Code fusion of prefix sum with ordered aggregation

Akanksha Saxena, Harika Chandamollu, Imran Sheikh, Niha Mohanty, Sneha Videkar, Yash Shah
Data and Knowledge Engineering
Otto Von Guericke University
Magdeburg, Germany
Email: akanksha.saxena@st.ovgu.de, harika.chandamollu@st.ovgu.de, imran.sheikh@st.ovgu.de,
niha.mohanty@st.ovgu.de, sneha.videkar@st.ovgu.de, yash1.shah@st.ovgu.de

***Abstract*—Ordered aggregation is one of the most frequently used queries for grouping data in business applications.These are generally processed sequentially by a CPU and hence are time consuming. In order to improve its efficiency, a prefix sum based parallel processing technique is followed. This technique requires 5 different parallel kernels to be run for providing the output. The synchronization time between these kernels leads to inefficiency in processing. In this work, we fuse 5 kernels for performing ordered aggregation using prefix sum into 2-kernel and 3-kernel architectures and compare it against 5-kernel architecture. Additionally, we improve the performance of these code-fused kernels by providing different variants, namely, direct atomics, chunked atomics, private variable and private array. For this, we first find the best architecture out of the three architectures, five, three and two and local size with unique and reduce data. We then use the winner architecture to find the optimal chunk size for unique and reduce data. We then evaluate the performance of winner architecture using the found local size and optimal chunk on the four variants. Overall, we provide a code-fusion technique for ordered aggregation and evaluate them based on hardware sensitive parameters. Our results show that using the optimal variant improves the throughput of up to 0.37x the base variant.**

***Index Terms*—Pre-Fix sum, ordered aggregation, code fusion**

I. Introduction

Todays processing systems have undergone a paradigm shift from using single-core systems to multi-core systems. This shift to multi-core systems is due to a huge surge of data volume. Voluminous data computations require systems with heterogeneous devices that support parallel execution of data. Parallel execution is supported by multi-processor devices with multiple CPU cores and numerous GPU cores. Execution on GPU is comparatively faster with a better response time for its operations than in a CPU. However, GPU can only perform simple and repetitive tasks more efficiently owing to the abundance of cores and not being as intelligent as the CPU. This is exactly what comes handy for executing ordered aggregation queries on large amounts of data. We explore the parallel processing capabilities of GPU for optimizing ordered aggregation. As already mentioned Ordered aggregation is one of the most frequently used queries for grouping data in business applications and is performed sequentially. In this work, we aim at reducing the execution time or increasing the throughput of ordered aggregation queries. Impact of hardware components on database query processing is a well-known research topic in the database field. P. Boncz et al. has analyzed TPC-H queries to find potential choke points [1]. The study suggests that utilizing hardware components capabilities will lead to performance improvement of analytical queries. This has encouraged several research projects to explore hardware component capabilities for optimizing ordered aggregation queries [3]. In this paper, we focus on code fusion of prefix sum using ordered aggregation which will be discussed in detail later.

The remainder of this paper is structured as follows. In the next section, we give a brief idea about the background knowledge on ordered aggregation, prefix-sum operation, and GPU architecture. During the project research, we came across a few related works. We list a few related research work papers which helped gain more detailed knowledge about the concept of the project in Section III. This also covers most related papers which are closely related to our work. Section IV gives a brief idea about the implementation which was done as part of this project. The evaluation performed for the implemented code are briefly explained in Section V. Finally, we conclude our paper in conclusion Section VII.

II. Background

In this section, we provide a brief background about ordered aggregation operation. We then discuss prefix sum operation and 3 step parallel prefix sum in detail. In the end, we discuss GPU parallelization.

A. Ordered aggregation

Ordered aggregation queries are typical queries used for statistical purposes. A simple example of ordered execution query is shown in Fig. 1. This query retrieves the number of students for each grade from Student database.

Unlike normal aggregation queries, ordered aggregation queries are executed on sorted data. Ordered aggregation queries are very frequently used queries. The execution of the ordered aggregation query is time intensive [2]. The underlying ordering operation for calculating the aggregate for each group can be explored further in such analytical

queries instead of using hashing for aggregating techniques. Ordered aggregation is mostly used to generate group-wise statistics.

```
SELECT Grade, COUNT(*)
FROM Student
GROUP BY Grade
ORDER BY Grade;
```

Fig. 1. A typical statistical query

An ordered aggregation is a reduce operation that takes a sorted vector as input and provides group-wise result as output. Ordered aggregation is executed by reading already sorted data and aggregating based on the groups present in the input data. Generally, ordered data is read sequentially and then the aggregation technique is applied to it which makes it a time consuming operation. The response time of ordered aggregation also increases with the increasing amount of data and especially if ordered aggregation is performed on large-scale business data. There is, therefore, a need to increase the efficiency of ordered aggregation operations by using parallel processing capabilities of the processors. Parallelization of ordered aggregation can be achieved by partitioning the input data into equal chunks and simultaneously applying multiple threads on these chunks. The idea, therefore, is to divide the task of ordered aggregation between various threads where each thread evaluates a chunk in parallel. However, on doing so there arises a few challenges.

The first challenge is contention between multiple threads. As different chunks may have same input data, aggregating and writing on same memory location may be required which leads to contention. The solutions to overcome this challenge is to apply prefix sum algorithm which will be performing atomic adds on the memory. The second challenge is identifying the final positions of the resultant array in the memory. Prefix sum algorithm also assists in exacting memory locations of the input data for ordered aggregation.

