An Investigation of Atomic Synchronization for Sort-Based Group-By Aggregation on GPUs

Bala Gurumurthy
David Broneske
University of Magdeburg
firstname.lastname@ovgu.de

Martin Schäler
University of Salzburg
martin.schaeler@sbg.ac.at

Thilo Pionteck
Gunter Saake
University of Magdeburg
firstname.lastname@ovgu.de

***Abstract*—Using heterogeneous processing devices, like GPUs, to accelerate relational database operations is a well-known strategy. In this context, the `group by` operation is highly interesting for two reasons. Firstly, it incurs large processing costs. Secondly, its results (i.e., aggregates) are usually small reducing data movement costs whose compensation is a major challenge for heterogeneous computing. Generally for `group by` computation on GPUs, one relies either on sorting or hashing. Today, empirical results suggest that hash-based approaches are superior. However by concept, hashing induces an unpredictable memory access pattern being in conflict with the architecture of GPUs. This motivates studying why current sort-based approaches are generally inferior. Our results indicate that current sorting solutions cannot exploit the full parallel power of modern GPUs. Experimentally, we show that the issue arises from the need to synchronize parallel threads that access the shared memory location containing the aggregates via `atomics`. Our quantification of the optimal performance motivates us to investigate how to minimize the overhead of atomics. This results in different variants using atomics, where the best variants almost mitigate the atomics overhead entirely. The results of a large-scale evaluation reveal that our approach achieves a 3x speed-up over existing sort-based approaches and up to 2x speed-up over hash-based approaches.**

I. Introduction

As data set sizes remain to grow exponentially [1], computing common database operations, such as join, aggregation or selection, becomes highly time-consuming. One well established strategy to keep pace with the vast amount of data is utilizing heterogeneous massively-parallel processing devices, such as GPUs [2]–[4].

In this paper, we address the problem of parallelizing a `group-by` operation followed by a subsequent `aggregate`. A corresponding example query is shown in Example I.1. The rational for studying this problem is twofold. Firstly, compared to other database operations, like joins, group-by operations are less affected by the data movement problem. The data movement problem occurs whenever data is shipped to or retrieved from a heterogeneous processing device. This may incur a major cost factor [5]–[7]. Secondly, computing the grouping and aggregate is highly compute intensive [8]–[10].

Example I.1 (SQL query with grouped aggregate)**.**

```
SELECT count(*), l_returnflag FROM lineitem
GROUP BY l_returnflag ORDER BY l_returnflag;
```

This work was partially funded by the DFG (SA 465/51-1 & SA 465/50-1.)

However, massively parallelizing a grouping and subsequent aggregate is challenging – independent of the processing device. The reason is that the data of one group is arbitrarily distributed over the data set and, thus, one requires for some kind of synchronisation between the threads. Relying on a GPU increases the difficulties, as a GPU's architecture is not designed for efficient inter-thread communication, which is e.g., done by atomic operations.

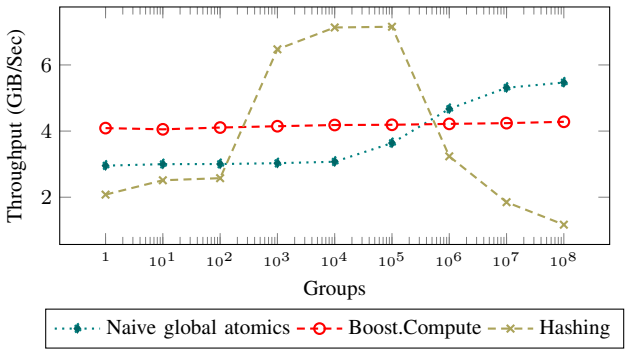


Fig. 1: Throughput of different group-by approaches on a RTX 2080 Ti GPU on 2^{27} uniformly random input values

Generally, for grouped aggregation on GPUs one relies either on sorting or hashing [11] with empirical results suggesting that hash-based approaches are superior [10], [12]. In Fig. 1, we depict the throughput of a recent hash-based grouped aggregation and a sorting based grouped aggregation (i.e., Boost.Compute). We observe selecting the best algorithm depends on the number of groups. For reasonable group numbers between 10^2 and 10^6, hashing is best. For smaller numbers, Boost.Compute has the highest throughput. Adding a third approach, a naive sort-based aggregation using atomic operations (i.e., hardware-blocks), we observe that its throughput increases monotonically until each value is assigned uniquely to a group. From 10^6 distinct groups it offers even the best performance.

