Rapids Group-by GPU Hash Based Implementation

Tonglin Chen, Tianming Cui, Yadu Kiran, Christopher Patterson

May 15, 2019

1 Introduction

Group-by is a very common operation used in computer databases. It's applications range from authenticating forms, to calculating the average sales of a product in a store. In-memory database systems have a relatively well-developed technique for processing complicated queries. However, there is always a demand for faster performance, especially for large databases. Given the fact that GPUs can achieve massive parallelism and faster memory access rates, implementing the group-by operation on the GPU could yield significant speedup. There are many features to consider when implementing the group-by operation on the GPU. This includes exploiting different types of memory on the device such as global memory and shared memory. Memory structures, block parameters, and other miscellaneous items have to be optimized as well. This paper will provide insight on the challenges associated with creating a GPU implementation of Group-by.

For this project, we first developed a group-by algorithm that runs on the CPU. The CPU implementation was shown to be very slow for large input data sets, so we set a goal to make the GPU version 100 times faster than the CPU version. Our original attempt to meet that performance goal last semester failed to reach 100x speedup. In fact, for the largest data sets we tested, we only achieved a 40x speedup. Thus we kept the same goal of 100x speedup for our new hash-based implementation which we will discuss in this paper. Given the fact that the group-by operation is one of the most time-consuming operations in modern database systems, it is worthwhile to explore a well-designed implementation of a GPU based version of the algorithm that optimizes the overall time of grouping and aggregation operations. The development of this program was carried out on a NVIDIA Tesla GPU. CUDA programming techniques, optimization methods, and concepts learned during this course were applied during the development of this GPU group-by algorithm.

2 Design Overview

The basic premise of the group-by function is illustrated in figure 1. The group-by function takes an array of keys, values, and reduction operations as inputs, then groups elements with the same keys, and performs a reduction operation on the groups. Some example reduction operations include the following: min, max, mean, sum, and count. In some ways, the group-by algorithm is sequential. However, there are still ways to improve the performance of the algorithm by exploiting the parallel architecture of the GPU. For example, reduction operations like sum or mean can be parallelized for enhanced performance. In addition, GPU based sorting algorithms can be used to quickly group together identical input keys. The implementation of these parallelized functions are discussed in the following section.

					ops[]	Max	Count
key[0]	1	2	3		value[0]	1	2
key[1]	1	2	3		value[1]	3	4
key[2]	4	5	6		value[2]	5	9
key[3]	9	2	4		value[3]	7	3
key[4]	8	9	1		value[4]	1	1
key[5]	4	5	6		value[5]	8	9

(a) Input data set example.

					ops[]	Max	Count
key[0]	1	2	3		value[0]	3	2
key[1]	4	5	6		value[1]	8	2
key[2]	8	9	1		value[2]	1	1
key[3]	9	2	4		value[3]	7	1

(b) Output data set example.

Figure 1: Example Group-by input and output datasets.

3 Implementation

For this project, we build on the foundations of the GPU group-by implementation from our work last

semester. Inspired by the work done by Karnagel et al [2], we implemented a hash based approach which supplanted our sort-based algorithm. The hash table consists of 3 parts. The first list is an entry storing key index from the original table. It is either a valid index to represent the key of the table, or a -1 to represent an unwritten state. The second list is the number of written data entries to this row. This is required because the mean operation is not commutative. The third list is the result of hash table, stored in column major. First we initialize the hash table to all the key index to -1, min to the highest possible value of type, max to the lowest possible value, and others to 0. Then each thread is responsible for one data entry (in each iteration with grid stride loop). Each thread calculates the position with Hashed key, and tries to insert to the key index array with atomic-CAS operation, then compares the key in the current position with the current key to see if the insertion failed, or if there is already an index stored at position. If the comparison succeeds, the thread updates the count and result column with atomic operations. Otherwise the thread looks for the next position using linear probing.

Each set of keys is hashed to a hash code in this implementation, which is done by the hashcode() function. This function takes all keys of the corresponding entry as the input and generates a corresponding hash code as the output. In general, the hashcode() function should be able to generate well-spread hash codes for the whole input data set. In other words, different keys should generate different hashcodes. It is also vital that the distances between different hash codes should be as far as possible, so that it will be safe to add new entries between them. There are many different hashcode() algorithms developed by previous researchers, and for our implementation, we chose the following function:

$$hashcode(keys) = keys[0] + keys[1] * 31^1 + keys[2] * 31^2 + ... + keys[N] * 31^N$$

This hash function is widely used, as it provides the following advantages:

First, as 31 is a prime number, the resulting hash codes will have less common divisors and make fewer collisions. Our tests indicate that does translate into performance , as unique keys get evenly hashed into the hash table.