The ordered aggregation operation is also extended to GPU for faster execution owing to its abundance of cores. Also, ordered aggregation in GPU is performed atomically. An atomic operation means that one thread can operate on one memory location at a time and not be interrupted by another thread that is in contention of writing on the same memory location with the same data. This is achieved by a locking mechanism of the memory and it also helps in avoiding contention between threads. In the following section, we discuss prefix sum operation.

B. Prefix sum operation

Prefix sum is an operation that takes an ordered array of data values as input. It then computes and provides an equal sized array as the output as shown in Fig. 2.

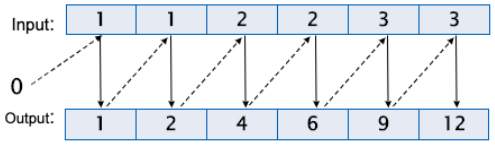

Fig. 2. Prefix sum operation

Every element of the output array is computed by adding the input element to the previous output element. For example, value at position 2 of the output array is computed by adding value at position 2 of the input array and value at position 1 of the output array [3].

Fig. 2 provides a better explanation of prefix sum operation. Here, the input array and output array are each of length 6. The first output element is calculated by adding a 0 to the first input element. Subsequently, the second output element is calculated by adding the first output element to the second input element. Thus, each element of the output array is calculated by adding all previous elements to the current element.

The prefix sum algorithm is a simple yet efficient algorithm for parallel processing. This is primarily because prefix sum operation provides synchronization of all the work items which are running in parallel. In addition, the prefix sum operation provides the location of the result. For these reasons, the prefix sum algorithm is a preferred implementation choice.

C. 3-Step parallel prefix sum

In this section, we will delve deep into 3- step parallel prefix sum method. The 3- step parallel-prefix sum method is used as it helps in avoiding the contention of parallel work items. The input data is assumed to be ordered. The data is also mapped such that if the consecutive elements are same, they are assigned a 0 and if there is a change in the consecutive elements, it is assigned a 1. In addition, the data is equally chunked where one thread works on each chunk or work item and all threads work in parallel. This mapped vector of data is then given as input to the 3-step prefix sum algorithm, Fig. 3.

1) Partial prefix sum computation: First, the partial prefix sum of the mapped input data is computed. As data is chunked and is being processed in parallel, the prefix sum is computed individually for each chunk. The output of this step will be the input to the next step.

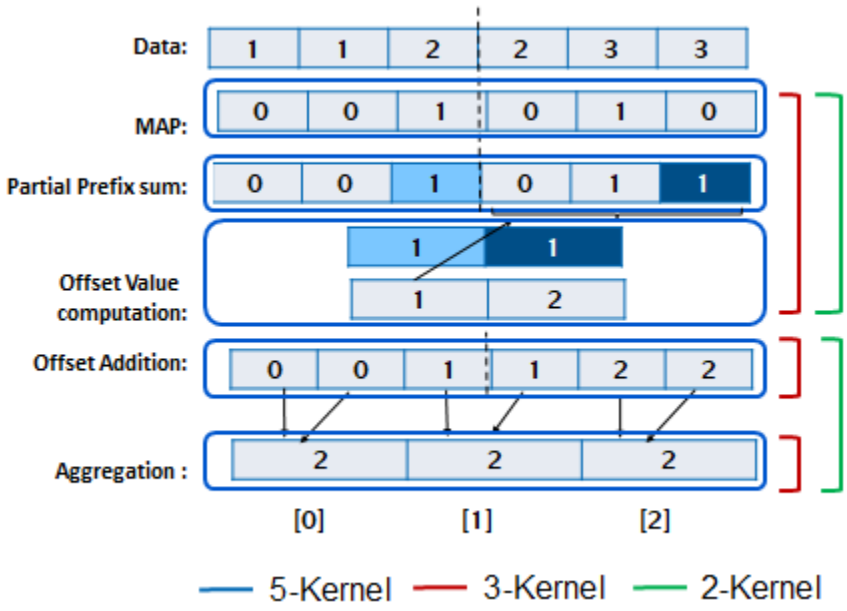

Fig. 3. Ordered aggregation using 3 step parallel- prefix sum

2) Offset value computation: Offset value computation takes in partial prefix sum as an input and gives an offset value vector of size equal to work items as output. This offset value is computed in two steps. First, the last digits of all the partial prefix sum chunks are only considered and stored separately. Second, the prefix sum is applied to it. The resultant vector is the Offset Value.

3) Offset addition computation: In the third step of prefix sum computation, the offset value evaluated in the previous step is back-propagated to the partial prefix sum vector. The first element of the offset value vector is added to each element of the second chunk of partial prefix sum vector. Subsequently, the second element of the offset value vector is added to each element of the third chunk and so on. No value is added to the first chunk. Also, the last value of the offset vector is ignored. The resultant array is the position vector of the input array.

Once the partial prefix sum is computed, the location of each element of the input array is known. These locations are then used to find the aggregate of the input array. Each work item working in parallel can then update the values in the aggregation array based on the location it holds atomically.

Ordered aggregation using parallel prefix sum is a 5-kernel process, as shown in Fig. 3. In the first kernel, input data is stored and mapped to 0s and 1s based on mapping technique. In the second kernel, partial prefix sum is computed for the mapped input. The offset value is computed in the third kernel and offset addition, in the fourth kernel. Eventually, the aggregation is performed in the fifth kernel.

Executing ordered aggregation in five different kernels has some performance drawbacks. Hence, for this project, we will try to fuse the kernels together and extend the ordered aggregation operation to GPUs for performance evaluation.