Despite this remarkable result, our hypothesis is still that in current sort-based solutions, all threads aggregate data simultaneously and block each other in the case of small group sizes. Hence, one does not exploit the massive parallel power that modern GPUs offer. To this end, we first investigate whether the synchronisation overhead is the decisive

bottleneck. Then, we aim at proposing a solution that mitigates the synchronisation overhead aiming at a throughput that is at least equal to – or even superior – to a hash-based solution or Boost.Compute depending on the number of groups. Our investigations result in the following contributions:

1) Our examination reveals that the synchronisation step for merging partial group results is an important bottleneck for sort-based aggregation.
2) We propose sort-based aggregation approaches mitigating the synchronisation overhead by reducing the amount of issued atomics. For instance, one approach requires 2 atomics per GPU thread independent of the data distribution. Afterward, we examine how the number of concurrent threads and chunk sizes affect the throughput of our approaches.
3) Our results suggest that atomics-based approaches are in general 3x faster than Boost.Compute and up to 2x faster than hash-based approaches for reasonable number of groups, e.g., found in the TPC-H benchmark.

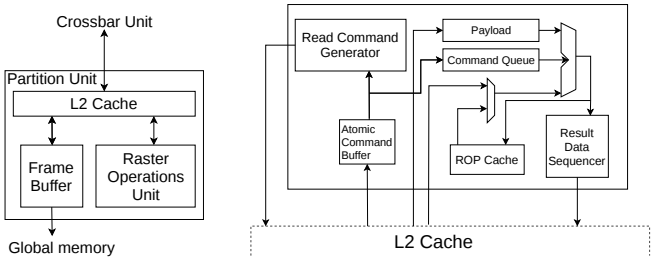
II. Atomics in GPU

In this section, we examine our hypothesis that sort-based group-by approaches suffer from the issues that all threads request synchronisation simultaneously leading to lock congestion. To this end, we first investigate the execution of atomics in GPUs. Then, we conduct an experiment to examine the validity of our hypothesis.

A. Architectural Components Involved in Atomic Execution

GPUs contain multiple `Memory Partition Units` (MPU) to handle upcoming data access requests (see Fig. 2(a)). These MPUs favor coalesced memory accesses to hide memory latency for parallel threads to improve efficiency. Furthermore, it is the main component, where atomic operations are handled.

Whenever a thread encounters an atomic instruction, it sends an *atomic command* to the MPU. The command contains the target operation (add, sub or exchange) and a payload value. This command is stored in a *command buffer* until the targeted shared data is fetched. Once fetched, the command buffer forwards the data and the atomic command to the raster operation unit (ROP) for execution (see Fig. 2(b)).



(a) MPU components (b) Atomics in ROP based on [13]

Fig. 2: Components involved in global memory atomics [14]

The forwarded atomic command is stored in an *atomic command buffer* - a FIFO queue to ensure serialized atomics. Using this queue, the ROP updates the shared result atomically. Finally depending on the type of atomics, the result is either returned to the target SM (in case of increments, decrements or addition commands) or simply stored in the global memory (min, max or exchange commands).

B. Profiling Atomics

Next, we study the negative impact of atomics on group-by aggregations determining an upper bound or worst case. This shall indicate the general potential that we can expect when mitigating the synchronisation overhead.

1) Upper Bound of Atomics Throughput: Normally, increasing the concurrency in a GPU improves the throughput. However, increasing concurrency with atomics creates a backlog of threads waiting to access a memory location decreasing the throughput. Naturally, the severity of this backlog grows with increasing concurrency. Specifically, when there is only a single shared memory target, i.e., the input contains a single group or a `reduction operation`. The throughput of such an execution represents a worst case allowing to measure the maximum negative impact of atomics on throughput. Hence, we measure this impact based on the number of concurrent threads, as GPUs generally benefit from massive parallel computing power. However, in case atomics are the major bottleneck, we should observe that the throughput declines for high numbers of concurrent threads.

