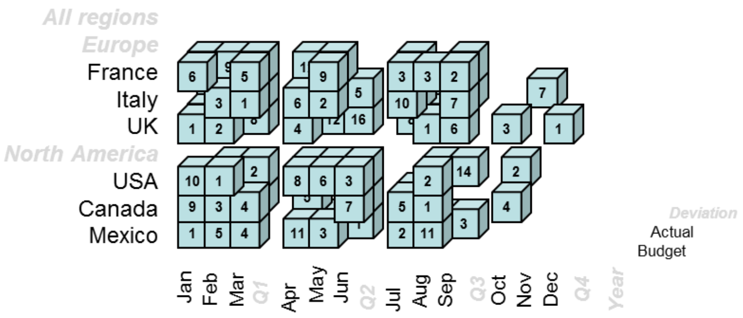


Fig. 2. Only base cells are stored with in-memory databases. All aggregates are calculated on demand.

The main contribution of this paper is a massively parallel approach to the computation of quantification filters, with an implementation for graphics processing units using Nvidia's CUDA. Since the actual filter is applied on a pre-aggregated area (or, sub-cube), the first part of our contribution is an efficient parallel aggregation method for large (and possible sparse) areas of aggregated cells, also implemented in CUDA. All of the described algorithms have been integrated in the commercially available Jedox OLAP Server [13].

2 Motivation: Computation of Quantification Filters

A quantification filter as described above can be seen as a two-stage process. In a first step the values that need to be inspected are computed and collected such that each element of the filter dimension is associated with a corresponding slice of the OLAP cube. For instance, assume the filter dimension consists of the products sold by a company. In this case, each product might be associated with the cube slice containing sales figures of this product (over a time span) for each individual store, but totaled over all other dimensions. Obtaining all the slices involves a lookup and pre-aggregation procedure resulting in an *area* or "grid" of values, whose parallel computation is described in section 3.

In the second step, explained in section 4, the filter condition is checked for each element of the filter dimension, i.e. for the values in each slice of the area computed in step 1. For instance, the filter condition might require that the total sales for a product in ALL stores (i.e. in every single store) are greater than US$ 100,000. In this case, all values in the slice have to be inspected to verify the condition. Note that each slices can be an up to (n-1)-dimensional sub-cube of the original n-dimensional OLAP cube. Hence, both of the above steps can become computationally expensive, the first due to its potentially large output, the second due to the size of its input (the output of the filter is limited by the cardinality of the filter dimension).

3 Parallel Computation of Aggregate Areas

It has been shown in previous work that individual OLAP aggregations as well as reasonably sized bulks of aggregations can be accelerated significantly by using GPUs [4]. However, in quantification filters a large structured area of aggregated values, potentially consisting of billions of cells, must be computed. In OLAP scenarios, such areas are typically very sparse, meaning that a majority of the values are zero because no base data exist to enter the aggregate.

If such values are calculated by a standard target-driven approach such as the one described in [4] and summarized in section 3.1, many unnecessary queries will be launched. Added up, these queries can significantly increase overall response time.

We therefore propose an alternative, source-driven aggregation approach designed for large areas of aggregated values, in section 3.2.

3.1 Target-Driven Aggregation

In target-driven approaches, the computation takes the targets as a starting point. A variable is assigned (i.e. memory is allocated) for each target cell to hold its output value (and any intermediate values during the computation). Then, for any source cell to be examined, the aggregation algorithm would typically check whether that source contributes to the target, and add the source value to the current aggregate value, taking into account possible weight factors defined on dimensional hierarchies. For checking source cells, a parent-to-child map of dimensional hierarchies is required, which provides, for each coordinate in a target path, the coordinate(s) of all matching source cells and corresponding weights.

Such an approach works very well for individual targets and can also nicely be parallelized with GPUs: checking source cells can be distributed among as many threads as are available, and results can be aggregated by parallel reduction, a well-known parallel building block [3]. The CUDA implementation minimizes thread divergence and maximizes coalesced memory accesses. If the data is kept in GPU memory, tremendous speedups over CPU solutions can be achieved, especially for high-level aggregations. More details can be found in [6].