D. GPU parallelization

The processing of ordered aggregation of a large amount of data on a Central Processing Unit (CPU) is time consuming [1]. Similar operations on a Graphical Processing Unit (GPU) takes less time due to high computing prowess of the GPUs [1]. Both CPU and GPU consist of multiple cores, however, GPUs have a large number of contexts(thread) and hence provide highly efficient parallel processing abilities. In addition to that, GPUs provide competent memory optimization techniques [1].

To implement ordered aggregation using prefix sum algorithm, we intend to use GPU's parallel processing capabilities. Furthermore, the open computing language (OpenCL) framework will be used for the implementation of 2-kernel and 3-kernel architectures on GPU. OpenCL programming is platform independent, hence the code can be executed on various different GPUs. The following section provides insights into related and relevant work for prefix-sum and ordered aggregation operations.

III. Related work

Hardware devices like CPU, or GPU play an important role in relational query processing. Prerequisites of any query processing are initialization, management of data, and I/O transfer between CPU and GPU. Bingsheng et al. focused

on improving the time required for initial activities only, however optimization of query handling by GPU was not addressed [4]. In our project, we will analyze the working of GPU for improved query handling.

Likewise, P. Bakkum et al. suggests that it is not always necessary to implement kernel code for better performance. In this paper, open source database platform SQLite was considered for research purpose. SQL query performance was accelerated by implementation of SQLite virtual machine instead of CUDA kernel. However, there were many limitations addressed for SQLite approaches like memory limitations and memory transfer time [5].

Neumann explains query processing in the CPU. The study suggests that the performance of the handwritten program for aggregation query performs efficiently if compared to fast vectorized systems . So he created a compilation framework, which concentrates on data rather than an operator. However, data were linearly processed and CPU registers were used to achieve data locality. In contrast to this, we will make use of GPU parallelization concept for processing of data in parallel [6].

Few studies have considered hash based grouped aggregation for research. T. Karnagel et al. proposes hash-based grouping has an advantage up to a certain point. After the cut-off point, the sorting mechanism provides better speed as compared to the hash-based technique despite having a high initial cost [2]. The prerequisite for our project is to use sorted input data.

Adam Morrison et al. has introduced synchronization primitives like Compare-and-Swap (CAS) or Fetch-and-Add (FAA) in their research [7]. Likewise, Hermann Schweizer et.al has analyzed the performance of atomic operations (atomic) that uses the concept of synchronization primitives. CAS and FAA primitives are mentioned above are unavoidable in parallel processing. Both FAA and CAS builds diverse lock-free and wait-free algorithms.The study suggests that Compare-and-Swap is comparatively slower than Fetch-and-Add [8]. In our project, we have used the FAA atomic operation concept in which a value is fetched from a memory location and is stored into a register. Later on, this value is added to the previously fetched value from register to memory location.

Generation of thousands of variants for even simple operations like selection and aggregation has been demonstrated by Rosenfeld et al. This paper also describes selecting the right variant and selection algorithm, where picking of optimal variant is strongly device dependent. Their study is closely related to ours as they have used operator variants for selection and aggregation operations. However, they do not focus on operations like ordered aggregation [9]. In order to avoid maintenance and development cost caused by hand-written code (which can be highly optimized), they have used heterogeneous parallel programming framework OpenCL, which offloads the creation of device-specific code. In our project, we will be analyzing different operator variants for the specific device configuration using the OpenCL framework.

IV. Implementation

As already mentioned that in our project we are attempting to evaluate the performance of ordered aggregation using prefix sum on 3-kernel architecture as opposed to the current 5-kernel architecture. In Fig. 3, we show the various operations that are performed for the baseline architecture. Ordered aggregation using prefix sum in 5-kernel architecture is processed chronologically as shown in Table I.

TABLE I
5-Kernel Architecture

Kernel	Operations
1	Bit masking of Input data
2	Partial prefix sum
3	Offset computation
4	Offset addition or Back propagation to provide prefix sum
5	Aggregation

In this regard, we combine kernels such that there are only two kernels. The first kernel is for mapping and partial prefix sum computation and in the second kernel offset value computation, offset addition and aggregation is performed. The detail of our execution architecture is given below.

A. 3-Kernel architecture

Unlike the 5-Kernel architecture where one kernel is responsible for a particular task of ordered aggregation of input data. The 3-kernel architecture is modeled to fuse multiple operations into one kernel.

TABLE II
3-Kernel Architecture

Kernel	Operations
1	Partial prefix sum and Offset computation
2	Offset addition or Back propagation to provide prefix sum
3	Aggregation

Table II shows the chronological execution of various operations of different kernels in 3-Kernel architecture.

1) Kernel 1 - partial prefix sum and offset computation: As this is an ordered aggregation we are therefore dealing with sorted input data and we also assume that the size of input data is in the order of 2^N. The input data is divided into equal chunks. The elements inside each chunk define the chunk size or intent. The intent is also referred to as a work item. The number of chunks is evaluated by dividing

the total element size by intent (chunk size). The chunking also decides the number of threads that are to be spawned with distinct IDs starting from 0. However, the spawning of threads is random. The threads operate on their respective chunks i.e. the thread with ID 0 operates on the first chunk, the thread with ID 1 operates on the second chunk and so and so forth.

The partial prefix sum is calculated by comparing the values of an array in the global memory. The size of the global array is the input data size. A thread with a particular ID compares the elements of its respective chunk. It compares the first element of the chunk with the previous element in the global array and if the values are same then the resultant partial prefix sum array is assigned with 0 if not 1. The thread is then moved to the next element in the chunk and a similar comparison is performed. The result of the comparison i.e. 0 or 1 is then added with the value of the previous element of the chunk of the resultant array and is stored in the next position of the resultant array. After all the threads have finished executing their respective computations on their respective chunks, then the partial prefix sum is fully obtained in the resultant array.