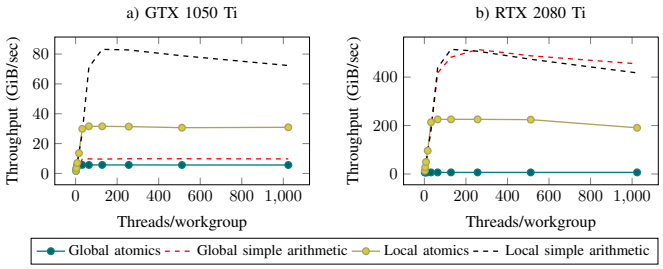


Fig. 3: Throughput for naive atomics and arithmetics

2) Simple Arithmetic Operation as Optimal Throughput: To be able to quantify the impact of atomic execution, we also execute a naive arithmetic operation on the same location with no synchronisation at all (using a simple arithmetic operation). This way, the arithmetics may cause dirty writes. However, as their overall flow remains the same as atomics, this is a good way of quantifying the impact of atomics. Note, in the experiment we use local and global atomics as well as local and global arithmetics.

In Fig. 3, we plot the results of our experiment using 2^{27} integers on different GPUs. The results suggest three insights:

1) Comparing Fig. 3 (a) & (b), the throughput for *local* memory atomics in newer generations is improved significantly (instead of being 60% slower on GTX 1050 Ti, local atomics are only half as slow as local arithmetics on RTX 2080 Ti). Hence, atomics get promising on modern GPUs.

2) The throughput difference for arithmetics and atomics is large with local atomics having a penalty of 2.0x to 2.6x on either GPU and global atomics with up to 1.75x on GTX

1050 Ti and up to 77x on RTX 2080 Ti compared to their simple arithmetic counterparts. Hence, we need to mitigate this penalty to unleash the full parallel power of present-day GPUs.

3) When using atomics, the best performance is reached for a small number of concurrent threads. Increasing thread count, may even reduce performance. This is the expected undesired behavior further indicating that one cannot exploit the massive parallel power GPUs offer.

These results may – at first sight – suggest to rely on local atomics rather than on global ones. Indeed local atomics are faster due to less threads accessing the same memory. That is, an atomic operation over this memory serializes only the associated threads in the work group. However, relying on local atomics would require an additional synchronisation step when combining the partial results in the local shared memory to the final result. Furthermore, the small size of local memory limits its use for group-by aggregation.

III. Atomics for Sort-Based Aggregation

As we can infer from the previous section, multiple components are involved in atomic execution, which incurs a considerable overhead. Therefore, minimizing the number of atomics issued should significantly improve the overall throughput. To this end, we first present the naive atomic aggregation and, afterward, introduce optimizations that we apply, which aim at reducing the amount of issued atomic operations.

A. Sort-Based Aggregation on a GPU

A traditional (sequential) sort-based aggregation sorts the grouping attribute to identify the groups inside. This mechanism has two phases: The first phase sorts the input into clusters according to the group keys, which forms a sequence of groups. The second pass sequentially aggregates the groups present in the sorted input. In order to parallelize this processing for GPUs, additional phases are needed, as explained at the example of a COUNT aggregation below.

The sort-based aggregation on GPUs has three phases [11]: map, prefix-sum and aggregate (four, if we consider sorting). First, the *map* phase compares two consecutive sorted-input values and returns 0 in case they match; 1 otherwise. As shown in the example in Fig. 4, this phase marks the group boundaries of a given sorted input (with a 1). Next, the *exclusive prefix-sum* computes the target aggregate location for each group. As these two phases are well known on a GPU, we use standard operators for them. The final *aggregation* phase aggregates the input values according to the target positions from the prefix-sum. For this phase, our atomic based aggregation is used to compute aggregated group-by results.

