Distributed and Parallel Databases (2023) 41:387–409
https://doi.org/10.1007/s10619-023-07424-2

Novel insights on atomic synchronization for sort-based group-by on GPUs

Bala Gurumurthy[1] · **David Broneske**[2] · **Martin Schäler**[3] · **Thilo Pionteck**[1] · **Gunter Saake**[1]

Accepted: 10 April 2023 / Published online: 24 April 2023

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 conflicting 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.

Keywords GPU-based DBMS · GPU-accelerated DBMS · GPU-libraries · Performance comparison

✉ Bala Gurumurthy
bala.gurumurthy@ovgu.de

[1] University of Magdeburg, 39104 Magdeburg, Germany

[2] German Center for Higher Education Research and Science Studies, 30159 Hannover, Germany

[3] University of Salzburg, 5020 Salzburg, Austria

1 Introduction

As data set sizes are bound 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 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], and thus a perfect use case for parallelization.

Example 1 (SQL query with grouped aggregate)

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

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 some kind of synchronization 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.

Generally, for grouped aggregation on GPUs, one relies either on sorting or hashing [11] with empirical results suggesting that hash-based approaches are generally superior [10, 12]. In Fig. 1a, we depict the throughput of a recent hash-based

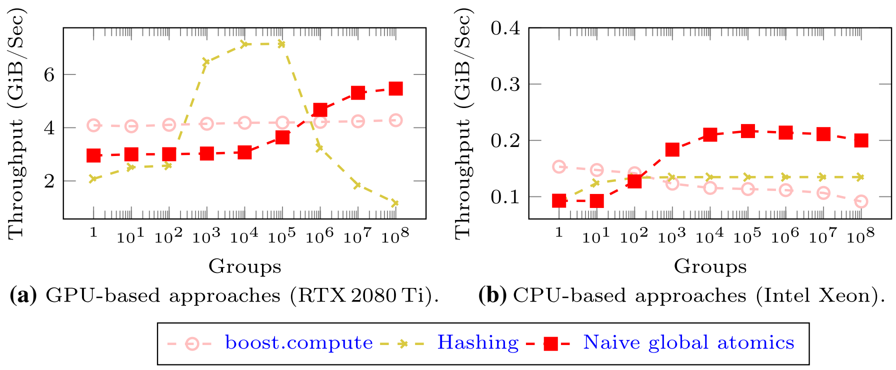

Fig. 1 Throughput of different group-by approaches on a RTX 2080 Ti GPU and Intel Xeon CPU on 2^{27} uniformly random input values. Note, the different scale of the y-axis

grouped aggregation and a sorting based grouped aggregation (i.e., boost.compute). We observe that 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-based), 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. By contrast in Fig. 1b, we depict the throughput when applying the same techniques to the CPU. We observe firstly that the performance pattern is entirely different, with the atomic-based approach being superior for a wide range of group numbers. Secondly, the CPU versions are more than a magnitude slower, i.e., there is a substantial throughput benefit one can invest to move data to the GPU, in case it not already resides there.

Despite this remarkable result, our hypothesis is still that in current sort-based solutions, all threads aggregate data simultaneously and block each other. This is supposed to hold especially 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 synchronization overhead is the decisive bottleneck. Then, we aim at proposing a solution that mitigates the synchronization 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 synchronization step for merging partial group results is an important bottleneck for sort-based aggregation.
2. We propose sort-based aggregation approaches that mitigate the synchronization 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 a reasonable number of groups, e.g., found in the TPC-H benchmark.

This is an extended version of [13] and, in addition to the original contributions, this paper also features:

1. An investigation how latest advances in GPU's architecture change the significance of our contributions w.r.t. state-of-the-art hash-based approaches. The key result is that the superiority of our atomic-based solution improves due to the larger number of available HW-based atomics processing component in the latest GPU generation.
2. An examination how different data distributions affect the performance of our contributions. The results suggest that the distribution has only a marginal effect, and thus our conclusions hold independent of the data distribution.

3. We put the GPU results into context of results one can expect on present-day CPUs. The key insight is that our GPU-based solutions are on average by one order of magnitude faster.

The remainder of the paper is structured as follows. In Sect. 2, we present preliminaries on the execution of atomics on GPUs and their performance. Afterward, we introduce several alternative approaches for using atomics for a sort-based group-by (Sect. 3). In Sect. 4, we detail our extensive evaluation using microbenchmarks and a comparison of the full-fledged group-by operator with state of the art approaches. Finally, we review related work in Sect. 5 and conclude in Sect. 6.

2 Atomics in GPU

In this section, we examine our hypothesis that sort-based group-by approaches suffer from the issue that all threads request synchronization 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.

GPUs favor an improved throughput instead of latency [14]. A GPU architecture contains multiple *graphical processing clusters* (GPCs), *memory partition units* (MPUs) and an off-chip DRAM also known as the *global memory*. The cores access global memory and execute atomics over them using the MPUs.

To ease memory bandwidth, there is also limited shared memory or *local memory* available that is only accessible within threads of a working group, which is significantly faster than using the global memory. Note: we follow the OpenCL naming conventions for the GPU components and implementation throughout this work. For example, the execution of any one of our variants would take the input from global memory (DRAM) and place them into registers. The threads within a work-group compute partial aggregates and place them in the local memory and finally synchronize them back into the global memory.

In this section, we provide an overview of these components involved in atomic execution. Note, since the architecture of a GPU is a black box, we explicitely refer to the work of Aamodt et al. and Glasco et al. [15, 16] for our work. We highly recommend these articles for more insights.