The main shortcoming of this approach is that this parallelization only works with a limited amount of target cells. This is because of memory limitations, regarding both the number of registers available per thread and of the on-device shared memory required for efficient thread communication and cooperation. If a large number of aggregated cells are to be computed, they will have to be split into smaller bulks which must be computed by separate kernel calls. Hence, roughly speaking, the approach can be described as a sequence of parallel computations, and its runtime depends heavily on the number of targets to be computed (apart from the size of the input, i.e. the number of source values involved).

There are two more disadvantages when dealing with large and sparse areas of targets. First, a lot of memory has to be allocated, much of which will be unused when many of the results are zero. Given the limited memory of GPUs, this soon creates a problem. Second, even just checking for a value that will be zero is computationally expensive if it happens many times. In such a case, GPU-based calculation can actually become a disadvantage rather than an advantage, as the invocation of GPU kernels typically takes longer than invocation of a pure host method.

3.2 Source-Driven Aggregation

An alternative approach to computing aggregates is to drive the calculation not by targets, but by the source cells. Instead of a parent-to-child map in dimensions, we use a child-to-parent mapping, which allows a thread to look up, for a given source cell, all the target paths of the query to which it contributes (see Fig. 3), together with the corresponding weights. The thread can then "construct" the target paths and add the value of this source to all appropriate target values. Hence, unlike the target-driven method, which could be labeled a "sequence of parallel aggregations" (in CUDA, it would consist of multiple calls of one simple kernel), the source-driven one is a "parallel execution of serial aggregations" (one complex CUDA kernel).

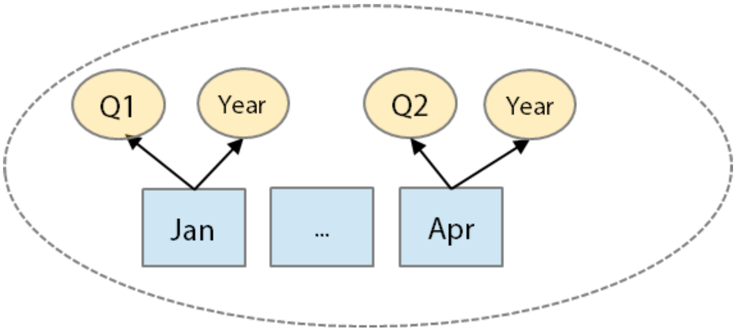


Fig. 3. Child-to-parent map required for source-driven aggregation

The most important advantage of such an approach is that targets with zero value are never created, i.e. no memory must be allocated for them and no computational resources are wasted for them. To achieve this, existing targets will be placed (and looked up) in a common hash table [1] located in GPU global memory. Using the terminology of [9], our approach would classify as "shared" for each GPU, but "independent" between multiple GPUs. Figure 4 gives an overview of the overall method on one GPU. The dotted line of cells labeled "target cell area" is not fully materialized – only existing target cells are created and placed in the hash table.

This approach results in a time complexity that is independent of the target area size, but depends on the actual number of non-empty targets. Moreover, since only one kernel call is require even for a large number of targets, each input will be inspected only once, which reduces actual running time.