Offset computation is calculated by taking the last element of all the chunks in the resultant partial prefix sum array. Meaning the resultant offset array has the size of the number of chunks in the resultant partial prefix sum array. A thread with a particular ID in the partial prefix sum array works on the respective offset array position and performs an atomic add to ultimately obtain the offset values in the resultant offset array. In an atomic add operation, the last value from the resultant partial prefix sum array is added to the respective position of the resultant offset array and to every other position to its right. The atomic operation happens sequentially one thread at a time. After all the threads have completed their atomic adds then the resultant offset array is fully obtained. The Fig. 4 is a pictorial representation of the description.

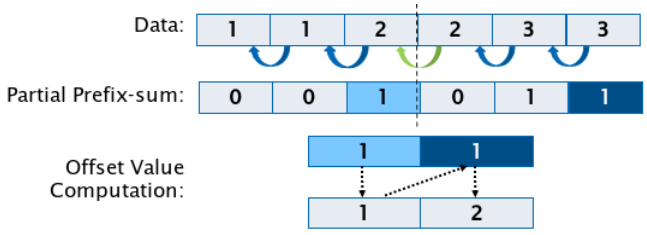

Fig. 4. Partial prefix sum Computation

CF No. 1 also represents the above process and the partial prefix sum is computed in the first **for** loop and the offset value is computed in the second **for** loop.

Algorithm 1 Partial prefix sum

Require: $Intent,\ Partial\ Prefix\ Sum\ Array,\ Offset\ Array\ \&\ Sorted\ Array$
Ensure: $INTENT \leftarrow Intent,$
$ARR \leftarrow Sorted\ Array,$
$PPS\_ARR \leftarrow Index\ Array\ \&$
$OFFSET\_ARR \leftarrow Offset\ Array$

$mIndex \leftarrow (thread\_id * INTENT)$
$mIndexLast \leftarrow (mIndex + INTENT)$

for $i \leftarrow mIndex,\ mIndexLast$ **do**
$prev\_index\_val \leftarrow ((i == mIndex)\ ?$
$0 : PPS\_ARR[index - 1])$

$PPS\_ARR[i] \leftarrow (i == 0)\ ?$
$i\ : !(ARR[i]\ ==\ ARR[i-1])\ +\ prev\_index\_val$
end for

$last\_offset\_val \leftarrow PPS\_ARR[mIndexLast - 1]$

for $k \leftarrow thread\_id,\ Offset\_size$ **do**
$atomic\_add(OFFSET\_ARR[k],\ last\_offset\_val)$
end for

2) Kernel 2 - offset addition or back propagation to provide prefix sum: The partial-prefix sum and offset array derived from kernel 1 is used as input for computing prefix sum in kernel 2. The first offset value of the offset array is added to all the values of the second chunk of the partial prefix sum array. The second offset value of the offset array is added to all the values of the third chunk of the partial prefix sum array and so and so forth, except for the last value of the offset array which is discarded. However, the first chunk of the partial prefix sum array is kept as it is in the prefix sum array as the first offset value is derived from the first chunk of the partial prefix sum array. The Fig. 5 is a pictorial representation of offset addition. Thus, in kernel 2 prefix sum is obtained by offset addition or back propagation of offset values which are then added to the partial prefix sum array.

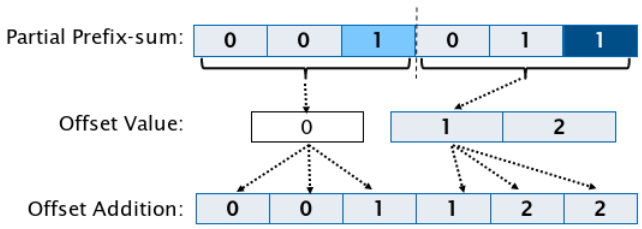

Fig. 5. Offset value addition

The CF No. 2 mentioned here also represents the same process and the **for** loop indicates the process of how prefix

sum is computed.

Algorithm 2 Offset value addition

Require: $Intent,\ Partial\ Prefix\ Sum\ Array,\ Offset\ Array\ \&\ Prefix\ Sum\ Array$
Ensure: $INTENT \leftarrow Intent,$
$PPS\_ARR \leftarrow Index\ Array$
$OFFSET\_ARR \leftarrow Offset\ Array\ \&$
$PS\_ARR \leftarrow Sorted\ Array,$

$mIndex \leftarrow (thread\_id * INTENT)$
$mIndexLast \leftarrow (mIndex + INTENT)$
$mOffsetVal \leftarrow (!thread\_id)\ ?$
$0 : OFFSET\_ARR[thread\_id - 1]$

for $i \leftarrow mIndex,\ mIndexLast$ **do**
$PS\_ARR \leftarrow mOffsetVal\ +\ PPS\_ARR[i]$
end for

3) Kernel 3 - aggregation: Prefix sum derived from kernel 2 is used as input in kernel 3 for performing ordered aggregation. The size of the ordered aggregation array is calculated by adding 1 to the last value of the prefix sum array. The values of prefix sum array provide the location of elements of ordered aggregation array. Threads spawned for chunks of the prefix sum array executes each chunk by performing an atomic add of 1. After all the threads have completed executing their respective chunks the resultant ordered aggregation array is achieved. This aggregation is performed using 4 different variants and will be described in detail in the coming section.

B. 2-Kernel architecture

Unlike the 3-kernel architecture, the 2-Kernel architecture is a more compact one. Ordered aggregation of an input data set is performed in 2 kernels. The Fig. III shows the chronological execution of various operations of different kernels in 2-Kernel architecture.