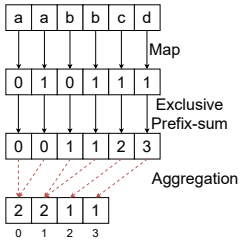


Fig. 4: Three-phase atomic COUNT aggregation

B. Minimizing Atomics Using Private Space

The naive sort-based aggregation issues one atomic operation per input value, i.e., the amount is equivalent to the data set size. Considering the processing of atomics on the GPU, it is reasonable to reduce the contention of threads by a more complex operator design. To this end, we exploit the fact that group values inside a sorted array are sequential such that all values of a group appear after one another before the next group starts. Now, imagine the following hypothetical scenario, where we chunk the sorted data s.t. all values of a single group are assigned to a single thread. Hence, no synchronisation issues can occur, removing the need for atomic operations and exploiting the full parallelism of GPUs. Of course, determining such a perfect chunking creates large overhead and leads to load imbalances. Nevertheless, as we will see, our solutions get fairly close to this ideal scenario.

The distinction of when and how to synchronize the partial result of a thread allows proposing two algorithms: (1) using a private aggregate variable and (2) using a private aggregate array. Both versions are shown in Fig. 5, where two threads aggregate their own chunk of three values.

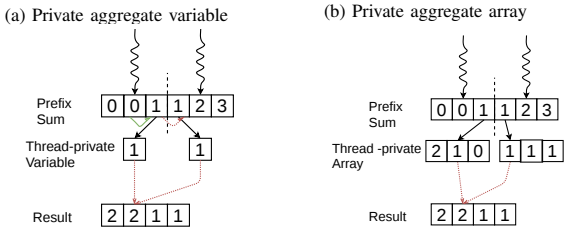


Fig. 5: Using private space in aggregation phase

The execution flow of both variants is roughly the same. In both, a thread sequentially reads its chunk of the prefix-sum and aggregates the corresponding input values within its private space until it encounters a group boundary. However, the variants differ in handling their partial aggregates and thus in the number of required atomics.

a) Single Private Variable Result Buffer: A thread using a private variable as a result buffer conducts an atomic operation whenever it encounters a group boundary, because it only buffers the aggregate of a single group. Therefore, this variant issues as many atomics as there are groups in its input chunk. As a result, the best number of required atomics is 1, in case there only is a single group per thread. The exact number of atomics and the time when they are issued depends on the data distribution. This is important, as this leads to the desired effect that, assuming group boundaries are evenly distributed, the number of concurrent atomics declines.

b) Private Array Result Buffer Variant: Instead of using a single variable as buffer, this variant uses a private array to buffer the aggregates of all groups it processes. In the private array variant, a thread sequentially traverses its input and aggregates into the current result buffer position until a group border is found. Then, the next position is used for the next group aggregate. Since the arrays in a GPU are initialized statically, the result buffer must have the same size as the input

data to cover the case that all input values belong to a distinct group. This limits the chunk size when the array is stored in local memory.

Once aggregated, the threads propagate their private result into the shared memory containing the overall result. To further mitigate the negative effects of excessive atomics usage, we conduct another optimisation reducing the number of required atomics per thread to exactly 2. This makes the number of required atomics independent of the data distribution depending only on the number of concurrent threads.

It works as follows: As the input data is sorted, synchronisation issues may only arise for the first and the last group processed by a thread. The first group may have already begun in the prior thread's data input. The final group may continue in the next thread's data input. All other groups are only processed within the current thread. Thus, the approach pushes these result to global memory without synchronisation having the *optimal* performance shown in Fig. 3 (global arithmetic).