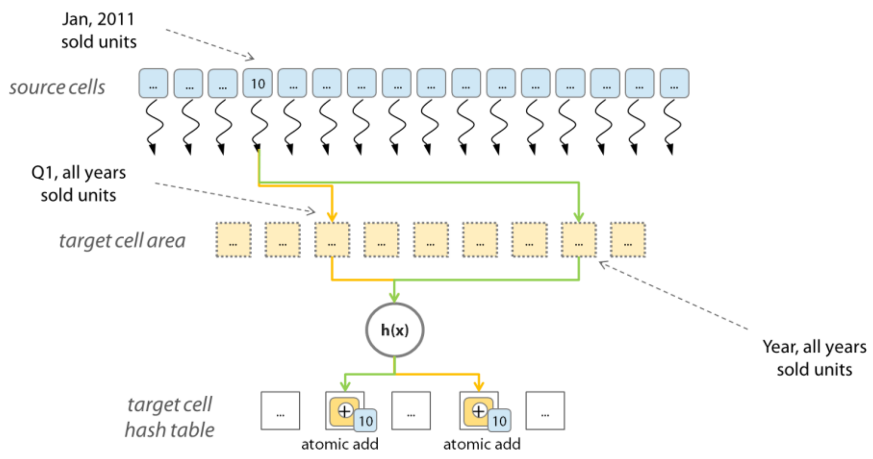


Fig. 4. Algorithm for source-driven aggregation. Source values are added to targets stored in a hash table in GPU RAM.

However, there are several challenges, regarding the SIMD-like parallelization that GPUs are built for: First, the number of targets for different source cells may vary dramatically; hence some threads will do significantly more work than others, and considerable thread divergence is not only likely but almost inevitable. Our solution benefits from the locality effect caused by a global sorting of source cells: adjacent source cells, which are handled by threads in the same *warp* (a group of 32 threads scheduled together on the same multiprocessor), are likely to have a similar number of targets to which they contribute.

Second, coalesced memory accesses (especially writes) are almost impossible to achieve, with target cells being scattered across a hash table. Even worse, many threads may compete to update (i.e. add their value to) the same target at a time, which makes it necessary to use atomic operations on the hash table, leading to congestion (see Fig. 5). On Nvidia GPUs with compute capability 1.3 or lower, the latter problem made the source-driven approach completely infeasible, leading to crashes and instabilities.

However, atomic operations have been improved significantly with both the Fermi and the Kepler architectures, and new features of the CUDA programming model can improve this even further. Since version 4.0, CUDA supports the *__ballot* operation, which allows threads in the same warp to compare their values of the same thread-local variable. This allows the pre-aggregation of values from all those threads which try to write to the same target location at the same time, and then let only one thread do the writing to the hash table. This warp-internal pre-aggregation (see. Fig. 6) can drastically reduce congestion, but warps from different multiprocessors may still compete for writing to the same memory address.

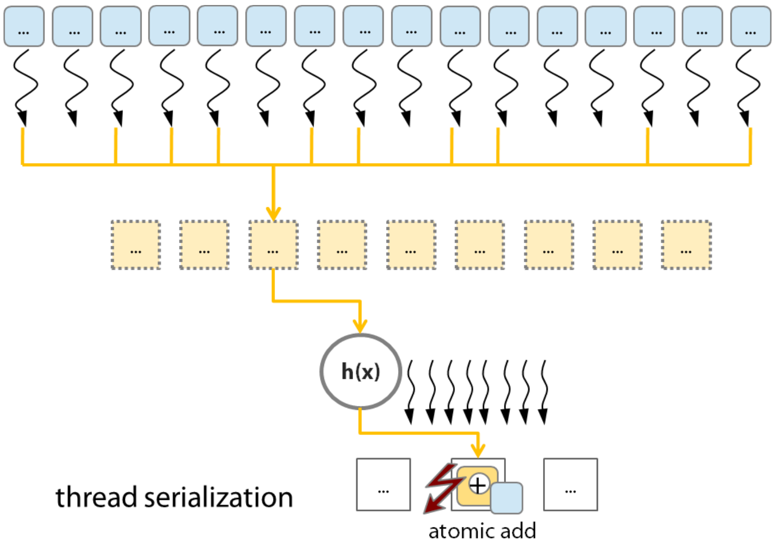


Fig. 5. Contention caused by thread serialization through atomic memory operations