TABLE III
2-KERNEL ARCHITECTURE

Kernel	Operations
1	Partial prefix sum and Offset Computation
2	Aggregation

1) Kernel 1 - partial prefix sum and offset computation: The operations in this kernel is same to that of kernel 1 of the three-kernel architecture, where the partial prefix sum and offset value is computed.

2) Kernel 2 - aggregation: In the 2-Kernel architecture ordered aggregation is also performed in four different ways like the 3-Kernel architecture. The execution of the four variants are similar. The difference, however, is that, instead of saving the offset addition value to the prefix sum array, we directly use the values to compute the resultant ordered aggregation array using the four different variants to be described in the next section.

C. 1-Kernel architecture

1-Kernel architecture is also possible where all the operations are performed in one kernel. This can be done using barriers. However, in one kernel architecture the wait time will increase greatly and hence is not advisable.

D. Variants of ordered aggregation

We have implemented ordered aggregation on four variants. These variants are described in the following section.

1) Direct atomics: In this variant, the number of threads spawned are equal to the number of elements in the prefix sum array. Every thread spawned has a particular global ID and it is responsible for performing an atomic add of 1 in the respective position of the resultant ordered aggregation array.

Fig. 6 , gives basic overview of direct atomics. Here as we can see a thread is spawned for each element of prefix sum array and the value fetched by the threads represents the position where atomic add of 1 needs to be performed in the resultant array.

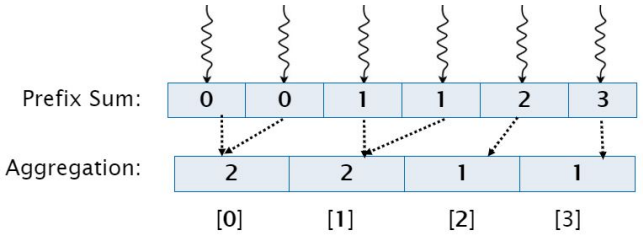

Fig. 6. Direct atomics

The CF No. 3 describes the above process and the atomic operation is performed in the last line of the algorithm.

Algorithm 3 Aggregation using direct atomics

Require: $Prefix\ Sum\ Array\ \&\ Aggregation\ Resulatant\ Array$
Ensure: $PS\_ARR \leftarrow Prefix\ Sum\ Array\ \&$
$AGG\_RES\_ARR \leftarrow Aggregation\ Resulatant\ Array$

$atomic\_add(AGG\_RES\_ARR[PS\_ARR[thread\_id]],\ 1)$

2) Chunked atomics: In this variant, the prefix sum array is divided into equal sized chunks and the number of threads spawned are equal to the number of chunks. A thread with a particular global ID performs an atomic add of 1 in the respective position of resultant ordered aggregation array. This atomic execution is performed sequentially on the elements of the chunks. The resultant ordered aggregation array is achieved after all the threads have completed performing their atomic adds.

Fig. 7 , shows us how aggregation is performed on chunks of data using atomic process. As we can see a thread is spawned for each chunk and the iterates through the chunk in a sequential way thus accessing each element in the chunk of the prefix sum array. Then the thread performs atomic operation of 1 in the resultant array at a location which is determined with the value fetched by the thread.

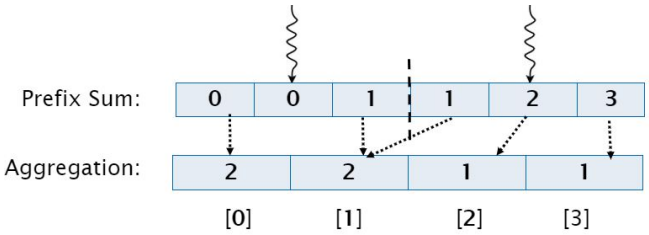

Fig. 7. Chunked atomics

The CF No. 4 describes the above mentioned process where the atomic operation is specifically performed within the **for** loop.

Algorithm 4 Aggregation using chunked atomics

Require: $Prefix\ Sum\ Array$ & $Aggregation\ Resulatant\ Array$
Ensure: $INTENT \leftarrow Intent$,
$PS\_ARR \leftarrow Prefix\ Sum\ Array$ &
$AGG\_RES\_ARR \leftarrow Aggregation\ Resulatant\ Array$

$mIndex \leftarrow (thread\_id * INTENT)$
$mIndexLast \leftarrow (mIndex + INTENT)$

for $i \leftarrow mIndex,\ mIndexLast$ **do**
  $atomic\_add(AGG\_RES\_ARR[PS\_ARR[i]],\ 1)$
end for

3) Atomic using a private variable: In this variant, private variables are used as an intermediary step to perform ordered aggregation. Each equal chunk of the prefix sum array has a respective private variable. A thread executes the elements of a chunk in a sequential manner. The private variable is initialized with 0. The first value of the chunk is compared with itself and the variable gets incremented by 1. The next value is then compared with the previous value. If they are same, then the variable gets incremented by 1 again. If not, an atomic add of the current value in the variable is performed on the resultant ordered aggregation array. The position of the atomic add is given by the previous element in the chunk when there is a mismatch between the compared values. The Fig. 8 is a pictorial representation of this description. Atomics are performed under two conditions. Firstly, when there is a mismatch in the compared elements within a chunk and secondly when a chunk boundary is encountered. After all the threads complete executing their respective chunks, the resultant ordered aggregation array is achieved.