2.1 Architectural components involved in atomic execution

GPUs contain multiple `Memory Partition Units` (MPU) to handle upcoming data access requests (see Fig. 2a). 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

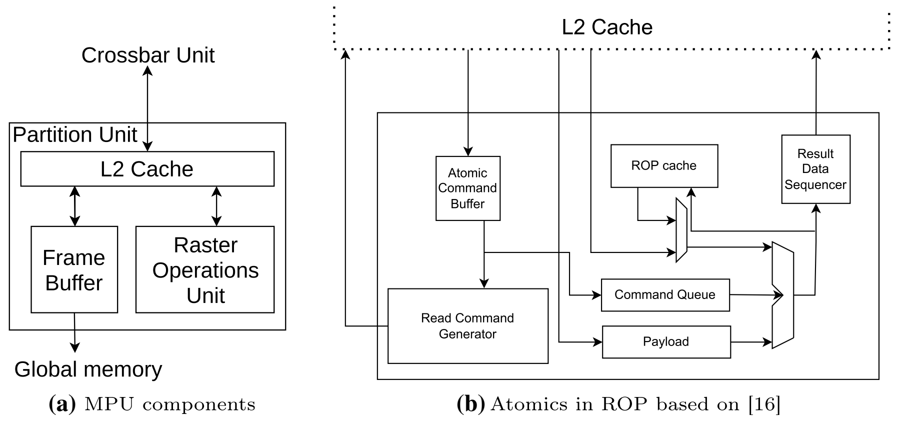

Fig. 2 Components involved in global memory atomics [15]

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. 2b).

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).

2.2 Profiling atomics

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

2.2.1 Upper bound of atomics throughput

Normally, increasing the concurrency in a GPU improves the throughput. In contrast, increasing concurrency with atomics creates a backlog of threads waiting to access a memory location, adversely affecting throughput. Naturally, the severity of this backlog increases with increasing concurrency. Specifically, the severity is high when only one shared memory target is accessed, such as when the input contains a single group or `reduction operation`. The throughput of such an execution represents the worst case, allowing us to measure the maximum negative impact of atomics on a GPU's throughput. Here, we run reduce operation with increasing concurrent threads. In the case of atomics, we observe a major bottleneck due to which throughput declines for high numbers of concurrent threads.

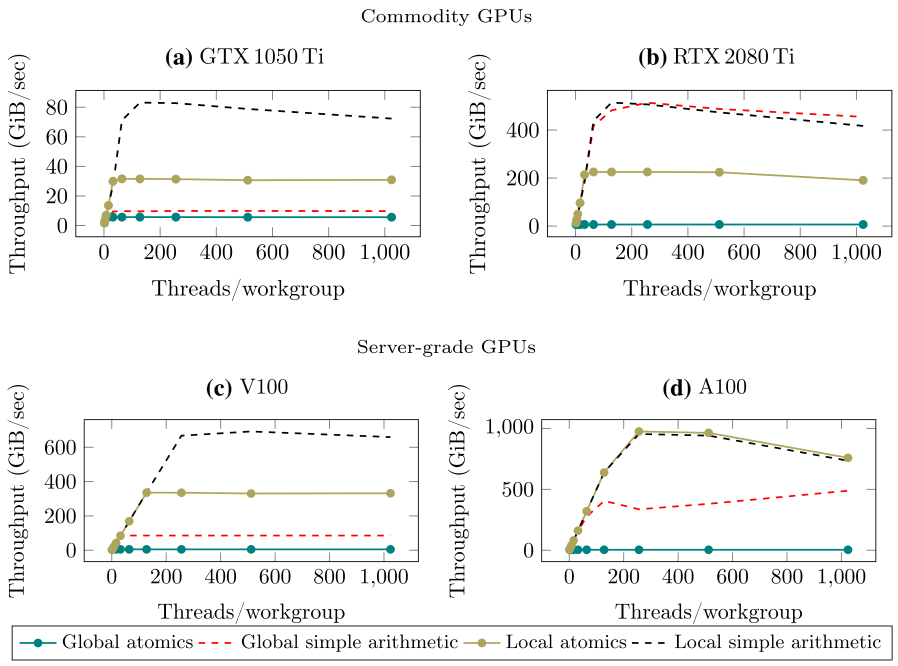

Fig. 3 Throughput for naive atomics and arithmetics on four different GPUs. More details on the GPU specifications are summarized in Table 1

2.2.2 Simple arithmetic operation as optimal throughput

To quantify the impact of atomic execution, we also execute a naive arithmetic operation on the same location, which incurs no synchronization. As their overall flow remains the same as atomics, this is a good way of quantifying the impact of atomics. We consider both the global and local memory of GPU for our experiment. The resultant throughput ranges across different GPU devices is plotted in Fig. 3, using 2^{27} integers values as input. The results suggest three insights:

(1) Comparing Fig. 3a–d, the throughput of atomics in *local* memory in newer generations has significantly improved (instead of being 60 % slower on GTX 1050 Ti, local atomics are only half as slow as local arithmetics on RTX 2080 Ti). A similar trend can also be seen in server-grade devices, with A100 having better local memory atomics. Overall, we see an increasing throughput with atomics for each newer generation.

(2) The throughput difference for arithmetics and atomics is large with local atomics having a penalty of 2.0x to 2.6x on commodity GPUs 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. In case of server-grade devices, the V100 shows a performance difference of 3x. Hence, we need to mitigate this atomics penalty to unleash the full parallel power of present-day GPUs. Notably, the

A100 features improved local memory atomics, having even nearly the same performance as arithmetics. It seems that this bottleneck—though still being true for operations on global memory—will be resolved in all newer versions.

(3) When using atomics, the best performance is reached with a small number of concurrent threads. In case of commodity GPUs (GTX 1050 Ti and RTX 2080 Ti) we see the atomics throughput flat-line after thread count reaches 16. With V100 and A100, the maximum atomic performance is reached at 128 and 256 thread counts respectively. Therefore, increasing the thread count after this critical threshold may 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 relying on local atomics rather than on global ones. Indeed, local atomics are faster as only a limited set of threads access a local memory space. However, relying on local atomics would require an additional synchronization step when combining the partial results in the local shared memory with the final result. Furthermore, the small size of local memory limits its use for group-by aggregation.

3 Atomics for sort-based aggregation

As we can infer from the previous section, multiple components are involved in atomic execution, which incurs 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.

3.1 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 form a sequence of groups. The second pass sequentially aggregates the groups present in the sorted input. To parallelize this processing for GPUs, additional phases are needed, as explained in the example of a COUNT aggregation below.