IV. Experiments

In this section, we evaluate our approaches using micro benchmarks and a comparison to state-of-the-art competitors. For both parts, we use the same setup: Since the GPU hardware has direct influence on atomics, we profile our atomic-based aggregation on two GPU versions varying in their memory bandwidth - NVIDIA GTX 1050Ti and NVIDIA RTX 2080Ti. All our experiments are executed on a linux machine with GCC 6.5 and OpenCL 2.1. The input dataset contains 2^{27} (due to Boost.Compute's data size limitation) randomly generated integers representing our group-by keys. While for the micro benchmark and the first comparison, data is presorted (i.e., sorting time is disregarded), the unordered data is used for fairness for the final competitor comparison. Each measurement is repeated 100 times and we present the average throughput for all variants. For brevity, we present results for count aggregation, but the result also holds for different aggregate functions and also data sizes.

A. Micro Benchmark

The parameters affecting performance are (1) thread size per work group and (2) chunk size of input data per thread. To this end, we conduct experiments to examine their influence and find an optimal configuration used in the remainder.

1) Examining Optimal Thread Size for Naive Atomics: In this experiment, we identify the optimal thread size per work group for naive atomics serving as baseline. The resultant throughput of naive atomics on local or global memory with varying group and thread sizes is given in the heatmaps in Fig. 6. Notably, the implementation of the naive atomics variant on global memory is straightforward (i.e., the aggregation step in Fig. 4 uses an atomic operation on the global memory). However, the atomic variant on local memory needs an additional merging step. This step is to merge the partial aggregates inside the workgroups' local memory into the final result in the global memory. In this naive local variant, we perform the merging similar to the approach used for our private array variant, where only the first and last positions are merged atomically. Our result reveals the best performance for a large number of groups and many threads, because many threads efficiently hide memory latency and because a higher number of groups (i.e., a larger spread of target locations in memory) create less concurrency on atomic writes. The results also clearly show an improvement from using local memory as cache for partial aggregates. Still, the penalties of an extra merging step are significant and thereby reduces the overall throughput. As an overall result, the best thread sizes are 256 for GTX 1050 Ti and 1024 for RTX 2080 Ti, which we then use to compare naive atomics with our approaches and the competitors.

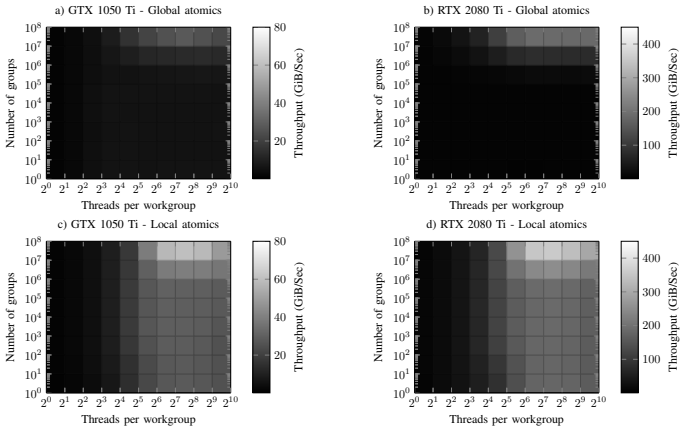


Fig. 6: Impact of varying group and thread sizes

2) Best Thread and Chunk Size for Atomic Variants: In addition to the thread sizes, our variants using a private array/variable (either in local or global memory) are also influenced by the number of input values per thread (chunk size). Hence, we average the variants' throughput over all tested number of groups and plot the results in Fig. 7 and 8[1].

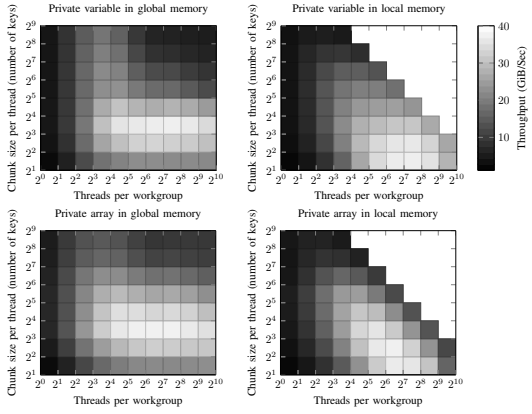


Fig. 7: Impact of chunk size and threads in GTX 1050 Ti

Considering the influence of the chunk size on the throughput, we observe that smaller to medium sized chunks ($2^2 - 2^7$) are beneficial compared to larger ones. In the latter case, the memory controller becomes the bottleneck, due to too many

[1]Note that not all combinations of chunks and threads are possible as they cross the physical limit of local memory that can be allocated.