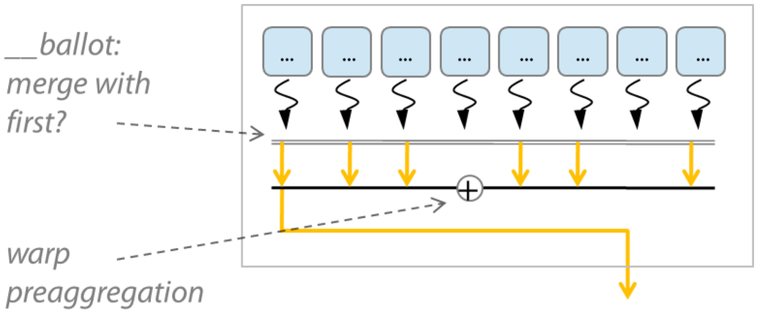


Fig. 6. Warp-wise pre-aggregation on Fermi or newer architectures

To further avoid contention, we can trade off some memory for better performance by allowing multiple hash functions. This is similar (though not identical) to the concept of “cloning”, as described in [9]. If different warps use different hash functions, they will write their results to different locations in the hash table. Of course this will increase the size of the hash table *k*-fold, if *k* hash functions are used. Also, an extra step is required to summarize the (up to) k results for the same target before returning them. This can be combined with a sort and compact step that removes the unused slots of the hash table and sorts the results according to their target path. The best value for the number *k* of hash functions cannot be determined a priori, since the number of existing targets in unknown; our implementation chooses k heuristically, taking into account the sizes of source and target areas, the number of warps used in the kernel, and the maximum allowed size of the hash table.

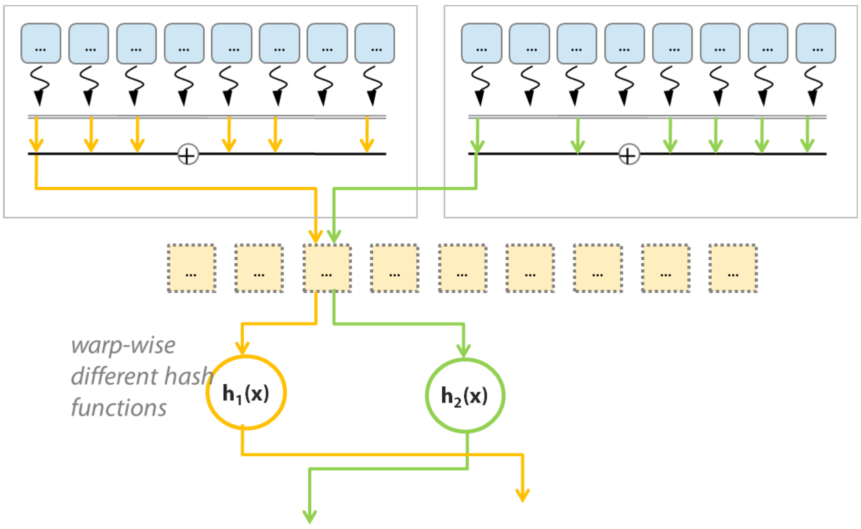


Fig. 7. Using different hash functions for different warps

3.3 Performance of GPU Aggregation Algorithms

While the target-driven approach is faster for small numbers of target cells, the source-driven method matches and outperforms it already at around 70 targets aggregated simultaneously. Fig. 8 shows a comparison of aggregation runtimes between the two approaches, on a cube containing approximately 128 million base cells. As can be seen, the time required by the source-driven algorithm is much less dependent on the target area size.

4 Parallel Filtering

As a result from step 1, we have a sparse aggregated area of cells, which can be imagined as a table, with each row corresponding to one element of the filter dimension and containing the slice with the respective values. Note, however, that the actual representation is simply a list of key/value pairs. Moreover, only cells with non-zero values are represented to account for sparsity. Since empty cells are still counted as 0 (zero) values for the filter condition, we have to distinguish 4 cases, depending on

(a) whether or not a 0 (zero) value satisfies the specified condition, and
(b) whether ANY or ALL cells of a slice are required to satisfy the condition.