Once the data is sorted, aggregating the groups within a GPU requires three additional 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

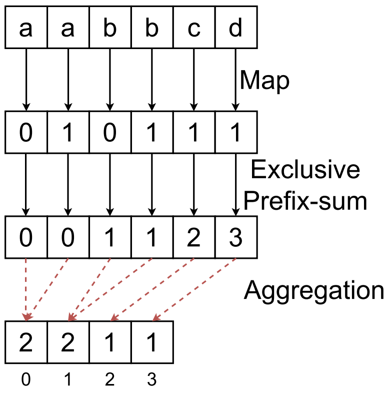

Fig. 4 Three-phase atomic COUNT aggregation

target positions from the prefix-sum. For this phase, our atomic-based aggregation is used to compute aggregated group-by results. We specifically explore atomics as it is the critical function necessary to aggregate the results in an input.

3.2 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 the group values inside a sorted array are sequential so 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 synchronization 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.

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.

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

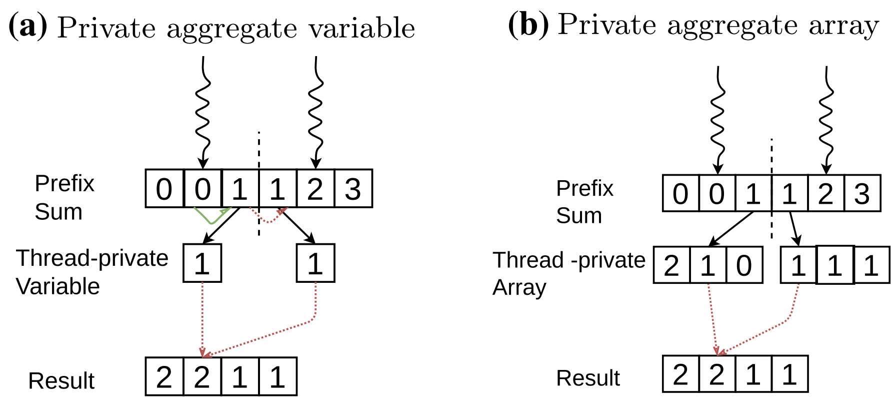

Fig. 5 Using private space in aggregation phase

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.

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 optimization 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, synchronization 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 synchronization having the *optimal* performance shown in Fig. 3 (global arithmetic).

4 Experiments

In this section, we evaluate our approaches using micro benchmarks and 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 four GPU versions with varying degrees of usage—NVIDIA GTX 1050 Ti,

Table 1 Device information of the GPUs used in our experiments

Device	Type	Architecture	Compute units	Local memory size (KiB)	Global memory size (GiB)
GTX 1050 Ti	Commodity	Pascal	6	48	3.939
RTX 2080 Ti	Commodity	Turing	68	48	10.76
V100	Server-grade	Volta	80	48	31.75
A100	Server-grade	Ampere	108	48	39.5.59

NVIDIA RTX 2080 Ti, NVIDIA V100 and NVIDIA A100. The device details are given in Table 1. We implement GPU driver code in C++ with variants in OpenCL[1]. All our experiments are executed on a Linux machine compiled 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.

4.1 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.

4.1.1 Examining optimal thread size for naive atomics

In this experiment, we identify the optimal thread size per workgroup for naive atomics serving as the baseline. 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 workgroup's local memory into the final result in the global memory. In this naive local variant, we perform merging similar to the approach used for our private array variant, where only the first and last positions are merged atomically. The throughput ranges for this experiment across GPU devices is depicted in Fig. 6.

Though, the specific throughput varies highly among the GPUs as expected, the overall result pattern is uniform across all devices. Concisely, the primary observation is that we observe the best throughput for large input groups, when spawning

[1] Code available here: https://git.iti.cs.ovgu.de/bala/DBPV/-/tree/master/Dispatcher/openCLDispatcher.