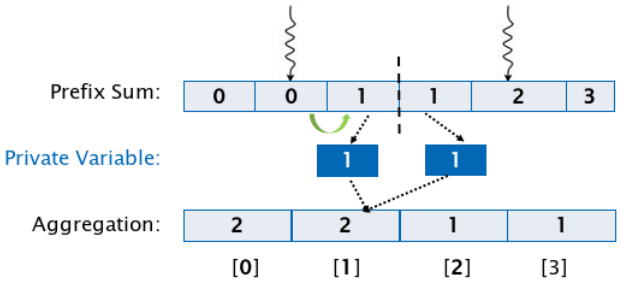

Fig. 8. Aggregation using private variable

The CF No. 5 is algorithm for the above said process and the aggregation is performed once in the **for** loop and once in the end of the algorithm which is the end of execution of each chunk.

Algorithm 5 Aggregation using private variable

Require: $Prefix\ Sum\ Array$ & $Aggregation\ Resulatant\ Array$
Ensure: $INTENT \leftarrow Intent$,
$PS\_ARR \leftarrow Prefix\ Sum\ Array$ &
$AGG\_RES\_ARR \leftarrow Aggregation\ Resulatant\ Array$

$mIndex \leftarrow (thread\_id * INTENT)$
$mIndexLast \leftarrow (mIndex + INTENT)$
$mLocalAgg \leftarrow 1$

for $i \leftarrow mIndex,\ mIndexLast$ **do**
  if $PS\_ARR[i] - PS\_ARR[i+1]$ **then**
    $atomic\_add(AGG\_RES\_ARR[PS\_ARR[i]],\ mLocalAgg)$
    $mLocalAgg \leftarrow 1$
  else
    $mLocalAgg++$
  end if
end for

$atomic\_add(AGG\_RES\_ARR[PS\_ARR[mIndexLast-1]],\ (mLocalAgg - 1))$

4) Atomic using a private array: In this variant, arrays along with private variables are used as an intermediary step to perform ordered aggregation. Each equal chunk of the prefix sum array has two respective arrays and a variable. The first array is position array while the second is a counter array. The variable used is a counter for unique elements within a chunk. Threads execute within a chunk in a sequential manner. When a new element is encountered in a chunk, the variable is incremented by 1, the new element is inserted into the position array at the position indicated by the variable and the counter array gets incremented by 1 in the same position. When the element encountered is the same as the previous, then the position array and the variable remains intact. Only the counter array is incremented by 1 again. After a thread encounters the chunk boundary the resultant ordered aggregation array is populated with values both by using atomics and directly, without using atomics. Atomics are performed only twice every chunk at the extremes. The final value held in the variable gives us the number of unique elements in the chunk. For e.g, as in the Fig. 9 the final value in the second variable is 2, atomic add will be performed on the elements in the 0^{th} and 2^{nd} position while the elements in the 1^{st} position will be added directly. Atomics are performed on the extremes only, to ensure same elements in different chunks are not in contention. This also saves time, as atomics are not performed on all the elements.

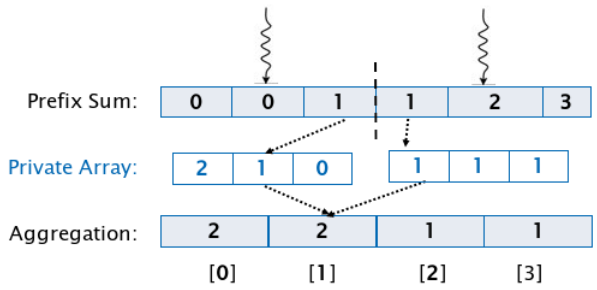

Fig. 9. Aggregation using private array

The CF No. 6 is algorithm of the above said process and the aggregation is performed from the step mentioned above the second **for** loop. To get the final aggregation the atomic operation is done twice and rest values are assigned directly within the for loop.

V. Evaluation

The OpenCL framework was used for the implementation of 2-kernel and 3-kernel architectures. OpenCL is a platform-independent framework and also supports GPU and CPU programming. We evaluated these 5-kernel, 2-kernel, and 3-kernel architecture. Moreover, we also evaluated four ordered aggregation variants, namely - direct atomics, chunked atomics, private variable and private array. Finally, each variant is evaluated for its run time for different data

Algorithm 6 Aggregation using private array

Require: $Prefix\ Sum\ Array\ \&\ Aggregation\ Resulatant\ Array$
Ensure: $INTENT \leftarrow Intent$,
$PS\_ARR \leftarrow Prefix\ Sum\ Array\ \&$
$AGG\_RES\_ARR \leftarrow Aggregation\ Resulatant\ Array$

$mIndex \leftarrow (thread\_id * INTENT)$
$mIndexLast \leftarrow (mIndex + INTENT)$
$l\_pos\_cnt \leftarrow 0$

$l\_ps[l\_pos\_cnt] = PS\_ARR[mIndex]$
$l\_agg\_res[l\_pos\_cnt] = 1$
for $i \leftarrow (mIndex + 1),\ mIndexLast$ **do**
  if $PS\_ARR[i] - PS\_ARR[i-1]$ **then**
    l_pos_cnt++
    $l\_ps[l\_pos\_cnt] = PS\_ARR[i]$
    $l\_agg\_res[l\_pos\_cnt] = 1$
  else
    $l\_agg\_res[l\_pos\_cnt]++$
  end if
end for

$atomic\_add(AGG\_RES\_ARR[l\_ps[0]],\ l\_agg\_res[0])$

for $i \leftarrow 1,\ l\_pos\_cnt$ **do**
  $AGG\_RES\_ARR[l\_ps[i]] \leftarrow l\_agg\_res[i]$
end for

$atomic\_add(AGG\_RES\_ARR[l\_ps[l\_pos\_cnt]],$
$l\_agg\_res[l\_pos\_cnt])$

sets. We have evaluated the implementation of various fused kernels as mentioned in Table II and Table III, on synthetic as well as real-world TPC benchmark-H (TPC-H) data. For synthetic data, we have used random distribution for unique and reduce data distribution. In unique data distribution, every element is unique. In reduce data distribution, the elements in the data are present repeatedly.