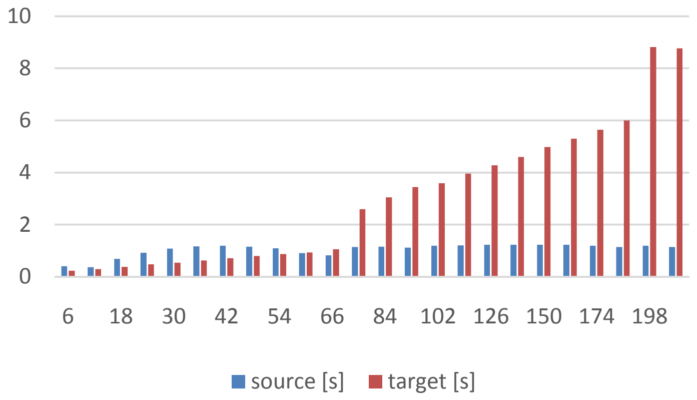
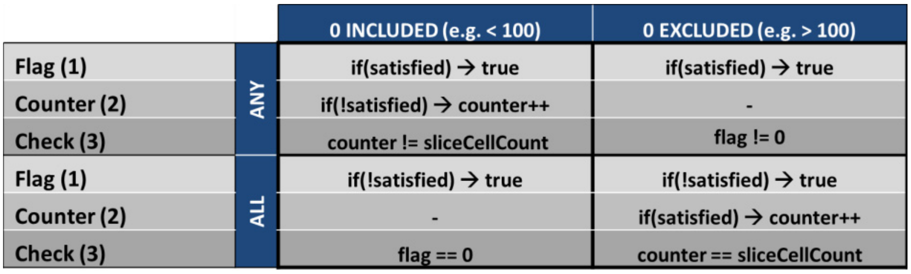


Fig. 8. Time (in seconds) for target-driven vs. source-driven aggregation by target area size

The four cases are depicted in Fig. 9. For example, consider the condition "ALL < 100", which means that all cells in the slice corresponding to an element of the filter dimension must have a value less than 100 for that element to be included in the output. In this case, any empty cells (i.e. cells with value 0) fulfill the condition. On the other hand, if the condition is "ALL > 100", the mere existence of an empty cell in the slice means that the element does not meet the condition.



		0 INCLUDED (e.g. < 100)	0 EXCLUDED (e.g. > 100)
Flag (1)	ANY	if(satisfied) → true	if(satisfied) → true
Counter (2)		if(!satisfied) → counter++	-
Check (3)		counter != sliceCellCount	flag != 0
Flag (1)	ALL	if(!satisfied) → true	if(!satisfied) → true
Counter (2)		-	if(satisfied) → counter++
Check (3)		flag == 0	counter == sliceCellCount

Fig. 9. Four cases of the Quantification Filter: ALL | ANY vs. zero included | not included.
(1) Flag: current cell value satisfies given condition (e.g. val > 1000)
(2) counter is incremented in special cases to keep track of all flag results
(3) Condition which has to be satisfied to add an element to result set

Our CUDA approach implements the computation of the second step filters by a parallel scan of all cells in all slices, using tens of thousands of concurrent threads. We use a variant of the parallel hashing scheme described in step 1 to hold the results. In addition, we employ a mechanism for minimizing unnecessary checks as much as possible, and a space-efficient method for checking multiple conditions (condition tree). The following paragraphs describe the procedure in more detail.

First, we need to initialize a GPU hash table with enough slots capable to hold all of the elements in the filter dimension (in the extreme case, all elements meet the filter criteria). Each slot in the hash table has space for an element ID, a value, a (binary) flag, a counter, and a lock. The lock is needed because all threads scanning cells of the same slice will try to write to the same hash table position; hence a synchronization mechanism is required. In order to save memory, our implementation uses the same variable for the flag and the lock.

The scan of the cells is implemented as a CUDA kernel, where each thread examines one cell at a time as follows:

(a) Compute the hash table position for the corresponding element, check existing flag and – if necessary – acquire the lock for that position.
(b) In the ANY (ALL) case, set the flag to TRUE (FALSE) if the value satisfies the condition and to FALSE (TRUE) if it does not (see Fig. 9).
(c) In the ANY (ALL) case, increment the counter if the condition is not satisfied (is satisfied) AND a zero value would satisfy (would not satisfy) the condition (see Fig. 9).
(d) Release the lock of the hash table position.