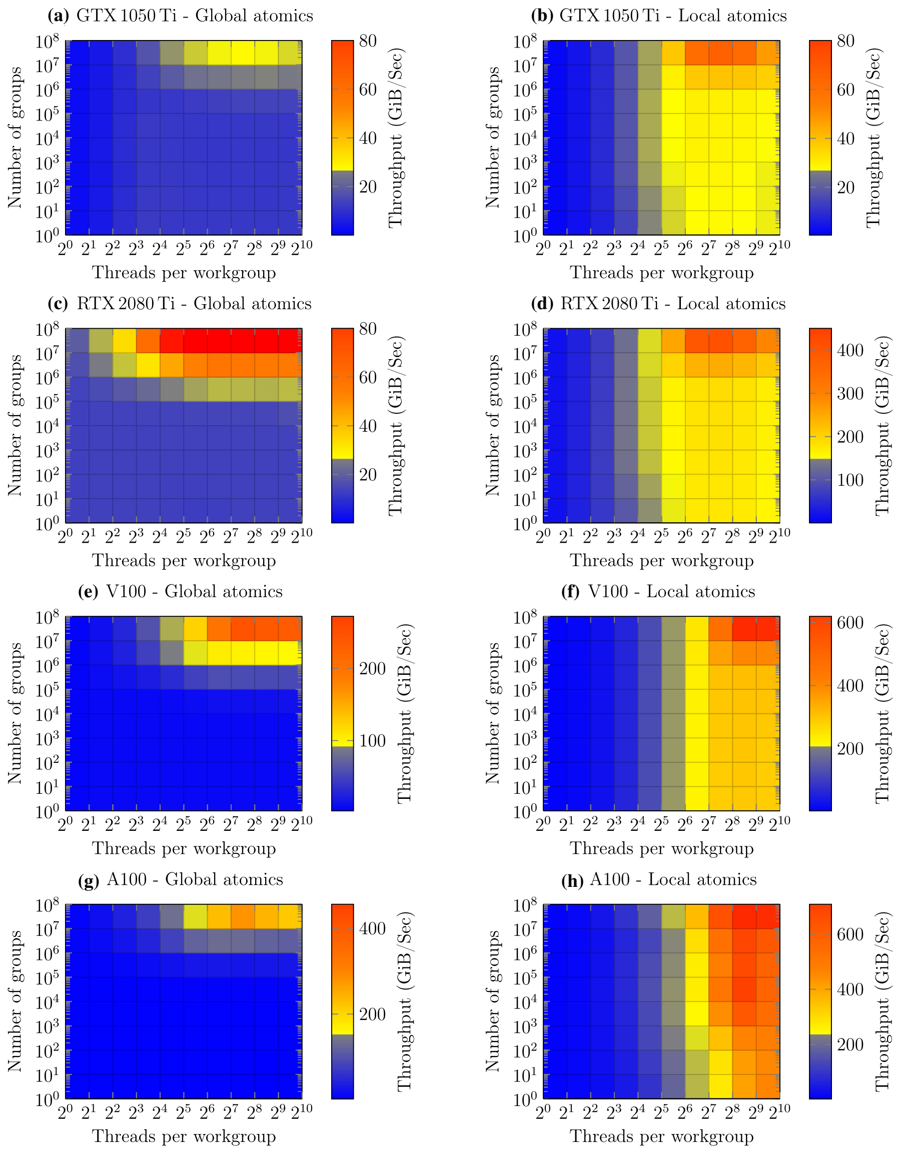

Fig. 6 Impact of varying group and thread sizes on the baseline variants. The left side contains the throughput of the local atomics variant, the right side the global atomics variant

the maximum number of threads. This holds for the local and global atomics variant. The rational is that multiple threads efficiently hide memory latency. Furthermore, a higher number of groups (i.e., a larger spread of target locations in memory) create less concurrency on atomic writes. Next, our results also clearly suggest an improvement from using local memory as cache for partial aggregates. The magnitude of the improvement is however device specific, ranging from approximately a factor of

2 for the GTX 1050 Ti and A100 to almost a factor of 5 for the RTX 2080 Ti. The improvement, thus, is smaller, as the pure speed difference of global and local GPU memory in isolation promises. The reason is the requirement of the extra merging step in the local atomics variant, which significantly reduces the overall throughput. Nevertheless, in any case, the improvement is significant.

As an overall result, the best thread sizes are 256 for GTX 1050 Ti & V100 and 1024 for RTX 2080 Ti & A100, which we then use to compare naive atomics with our approaches and the competitors.

4.1.2 Best thread and chunk size for atomic variants

In addition to 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 experimentally study the impact of this parameter on the throughput of our variants considering different chunk sizes and number of threads. To this end, we average the variants' throughput over all tested number of groups and plot the results in Fig. 7.[2]

In Fig. 7, we depict the heatmap that describes the impact of varying chunk and thread sizes on the throughput for all four private array/variable variants and all four GPU devices. Part (a) of Fig. 7 contains the results of all variants for the GTX 1050 Ti and part (b) contains the results for the RTX 2080 Ti, etc.

Universally, across all devices, we see that the private variable variant works better with medium-sized chunks (i.e., $2^2 - 2^7$). This is in opposition to the results of the naive atomics variants, where large chuck sizes are beneficial. Such a poor performance behavior with large chunk sizes for our variants is because of a bottleneck within the memory controller. The bottleneck arises due to too many 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. The negative impact of this effect increases for the variants running in local memory. This explains why the local memory variants prefer very small chunk sizes ($2^1 - 2^3$), whereas global memory benefits from slightly larger ones ($2^2 - 2^7$).

An additional interesting observation is that there is only a small difference between using a private variable and a private array to store intermediate results. On the contrary, the throughput behavior changes stronger w.r.t. devices, since there is a wide spectrum of well-performing variants on GTX 1050 Ti, which shrinks for the RTX 2080 Ti. This indicates that variants are sensitive to the underlying hardware and need a smart variant tuning procedure [17].

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

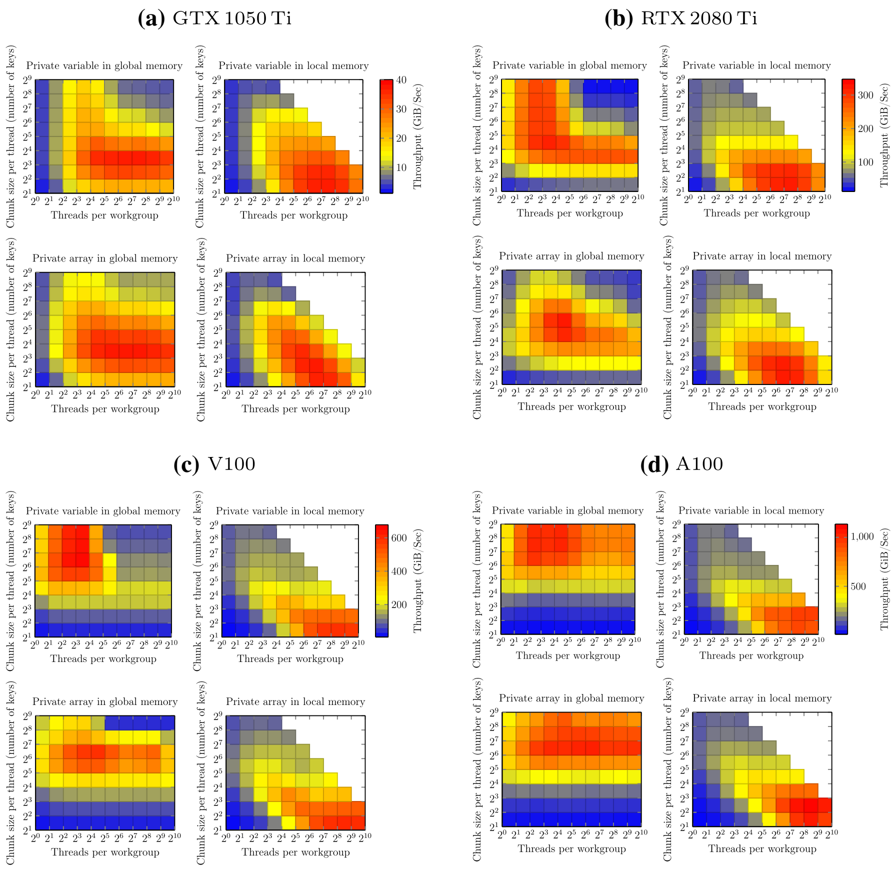

Fig. 7 Impact of varying chunk and threads sizes on the throughput for all four private array/variable variants and all four GPU devices

4.2 Comparative experiments

Based on the inferences above, now we can set the tunable parameters for each of our variants (and devices) to its optimal value. With this optimal setup, we can perform experiments to compare against state-of-the-art systems. Our experiments first study the performance impact across different variants. Next we compare against other state-of-the-art techniques (hashing and boost.compute's sort-based aggregation), which we then test in the following experiment for different data distributions and, finally, compare our GPU variants against CPU variants.