We evaluate our proposed method for the given datasets on the GeForce GTX 1050 Ti GPU supported by OpenCL 1.2 Device driver. The hardware has the capability to run 1024 work items within a workgroup and has a memory controller to handle all the atomic operations [10]. The GPU also has 2.94 GiB global memory and 48 KiB local memory space. On top of that, we use GCC 5.4 version for compilation running on Ubuntu 16.04 OS environment.

A. Impact of code fusion

To find the best kernel architecture and local size, we performed an evaluation on 2-kernel, 3-kernel and 5-kernel architectures using unique and reduce data. The winner architecture is the kernel that will perform best in terms of

throughput on both unique and reduce data set.

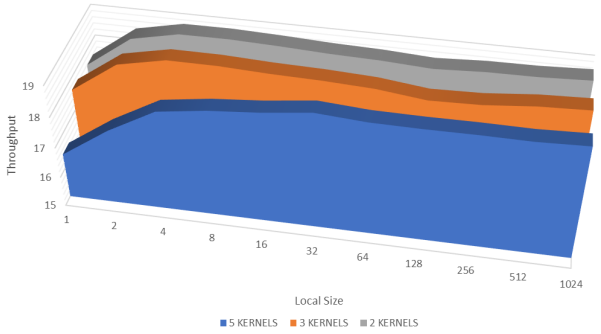

Fig. 10. Performance comparison on reduce data

Unique data is where every element in the data set is different whereas reduce data set contains duplicate elements. This gives us better insights as unique data set will help us find the best kernel when worst case of ordered aggregation. In reduce data set, there are chances of more contention. As shown in Fig. 10 & Fig. 11, 2-kernel and 3-kernel architectures have performed comparatively similar.

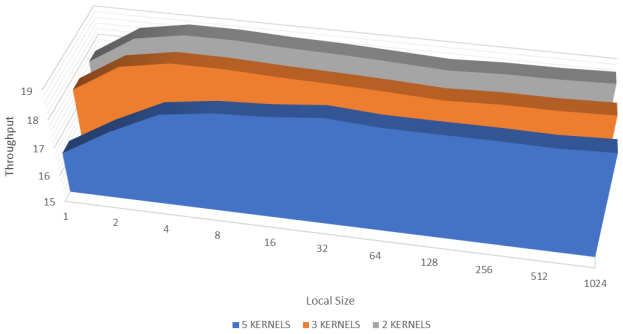

Fig. 11. Performance comparison on unique data

However, we selected 2-kernel architecture as the winner as it has performed marginally better as compared to 3-kernel architecture. The performance is the same for both reduce and unique data set. The local size selected for both reduce and unique is 8, as 2-kernel architecture has the most throughput for both the data sets.

B. Impact of chunking

Furthermore, we evaluated the winner kernel which is 2-kernel architecture to find the optimal chunk(intent) size.

In this, atomics were performed per chunk rather than on individual data. In order to find the optimal chunk size, we have fixed the local size. The local size is the same as the optimal local size derived from the previous evaluation. Fig. 12 shows throughput for varying intent sizes ranging

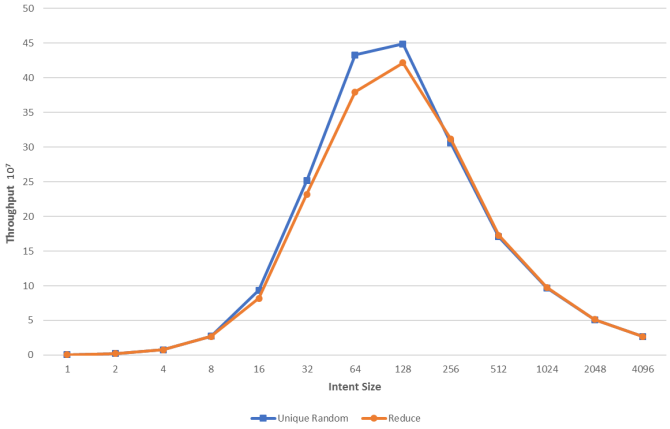

Fig. 12. Throughput of chunked atomics with varying intent size

from 2^0 up to 2^{12} with fixed local size to 2^3. As per the result, intent size 128 yields the highest throughput for both unique and random data. Moreover, the 2-kernel architecture with chunked atomics with 128 intent size shows significant improvement in throughput when compared with direct atomics.

C. Impact of private variable and private array

Next part of the evaluation is performed on synthetic data and real world data using the optimal local size and optimal chunk(intent) size with the two variants, private variable and private array. With the same data size, optimal local size and optimal chunk(intent) size are obtained from our previous evaluation. In Fig. 13 First two columns represent synthetic data where former denotes unique distribution and latter reduce distribution. Starting from third column, these represents columns of table line-item in TPC benchmark-H. As in Fig. 13 throughput of the system shows that private variable performs better than private array except for the case of reduced distribution. For that, private array performs a little bit better than private variable but the performance is comparatively similar for both.

For this project, evaluation is performed based on few parameters like local size and chunk size. More parameters can be involved in the evaluation which will lead to increased number of dimensions. In multi-dimensional problem, we do not get to know which parameters are dependent and how each parameter effects the performance of other parameter. In order to solve this mathematical problem, statistical methods need to apply and are open for such solutions.