After the kernel has finished, the flags (1) and counters (2) contain all the information required for deciding whether or not the corresponding element has to be included in the result. The bottom line (3) in each quadrant of Fig. 9 lists the condition to be verified for the decision.

However, the result might still be incomplete. This is because there might be elements for which no hash entry was created because no cells exist in the corresponding slice. If zero values also satisfy the filter condition (left quadrants of Fig. 9), these elements must be added to the result.

Fig. 10 summarizes the overall procedure. Starting with the preprocessed cells from step 1, the kernel checks each cell and fills the hash table. If necessary, elements for zero-values are added, yielding the list of result cells after some postprocessing.

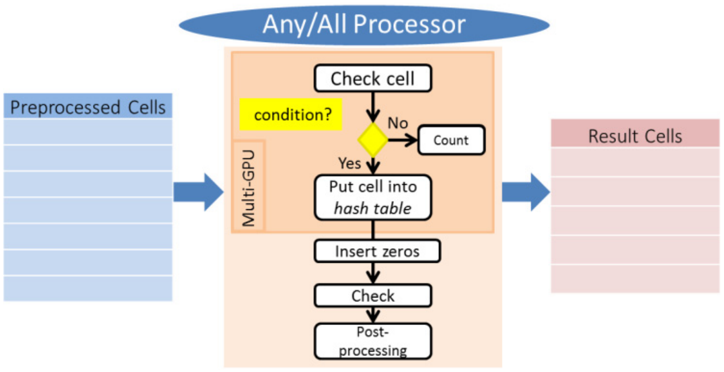


Fig. 10. Workflow of quantification process for ALL | ANY. Preprocessed cells are the result from step 1 (aggregation).

In order to minimize unnecessary checks, some optimization can be done. In particular, if the result is already clear, the check of a cell can be terminated without accessing the hash table, thus avoiding expensive locks and memory accesses. By checking the flag/lock at the hash table position of the processed cell we can discard those cells contributing to an element of the filter dimension which is already marked as "satisfied | not satisfied". The flow diagram of the optimized algorithm is depicted in Fig. 11.

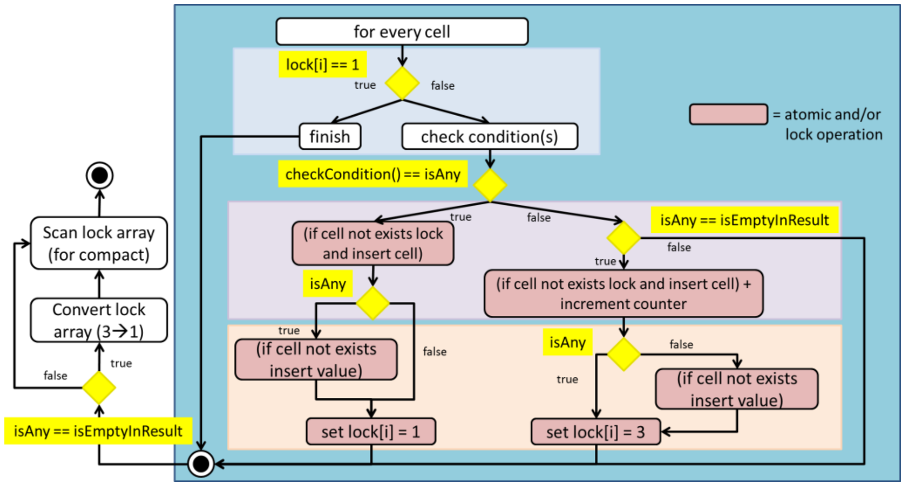

for every cell
lock[i] == 1
true
false
finish
check condition(s)
= atomic and/or lock operation
checkCondition() == isAny
true
false
(if cell not exists lock and insert cell)
isAny == isEmptyInResult
false
true
(if cell not exists lock and insert cell) + increment counter
isAny
true
false
(if cell not exists insert value)
isAny
false
true
(if cell not exists insert value)
set lock[i] = 1
set lock[i] = 3
Scan lock array (for compact)
Convert lock array (3→1)
true
false
isAny == isEmptyInResult