4.2.1 Comparison of variants

To compare against other baselines, we first identify the best variant of our approaches per device. To this end, we compare the throughput of different variants with their respective optimal parameter values. The results are shown in Fig. 8.

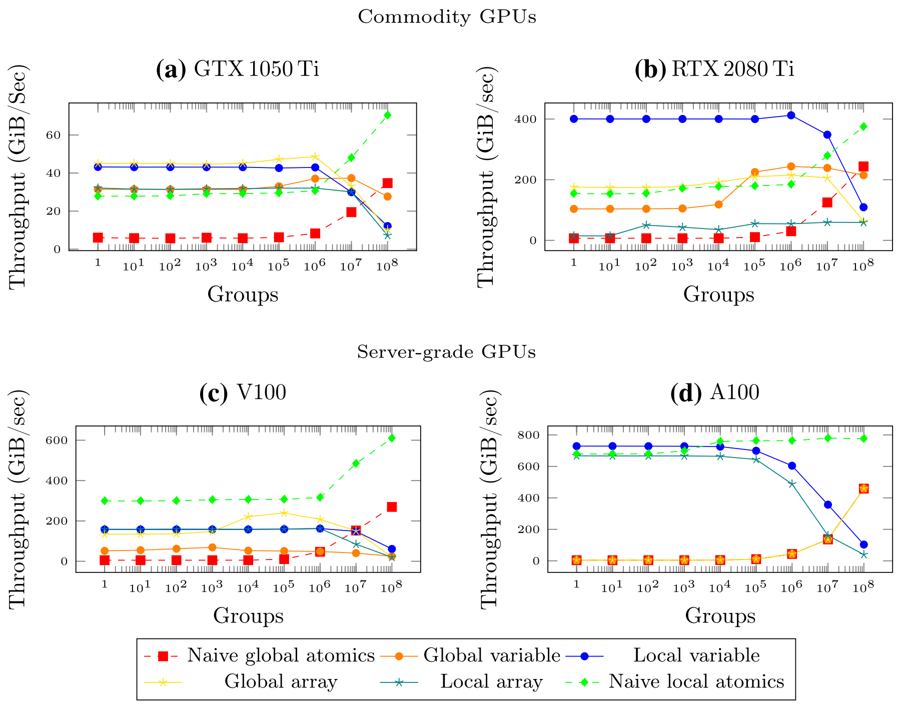

Fig. 8 Performance comparison of atomic variants

Our results show that the global array and local variable variants have higher throughput than the naive atomic variants for almost all group sizes. The only exception from this observation occurs for a large number of groups.

When comparing the GPU generations, we see the trend that throughput increases with newer versions of GPU. However, we also see that each device has its own throughput profile for variants. First in GTX 1050Ti, we see that global array is the fastest performing variant until the group size of 10^5, afterwards, naive local atomics takes over. The drop in global array performance is due to increasing number of atomic inserts from threads to the global memory. On the contrary, this is not a problem for naive local atomics as the push down from local to global has less atomic inserts. The variant also further benefits from local memory atomics in case of larger group sizes.

Next with RTX 2080 Ti, we see that local variable atomics is the fastest, except for group size 10^8 where again local atomics takes over. The change in variant performance can be inferred from the improvement in local memory bandwidth (cf. Fig. 3). The same inference can be made for the high throughput ranges for the local variable variant. Here, atomics over local memory is faster than accesses to global memory. The variant, which is a mix of these high bandwidth accesses, gives the best throughput ranges.

Finally, with the server-grade devices, we again see the local atomics to be the fastest performing variants altogether. With closer inspection, we see the throghput for the variant improves with groups for V100 whereas A100 seems to be consistent across the group sizes. We beleive that V100 behaves the same as RTX 2080Ti but runs with bigger bandwidth rates, as their architectures are similar (Volta and Turing, respectively). However for A100, local memory atomics have a minor impact leading to a nearly constant throughput range.

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. Overall, we also see only a small improvement using local memory for our variants on the GTX 1050 Ti, whereas with the newer devices we see local memory atomics improving significantly. This is consistent with the throughput results from local and global memory atomics given in Sect. 2.2. 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.

4.2.2 Comparison with hashing

As a next evaluation, we compare our performance with other state-of-the-art mechanisms. To this end, our best-performing atomic variants now include a sorting step (using boost.compute's sorting mechanism) before the aggregation step. We compare these against a sort-based aggregation of *boost.compute* (using sort_by_key() and reduce_by_key() functions, which we call *boost.compute* in the graph) and the hash-based aggregation by Karnagel et al. [12] (called *Hashing*). While the hash-based aggregation uses a pre-aggregation step in local memory up to 5120 groups, for bigger group sizes that do not fit local memory, the hash table is directly stored and accessed in global memory. As an additional indicator of performance boundaries, we also include the throughput for naive sorting for comparison to study the impact of the aggregation phases.