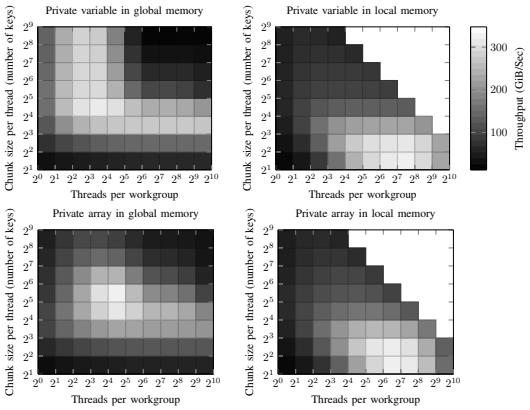


Fig. 8: Impact of chunk and threads in RTX 2080 Ti

requests from threads that fetch input data from global memory and from the execution of atomic operations. Since the MPU incurs coalesced accesses, fetching bigger chunks of data for multiple threads requires multiple cycles, which degrades performance. In contrast, the local memory variants prefer very small chunk sizes ($2^1 - 2^3$), whereas global memory benefits from slightly larger ones ($2^2 - 2^7$). Unlike the naive atomics where larger numbers are beneficial, chunking improves the performance of even smaller thread sizes. Interestingly, there is only a small difference between using a private variable and a private array for storing intermediate results. By contrast, the throughput behavior changes w.r.t. the devices, since there is a broad spectrum of well performing variants on GTX 1050 Ti, which shrinks for the RTX 2080 Ti. This indicates that the variants are sensitive to the underlying hardware and need a smart variant tuning procedure [15].

B. Comparative Experiments

1) Comparison of Atomic Variants: First, we identify the best variant of our approaches per device used for the final experiment. To this end, we compare the performance of the best performing chunk and thread size combination of the two private aggregate variants with the naive atomic variants with an optimal thread size. The results are shown in Fig. 9.

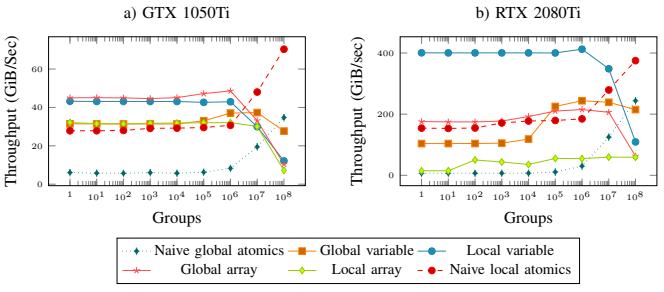


Fig. 9: Performance comparison of atomic variants

Our results show that global array and local variable have a higher throughput than the naive atomic variants for almost all number of groups (i.e., except a larger number of groups). This limitation of our variants is expected as a larger number of groups leads to multiple groups within a chunk. In this case, a thread has to repeatedly insert the final result into global memory degrading its performance. We also see only a small improvement using local memory for our variants on the GTX 1050 Ti, which in contrast is a huge improvement on the RTX 2080 Ti. This is consistent with Section II-B. Finally, for very high amounts of groups, the overhead of internal synchronization for the private aggregate variants does not pay off. Hence, naive local atomics perform best in this case.

In summary, our variants reach a speed up of 6x-12x compared to the naive global memory atomics and a speed up of 1.5 - 2.6x compared to the naive local memory atomics. For GTX 1050 Ti, the variant using a private array in global memory is the optimal variant with a speed-up of 6x the naive global memory atomics and 1.6x the naive local memory atomics. For RTX 2080 Ti, the variant using a local variable is clearly superior with a speed-up of about 12x the naive atomics and up to 2x the local memory atomics.