Fig. 11. Flow graph of optimized CUDA kernel minimizing locks and global memory accesses

The variable used for locking the access to a hash table entry is used at the same time as a flag to mark targets that have already been decided. If a thread discovers the flag is set for a target, it can immediately finish and continue with the next target. For targets that could still change, a different value for the lock is used. Actual locking is required only for the operations shown in pink boxes.

5 Performance Comparison

Our GPU algorithm can greatly speed up the computation of quantification filters. We compared the times of the standard (sequential) CPU algorithm (run on an Intel Xeon E5-2643) with our CUDA implementation on the same system with 2x Nvidia Tesla K40c GPUs, using a data model from Wikipedia.

The example data consists of the Wikipedia page access statistics of the year 2013. The main cube has over 1.8 million page names stored for every hour on every day in 2013. It also consists of different languages, projects and measures and sums up in over 300 million cells, corresponding to about 4.8 GB of compressed data in the GPU cube. We had three test cases for our test scenario:

(1) Popular Wikipedia pages, where we filtered out the top ranked pages depending on the page access count.

(2) Peak search, where we added a virtual measure "peak factor" which consists of the page access count of a given month divided by the page access count of a constant month (e.g. current month / August). The element is then a value less than 1 if the current month has a smaller page access count then the constant month, and a value greater than 1 if it has a higher page access count. By filtering on that measure, the top pages for every month can be obtained.

(3) "What's new" combines the peak search with an additional filter such that the result will be those pages that have a low peak factor since the selected month, and a high peak factor from then on until the end of the year. So we are searching not for a peak but for a constant high after a constant low.

All examples have to calculate the input of the quantification filter first. In this step the source size as well as the target size can be rather big (the page dimension consists of over 1.8 million elements) and could not be handled with the target driven approach (cf. section 3.1). So for all these quantification filters the source driven approach of our algorithm is used.

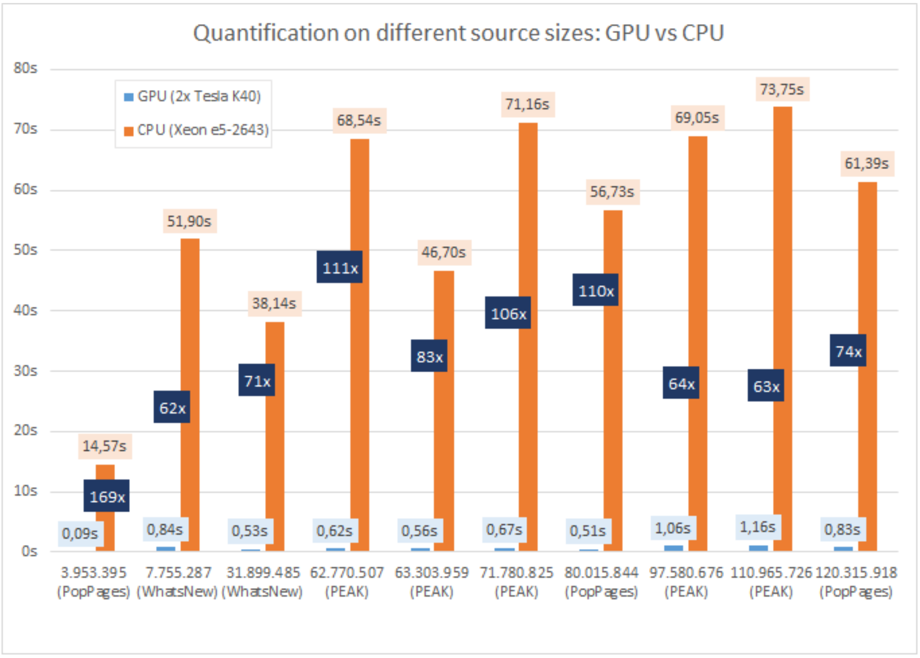


Fig. 12. Performance of quantification filters on GPU | CPU on Wikipedia example with different source sizes