Our results in Fig. 9 reveal that our complex atomic variants mostly lead to the best performance. Also comparing naive sorting with atomic aggregation, we see that aggregation has a significant impact on throughput. Nearly 50 % to 75 % of the execution comes from executing atomics. 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, the state-of-the-art hash-based aggregation delivers its best performance for groups numbers between 1000 and 100,000. Here, the result pattern clearly differs from the result pattern of all variants using atomics. However, only on RTX 2080 Ti hashing is superior to the best atomics variant. 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. Our experimental results further suggest two additional insights. First, the throughput gains on server GPUs (i.e., V100 & A100) are larger than those gained on commodity GPUs. We attribute this to the increased number of atomic operations/sec that (more expensive) server GPUs support. Second, on server GPUs, the technical progress between V100 and

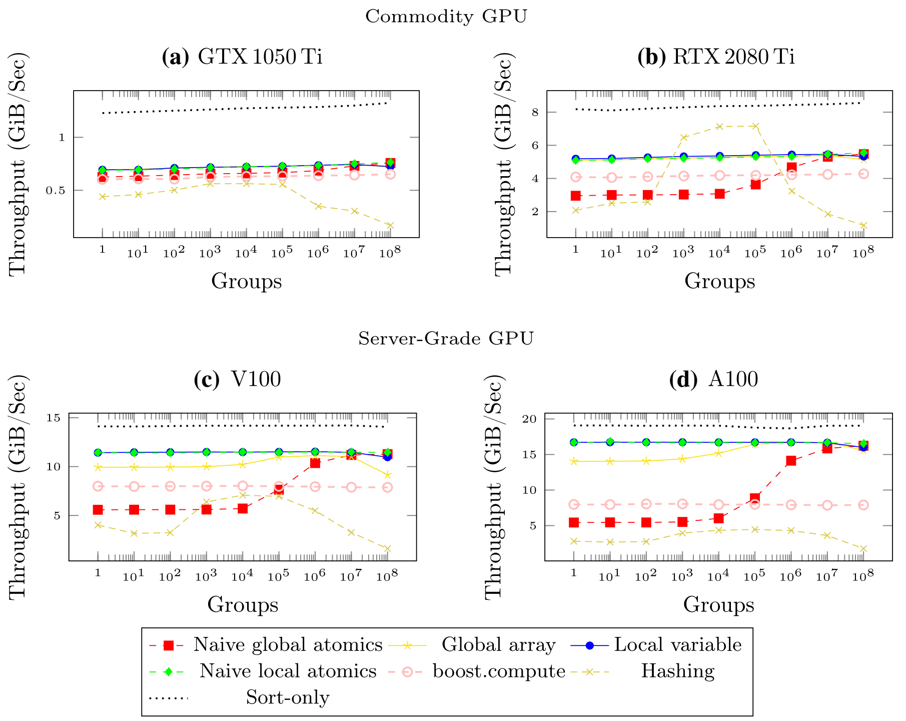

Fig. 9 Overall comparison against state-of-the-art competitors. The performance of atomic variants now includes sorting

A100 of about three years has notable impact on the throughput of our atomic variants. Specifically, we observe an increase of about 50 percent. Instead, the throughput of the hashing-based approach remains almost the same. That is, technical progress appears to increases the benefit of our solutions.

So far, we saw the performance of our variants with only random distribution. Next, we expand our evaluation with different data distributions.

4.2.3 Comparison across data distributions

In the previous section, we compared the different approaches with varying group sizes generated using random distribution. In this section, we expand our experiments by comparing the performance of our variants and other techniques with four different data distributions. Once again, we consider the input to be 2^{27} integer values. The distributions considered are: heavy hitter (90 % of input is a single group, total groups 6710886), random (10001 groups), exponential(lambda = 0.05, 6551904 groups), weibull (a=2.0 & b=4.0, 133958786 groups) and normal (134217733 groups). The corresponding throughput ranges across the different GPU devices is given in Fig. 10.

First, we can see a similarity between the results of RTX 2080 Ti and V100 while the other two devices behave differently. The similarity in the results

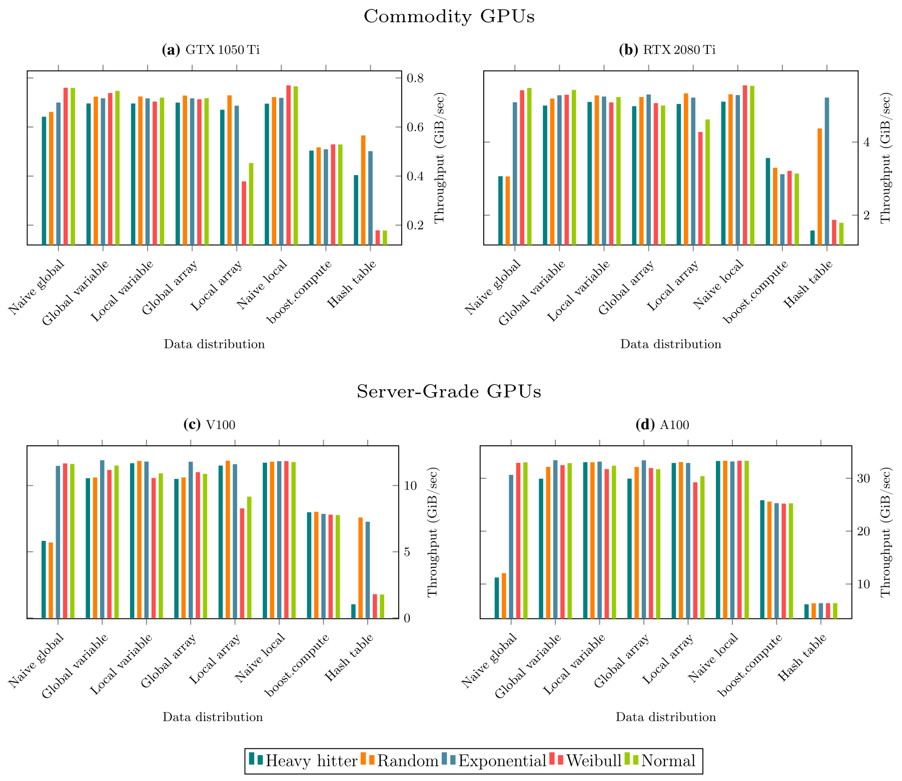

Fig. 10 Performance of aggregation techniques across various data distributions

confirms that the underlying architecture for RTX 2080 Ti and V100 are quite similar except their bandwidth.

Looking at the variants individually, naive global atomics performs poorly with heavy hitter & random distribution, but reaches better throughput ranges—comparable even to other variants for the remaining distributions. Better throughputs are only reached for larger groups in the input data. As we saw earlier, with larger groups, atomics congestion in the system is reduced—leading to a faster aggregation of results.