2) Comparison With State of the Art: As a final evaluation, we compare our performance with other state-of-the-art mechanisms. To this end, we include a sorting step before executing the best performing atomic variants and compare against a sort-based aggregation using Boost.Compute and the hash-based aggregation by Karnagel et al. [12].

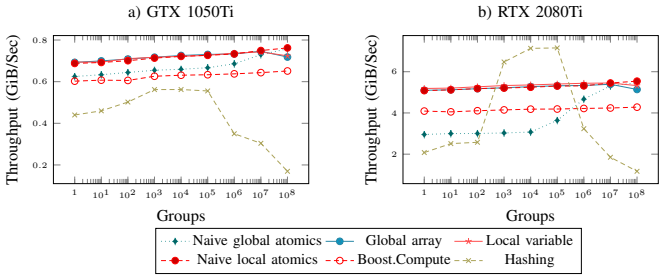


Fig. 10: Overall comparison against state-of-the-art competitors. The performance of atomic variants now includes sorting.

Our results in Fig. 10 reveal that our complex atomic variants mostly lead to the best performance. On the GTX 1050 Ti, we reach on average 20 % speed-up over naive global atomics and Boost.Compute, while it reaches nearly 2x the speed of hash-based aggregation. We see a similar speed-up on the RTX 2080 Ti except that our variant using a local variable reaches up to 1.25x the performance of Boost.Compute. Interestingly, hash-based aggregation is only superior for groups between 1,000 and 100,000. This is because a smaller number of groups leads to a synchronization overhead when accessing the shared global hash table concurrently and a larger number of groups increases the hash table beyond a manageable size.

Discussion: In summary, we can see that for the common use case of up to some hundred groups[2], a sort-based aggregation using atomics is the superior variant to be used. This is remarkable, as usually hashing is the best variant [10], [12]. We argue for a change of this general assumption for the following three reasons:

[2]For instance in the TPC-H, 11 out of 16 queries do a group-by on less than 500 groups. Seven of them operate on less than 10 groups.

- There are a lot of circumstances where presorted data is grouped (due to sort-merge join or a clustered index) or data has to be sorted after executing the grouping (due to an order-by statement). In these cases, it would be the natural option to also employ a sort-based grouping.
- Although the sorting time dominates the throughput of our variant in Fig. 10 (making up 80% of the execution time), it is still the most stable strategy on the GPU across the group sizes. The reason is a more cache-friendly access pattern and a better fit for the SIMT processing model of the GPU [16].
- Due to increased local memory performance of modern GPUs, the overhead of atomic operations can be effectively mitigated.

As a result, optimizing sort-based group-by operators is a reasonable future work not only for GPUs, but also CPUs.

V. Related Work

Since the usage of GPUs as general-purpose accelerators, many researchers use GPUs to accelerate DBMS operations. In the following, we list work that closely relates to us.

Modeling performance of atomics: Hauck et al. propose to buffer atomic updates to reduce contention in a reduction [17]. Hoseini et al. explore the impact for atomics on CPUs [18].

Sort-based aggregation on GPUs: Sort-based aggregation on a GPU was first devised by He et al. [11]. A similar method is followed by Bakkum et al. [2] using CUDA in SQLite. However, our result shows that their additional passes over the data cause more data access cost than using atomics.

Hash-based aggregation on GPUs: Alternatively to sort-based aggregation, hashing can be used for computing aggregates. Hence, there are several related approaches that tune hash-based aggregation for GPUs [10], [19], [20].

Non-grouped aggregation on GPUs: Simple aggregation has the same execution pattern as grouped aggregation, where a single output location is accessed by all threads. To mitigate contention, there are various approaches [12], [21].

VI. Conclusion

GPUs with their massively parallel processing have been used for more than a decade now to accelerate compute-intensive database operators. One such compute-intensive database operator is a grouped aggregation. Although, up to now, hashing is the predominant technique for grouped aggregations even on the GPU, a sort-based grouped aggregation is an important alternative to be considered – especially with an improved performance of atomics.

In this paper, we investigate how far we can tune a sort-based grouped aggregation using atomics in the aggregation step. To this end, we design two alternative variants using a private variable or array and investigate their performance improvement when using local or global memory followed by an atomic-based propagation of private aggregates.