In Fig. 12 the performance comparison of our GPU approach versus the CPU version is shown. The results are ordered by their source size from 3.9 million to 120 million (~6.25 million cells = 100 MB). As can be seen, the performance does not only depend on the input size but also on the different test scenarios. The blue boxes show the speedup factor of GPU versus CPU. With our new GPU approach it is now possible to obtain the results of data filters on big data sources in seconds instead of minutes and we gained speedups of about 60-100 times compared to the sequential CPU times.

Because the current maximum amount of GPU memory is 12 GB on a single Tesla K40, our GPU approach is limited to the amount of K40 card memory in the used system. With our test system (2x K40) we are able to store about 1 billion cells (16 GB) and then do complex filtering on the data which also requires temporary space on the GPU. High end systems can easily extend the amount of Tesla K40 cards to 4 or even 8 so that about 64 GB or 4 billion cells can be handled with our approach.

6 Conclusion

In this paper, we have presented a method for computing large and sparse areas of aggregated values for OLAP and other analytical database queries on GPU. In the first step we highlighted the differences of our source-driven approach to an earlier target-driven algorithm and described improvements to reduce contention related to atomic memory operations. In the second step we described our highly optimized approach of massive parallel quantification filters on big data sources and the excellent speedup compared to the CPU algorithms.

While first tests showed the usefulness of our approach in practice, the next step is to provide mechanisms and criteria which allow the decision on the best available algorithm (CPU, GPU target-driven, GPU source-driven) for a given query both effectively and efficiently, such as the method proposed in [2].

References

1. Alcantara, D.: Efficient Hash Tables on the GPU. PhD dissertation, University of California Davis (2011)
2. Breß, S., Beier, F., Rauhe, H., Sattler, K.-U., Schallehn, E., Saake, G.: Efficient Co-Processor Utilization in Database Query Processing. Information Systems 38(8), 1084–1096 (2013)
3. Wilt, N.: The CUDA Handbook (ch. 12: ”Reduction”). Addison-Wesley (2013)
4. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, M.D.: Fast computation of database operations using graphics processors. In: Proceedings of SIGMOD, Paris, France, pp. 206–217. ACM, New York (2004)
5. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. In: Transactions on Database Systems, vol. 34(4). ACM, New York (2009)

6. Lauer, T., Datta, A., Khadikov, Z., Anselm, C.: Exploring Graphics Processing Units as Parallel Coprocessors for Online Aggregation. In: Proceedings of DOLAP 2010, Toronto, Canada (October 2010)
7. Mostak, T.: An Overview of Map-D (Massively Parallel Database) Online whitepaper (2013), http://www.map-d.com/docs/mapd-whitepaper.pdf
8. Wu, H., Diamos, G., Sheard, T., Aref, M., Baxter, S., Garland, M., Yalamanchili, S.: Red Fox: An Execution Environment for Relational Query Processing on GPUs. In: International Symposium on Code Generation and Optimization (CGO) (February 2014)
9. Ye, Y., Ross, K.A., Vesdapunt, N.: Scalable Aggregation on Multicore Processors. In: Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), Athens, Greece. ACM (2011)
10. Ghodsnia, P.: An In-GPU-Memory Column-Oriented Database for Processing Analytical Workloads. In: VLDB 12 PhD Workshop, Istanbul, Turkey. ACM (August 2012)
11. Cognos TM1, http://www-03.ibm.com/software/products/en/cognostm1
12. Infor B, http://www.infor.com/content/brochures/infor10ionbicomprehensivebi.pdf
13. Jedox OLAP, http://www.jedox.com/en/products/jedox-premium/jedox-olap.html