Additionally, naive atomics for heavy hitter & random distribution in GTX 1050 Ti is considerably faster than running on other devices. This is mainly because of the smaller internal bandwidth in the device. With newer generations, this bandwidth's impact is clear.

Next, with global variable, local variable and global array variants, the data distribution has a smaller impact. Again, depending on the number of groups present in the input, we see slight variations in the performance. However, there is no significant difference in the throughput ranges.

The local array variant has a drop in performance with weibull and normal distributions across all the devices. Again, looking at the number of groups, we see that there are more than 99 % of the input only unique values. This again explains the poor behavior of the variant as the number of inserts to the global memory is high, leading to multiple pushes into the global memory, thereby affecting performance.

Compared to our variants, the other techniques boost.compute & hashing fair poorly in terms of throughput. boost.compute shows only little impact due to distributions, as previously seen (cf. Fig. 9). Such behavior is because of the internal aggregation function used that follows a non-atomics based aggregation, which is unaffected by groups. On the contrary, hashing is affected by both the distributions as well as the device it is run on. We can see that distributions with large groups sizes leads to poor throughput performance of the hashing technique, which is expected. In A100, hashing has the worst throughput across all devices. We believe this is mainly due to the random access of locations/buckets for both insertion and aggregation. Here, the device cannot pack the values together that they insert and aggregate one data at a time, leading to the poor performance.

Furthermore, the results for server-grade GPUs are consistent with each other in terms of their relative performance. The throughput ranges are higher for A100 compared to V100 due to its higher CUDA core count. However, in the case of commodity GPUs, hashing is comparatively faster for RTX 2080 Ti than for GTX 1050 Ti when subjected to random distribution.

4.2.4 Comparison with CPU

Due to a high degree of parallelism, GPUs are surely expected to have a higher throughput than a CPU. Nevertheless, with our experiments comparing all our atomics variants on GPUs to their counterpart on CPUs, we intend to study two objectives. Firstly, we want to examine whether our approaches and the results presented above are GPU specific, or whether one can generalize them. Secondly, we examine whether there is a *significant* improvement of using a GPU compared to a CPU, i.e., whether buying special hardware pays off.

To compare against the CPU, we run the same atomic-based aggregation and hash-based techniques in the Intel-Xeon Gold 5220R CPU (using all cores) and compare its throughput against the A100 GPU. That is we compare the newest GPU we consider to the latest CPU model, we have at hand.

As we see in Fig. 11, the throughput ranges of a GPU is in the order of 10x faster than that of a CPU. Additionally, the throughput of aggregation using a GPU has a clear difference in throughput across atomics and other techniques, whereas the hash-based and boost.compute-based aggregation in CPU are competitive with each other. In general, aggregation runs orders of magnitude faster, even with atomics, due to the efficient serialization of aggregation from parallel threads. Furthermore, we see a significant impact of the aggregation step, as the throughput drops significantly when comparing the naive sorting with sorted aggregation.

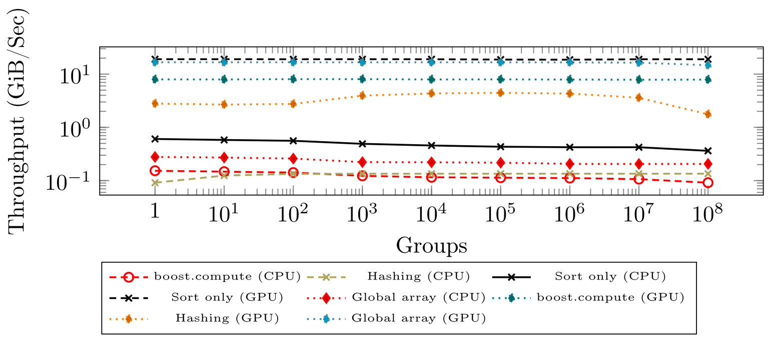

Fig. 11 Throughput comparison of grouped aggregation in CPU (Intel Xeon) and GPU (A100)

4.3 Discussion of core results

In summary, we observe that for the common use case of up to some hundred groups,[3] 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:

- 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. 9 (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 [18].
- Due to increased local memory performance of modern GPUs, the overhead of atomic operations can be effectively mitigated. Our results have also shown that with newer GPU versions, this performance advantage even increases.

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

4.4 Discussion of threats to validity

The core results presented above are primarily supported by empirical observations, i.e., experiments comparing our solutions to the state-of-the-art on present-day hardware. This method naturally faces threats to validity. This holds for the global findings and specifically for the identified break-even points. Therefore, to strengthen

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

validity of our results, we discuss the most relevant threats next. The threats are competitor implementation, used sorting technique, and programming interface.

4.4.1 Competitor implementation

In our study, we rely on the publicly available implementation of the hashing approach of Karnagel et al. [12] as state-of-the-art competitor. This approach uses a pre-aggregation in *local memory* for all use cases where the number of groups fit the local memory. Then, in a second step, the final result is computed by merging the local pre-aggregates in global memory. On the hardware that we use, this is possible until a group cardinality of 5120 groups. The rational for this two-step approach is that any work done in local memory is significantly faster than in global memory. When having more groups, one stores the hash table in global memory only.

Our experiments suggest that there is only a small benefit of using local memory. The reason is that the bottleneck is the final aggregation step in global memory, which is issuing too many atomic operations. Thus, in our plots, we do not distinguish between both variants.

We hypothesize that all work-groups, aggregating over the same amount of data, approximately finish simultaneously in local memory. Then, they simultaneously want to propagate their local results to the global memory. Since the number of groups they want to write to is small, they all access the same or similar memory locations causing heavy congestion of concurrent write accesses. Logic countermeasures within the approach of Karnagel et al. [12], such as using a different hash function per work group provoking a heavier distribution of accessed memory locations, do not shown an observable improvement. The same holds for trying to balance concurrent access (e.g., not all work groups start propagating their result starting with the first group). Nevertheless, it may be possible with more fine-tuned hashing techniques that the threshold of when sorting beats hashing can change. This, however, would mean to change core components of the approach of Karnagel et al. [12], which is, according to understanding, a novel approach.