VI. Future Work

Due to the system limitations, we have been able to perform evaluation on the data of size 2^{15}. For future work, we suggest to perform same evaluation on the data size

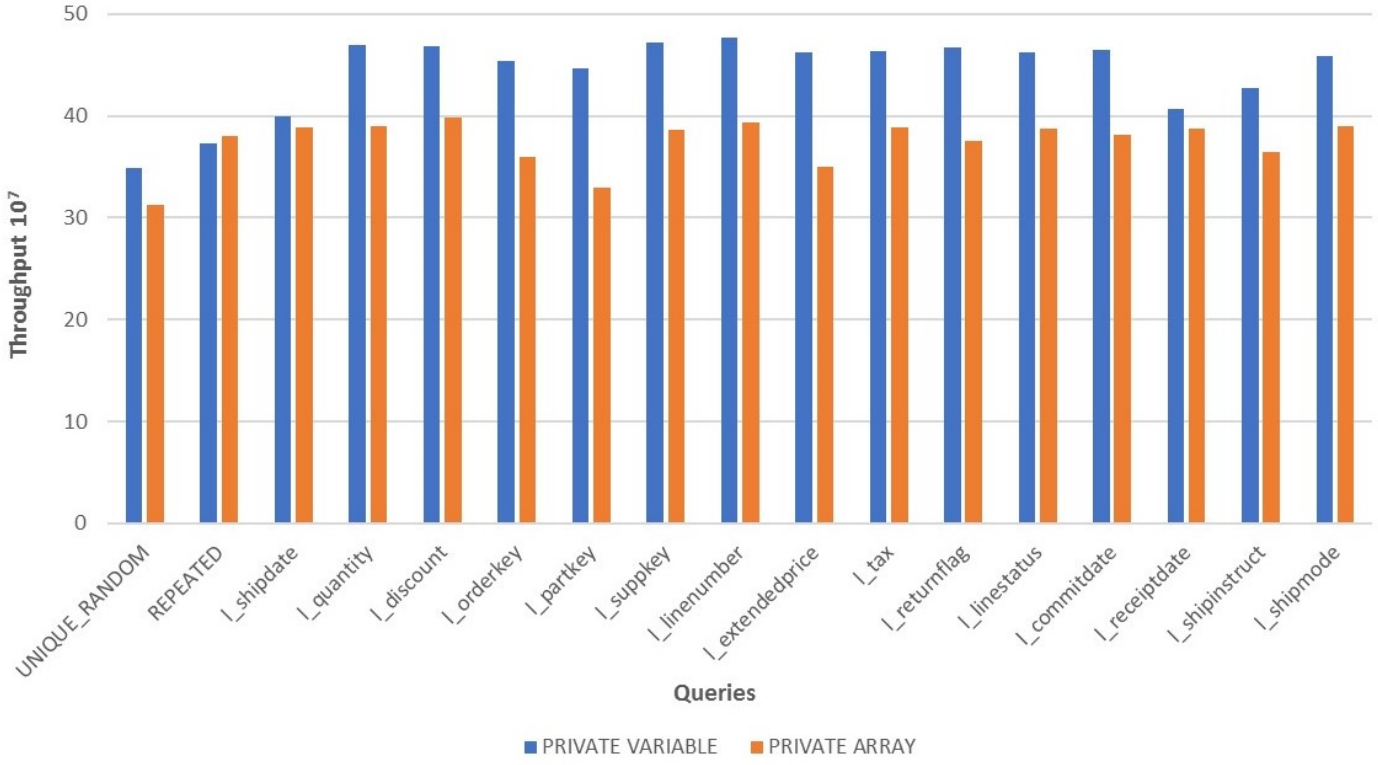

Fig. 13. Evaluation of Synthetic data and TPCH Line-item table with private variable and array

greater than 2^{15}.

VII. Conclusion

In this project, we implemented ordered aggregation using prefix sum on 3-kernels and 2-kernels by fusing the kernel code of 5-kernel architecture. We evaluated the performance of above mentioned three architectures for different data, local sizes, and intent size. Our results indicate that 2-kernel architecture performs best among the three for local size 8 on both reduce and unique data. Eventually, we evaluated 2 kernel architecture to be the best in term of throughput and execution time.

References

[1] P. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8391 LNCS, pp. 61–76, Springer Verlag, 2014.

[2] T. Karnagel, R. Mueller, and G. M. Lohman, "Optimizing GPU-accelerated Group-By and Aggregation," tech. rep., ADMS@VLDB, 2015.

[3] G. E. Blelloch, "Prefix Sums and Their Applications," tech. rep., Synthesis of Parallel Algorithms, 1990.

[4] B. Bingsheng He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *Association for Computing Machinery (ACM) Transactions on Database Systems*, vol. 34, pp. 1–39, 12 2009.

[5] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *In GPGPU 10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 94–103, Association for Computing Machinery (ACM), 3 2010.

[6] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proceedings of the VLDB Endowment*, vol. 4, pp. 539–550, 6 2011.

[7] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," *ACM SIGPLAN Notices*, vol. 48, p. 103, 8 2013.

[8] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the Cost of Atomic Operations on Modern Architectures," in *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 445–456, Institute of Electrical and Electronics Engineers Inc., 2015.

[9] V. Rosenfeld, M. Heimel, C. Viebig, and V. Markl, "The Operator Variant Selection Problem on Heterogeneous Hardware," tech. rep., ADMS@VLDB, 2015.

[10] NVIDIA Corporation, "NVIDIA GP100 Pascal Whitepaper," tech. rep., 2016.