Our results show that our variants speed up grouped aggregation compared to a naive usage of atomics by a factor of 1.5 to 2, when well configured. Furthermore, a sort-based grouped aggregation using atomics can outperform a hash-based aggregation by 1.2x to 2x for most used group sizes.

For future work, we envision the following directions. Since there are numerous configuration combinations, an automated exploration of the right configuration is to be done – especially to choose the right variant for a new GPU architecture. To this end, we propose to follow or extend the approach given by Breß et al. [22]. Furthermore, investigating the usage of atomics for hash-based and strided sort-based group-by (cf. Karnagel et al. [12]) is an important step forward.

References

[1] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, no. 2, pp. 171–209, 2014.

[2] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," *GPGPU*, pp. 94–103, 2010.

[3] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SIGMOD*, 2004, pp. 215—-226.

[4] H. Wu, "Acceleration and execution of relational queries using general purpose graphics processing unit," in *GPGPU*, 2015.

[5] I. Arefyeva, D. Broneske, G. Campero, M. Pinnecke, and G. Saake, "Memory management strategies in CPU/GPU database systems: A survey," in *BDAS*. Springer, September 2018, pp. 128–142.

[6] A. Becher, L. B.G. *et al.*, "Integration of FPGAs in database management systems: Challenges and opportunities," *Datenbank-Spektrum*, 2018.

[7] S. Breß, H. Funke, and J. Teubner, "Robust query processing in co-processor-accelerated databases," in *SIGMOD*, 2016, pp. 1891–1906.

[8] P. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark," in *TPCTC*, 2013, pp. 61–76.

[9] B. Gurumurthy, D. Broneske, M. Pinnecke, G. C. Durand, and G. Saake, "SIMD vectorized hashing for grouped aggregation," in *ADBIS*, 2018, pp. 113 – 126.

[10] T. Behrens, V. Rosenfeld, J. Traub, S. Breß, and V. Markl, "Efficient SIMD Vectorization for Hashing in OpenCL," *EDBT*, pp. 489–492, 2018.

[11] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM TODS*, pp. 1–39, 2009.

[12] T. Karnagel, R. Müller, and G. M. Lohman, "Optimizing GPU-accelerated group-by and aggregation." *ADMS*, pp. 1–12, 2015.

[13] D. B. Glasco, P. B. Holmqvist, G. R. Lynch, P. R. Marchand, K. Mehra, and J. Roberts, "Cache-based control of atomic operations in conjunction with an external alu block," Mar. 13 2012, uS Patent 8,135,926.

[14] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, "General-purpose graphics processor architectures," *SLCA*, pp. 67–76, 2018.

[15] V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl, "The operator variant selection problem on heterogeneous hardware." in *ADMS*, 2015.

[16] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *Proc. VLDB Endowment*, p. 1378–1389, 2009.

[17] M. Hauck, M. Paradies, and H. Fröning, "Software-based buffering of associative operations on random memory addresses," in *IPDPS*, 2019, pp. 943–952.

[18] F. Hoseini, A. Atalar, and P. Tsigas, "Modeling the performance of atomic primitives on modern architectures," in *ICPP*, 2019, pp. 1–11.

[19] D. G. Tome, T. Gubner, M. Raasveldt, E. Rozenberg, and P. A. Boncz, "Optimizing Group-By and Aggregation using GPU-CPU Co-Processing," *ADMS*, pp. 1–10, 2018.

[20] Y. Yuan, R. Lee, and X. Zhang, "The yin and yang of processing data warehousing queries on gpu devices," *Proc. of the VLDB Endowment*, vol. 6, no. 10, pp. 817–828, 2013.

[21] T. Lauer, A. Datta, Z. Khadikov, and C. Anselm, "Exploring graphics processing units as parallel coprocessors for online aggregation," in *DOLAP*, 2010, pp. 77–84.

[22] S. Breß and G. Saake, "Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS," *Proc. VLDB Endowment*, pp. 1398–1403, 2013.