4.4.2 Used sorting technique

Our contributions aim at fast merging of sorted runs to efficiently compute aggregates. Despite of this, the used sorting technique has a pivotal impact on the overall performance. In our experiments, we chose the best sorting technique available in OpenCL. Generally, any sorting technique can be used. Assuming there is a more efficient sorting technique, e.g., proposed in the future work or in NVIDIA-specific CUDA, this would strengthen the impact of our contribution. That is, the overall throughput of our sort-based aggregation would depend even more on the right atomic variants to be used, as the bottleneck of sorting is reduced. Therefore, we argue that our results of investigating the best atomic aggregation variant hold independent of the sorting technique and potential future improvements of sorting techniques.

4.4.3 Programming interface

Another impact factor to the reported performance is the used programming interface. Our variants are implemented using OpenCL, which is known to be portable, but also lacks performance compared to NVIDIA-specific CUDA. We make this choice since (1) many libraries have been written in OpenCL, which gives a variety of implementation alternatives to choose from and (2) also the competiting hashing technique has been implemented in OpenCL. Of course, CUDA-based implementations are valid alternatives (e.g., the implementations of the CUB library) and, thus, are an important future work.

5 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 [19]. Hoseini et al. explore the impact for atomics on CPUs [20]. Our approach combines these two approaches in exploring atomics for aggregation in modern GPUs. Our results show the benefits of using atomics for aggregation in GPUs.

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 costs than using atomics. Instead of these passes, our work uses atomics to reduce the number of data access.

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, 21–23]. However, our results show that random access in hashing degrades performance whereas sort-based aggregation has uniform access assisting in improving performance.

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, 24].

6 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.

Acknowledgements This work was partially funded by the DFG (Grant nos. SA 465/51-2 and PI 447/9-2).

Author contributions Conceptualization: BG, DB, MS; Methodology: BG, DB, MS, TP, GS; Formal analysis and investigation: BG, DB, MS, TP, GS; Writing—original draft: GB, DB; Writing—review and editing: BG, DB, MS, TP, GS; Funding: TP, GS; Supervision: DB, GS.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Competing interests The authors declare no competing interests.

References

1. Chen, M., Mao, S., Liu, Y.: Big data: a survey. Mob. Netw. Appl. **2**, 171–209 (2014)
2. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUDA. GPGPU 94–103 (2010)
3. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: SIGMOD, pp. 215–226 (2004)
4. Wu, H.: Acceleration and execution of relational queries using general purpose graphics processing unit. In: GPGPU (2015)
5. Arefyeva, I., Broneske, D., Campero, G., Pinnecke, M., Saake, G.: Memory management strategies in CPU/GPU database systems: a survey. In: BDAS, pp. 128–142. Springer, New York (2018)
6. Becher, A., Bg, L., Broneske, D., et al.: Integration of FPGAs in database management systems: challenges and opportunities. Datenbank-Spektrum **18**, 145–156 (2018)
7. Breß, S., Funke, H., Teubner, J.: Robust query processing in co-processor-accelerated databases. In: SIGMOD, pp. 1891–1906 (2016)
8. Boncz, P., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: TPCTC, pp. 61–76 (2013)
9. Gurumurthy, B., Broneske, D., Pinnecke, M., Durand, G.C., Saake, G.: SIMD vectorized hashing for grouped aggregation. In: ADBIS, pp. 113–126 (2018)

10. Behrens, T., Rosenfeld, V., Traub, J., Breß, S., Markl, V.: Efficient SIMD vectorization for hashing in OpenCL. In: EDBT, pp. 489–492 (2018)
11. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. **34**(4), 1–39 (2009)
12. Karnagel, T., Müller, R., Lohman, G.M.: Optimizing GPU-accelerated group-by and aggregation. In: ADMS, pp. 1–12 (2015)
13. Gurumurthy, B., Broneske, D., Schäler, M., Pionteck, T., Saake, G.: An investigation of atomic synchronization for sort-based group-by aggregation on gpus. In: ICDE Workshops 2021, pp. 48–53. IEEE (2021)
14. Owens, J.: Gpu architecture overview. SIGGRAPH **10** (2007)
15. Aamodt, T.M., Fung, W.W.L., Rogers, T.G.: General-purpose graphics processor architectures. Synth. Lect. Comput. Architect. **13**(2), 1–140 (2018)
16. Glasco, D.B., Holmqvist, P.B., Lynch, G.R., Marchand, P.R., Mehra, K., Roberts, J.: Cache-based control of atomic operations in conjunction with an external ALU block. Google Patents. US Patent 8135926 (2012)
17. Rosenfeld, V., Heimel, M., Viebig, C., Markl, V.: The operator variant selection problem on heterogeneous hardware. In: ADMS (2015)
18. Kim, C., Kaldewey, T., Lee, V.W., Sedlar, E., Nguyen, A.D., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. Proc. VLDB Endow. **2**(2), 1378–1389 (2009)
19. Hauck, M., Paradies, M., Fröning, H.: Software-based buffering of associative operations on random memory addresses. In: IPDPS, pp. 943–952 (2019)
20. Hoseini, F., Atalar, A., Tsigas, P.: Modeling the performance of atomic primitives on modern architectures. In: ICPP, pp. 1–11 (2019)
21. Tome, D.G., Gubner, T., Raasveldt, M., Rozenberg, E., Boncz, P.A.: Optimizing group-by and aggregation using GPU-CPU co-processing. In: ADMS, pp. 1–10 (2018)
22. Yuan, Y., Lee, R., Zhang, X.: The yin and yang of processing data warehousing queries on GPU devices. Proc. VLDB Endow. **6**(10), 817–828 (2013)
23. Lee, R., Zhou, M., Li, C., Hu, S., Teng, J., Li, D., Zhang, X.: The art of balance: a RateupDB experience of building a CPU/GPU hybrid database product. Proc. VLDB Endow. **14**(12), 2999–3013 (2021)
24. Lauer, T., Datta, A., Khadikov, Z., Anselm, C.: Exploring graphics processing units as parallel coprocessors for online aggregation. In: DOLAP, pp. 77–84 (2010)