Second, multiply by 31 can be implemented with left shift operations, followed by a subtraction, as x times 31 equals to $(x << 5) - x$. This operation is relatively cheaper to compute in binary than multiplication.

To make sure these hash codes can be used as the indexes within a hash table, we need to make sure it is smaller than the size of the hash table:

$$hashindex(keys) = hashcode(keys) \ \% \ length\_of\_hashtable$$

If the hash table is not large enough, it might fill up and prevent the kernel from adding new entries into the table. Even if the table is large enough to hold all entries, hash code collisions will influence the performance of the hash table, and we would still need a larger table if too many collisions occur.

In order to make sure the hash table size is acceptable, we designed an algorithm to predict the size of the hash table that would be needed. The algorithm was observed to be working well during our testing, but we would still need to have some safety checks to make sure the kernel is robust, so a relaunching strategy is used here.

Once the hash table is filled to 75% of its size, there will be significant collisions. So when the kernel detects 75% of the hash table is filled, we do a relaunch. The relaunching can be implemented with two approaches:

The first approach is to dynamically allocate more memory and relaunch the kernel directly within the previous kernel. This approach might perform better as it can reuse the hash table from the previous run by re-hashing it into the new large hash table. But the issue is, it also introduces divergence among threads, and doing re-hash introduces more overhead since it is hard for each thread to know which entries have already been hashed before and which have not.

So we actually used the second approach in our implementation, when is to set a flag in the unified memory. When the kernel decides to relaunch, it sets this flag as 1. On the CPU side, the host abandons old results and re-allocates a larger memory for the hash table if the flag was detected to be 1, then launch the kernel again. This strategy gets a good performance during our test: it works correctly and the cost of relaunching is acceptable. And, even with a extreme condition, where the kernel needs to be relaunch for a large amount of times, this implementation can still work correctly(while relaunching from the kernel has a depth limitation).

To eliminate as many relaunches as possible, we would like to measure how many unique keys are

in the data entries before we start. The problem is termed as cardinality estimation. Three methods could be used to exactly estimate the cardinality: Bitmapping, Sorting, and Hashing [1]. All three methods requires a thorough scan on all the entries. Bitmapping requires a bit string of all the possible keys which is not suitable in our case because the length of key could be varied. Since the time complexity of Sorting is $O(N \log N)$, and hashing is the whole purpose of this project, these three methods are not suitable. Here we use random sampling to estimate the cardinality. The estimation bases on the assumption that the unique keys are uniformly distributed in the data entries. Assume we have N unique keys and they are evenly distributed in the data set. Suppose we already know M unique keys ($M <= N$). Then the probability of selecting a known key from the set is $P(X_n = M) = M/N$, and probability of selecting an unknown key is $P(X_n = M + 1) = (N - M)/N$. Since the number of unique keys M is in discrete finite state space $\{0, 1, ...N\}$, one step of such test is a Markov process with the transition matrix:

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1/N & (N-1)/N & 0 & \dots & 0 & 0 \\ 0 & 0 & 2/N & (N-2)/N & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \\ 0 & 0 & 0 & 0 & \dots & (N-1)/N & 1/N \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix}$$

Utilizing the matrix, we could do random sampling and guess the cardinality N from the result. For CPU sampling, we implement sampling of two keys. The possibility that the two keys are equal is $P_{01}^2 = 1/N$. The sampling stops when there is 10 times that the keys are the same, or the number of experiment reached 1% of the original size of data. The GPU sampling uses three step sampling. The possibility that there are two different keys in three experiments is $P_{02}^3 = 3(N - 1)/N^2$. The reason for using three steps instead of two is that the standard deviation of guessed N is smaller for same amount of independent experiments. Using cuRAND library, 1% of sampling is done and N is derived from the results. The hash table size is 2.6 times the predicted N to ensure in most cases the hash table can fit all the keys.

We have test the performance of the prediction algorithm with several different input sets. With 1M input rows, for each unique keys number(10, 100, 1k, 10k), 20 repeated tests were done, and no relaunch happens, means the prediction size is quite reliable.

To copy back the values from hash table, Thrust was used to identify the indices in the hash table that were used. This way, we can ignore rows which do not contain a key. This is important because the hash table may be sparse if the ratio of unique keys to input rows is quite small. After obtaining an array of indices for each of the used hash table rows, the unique keys are copied to the output key matrix. We do this using a kernel by assigning threads an index from the Thrust operation. These threads will then access their hash table element and use the key index stored to find the correct row for the unique key in the input key matrix. All columns of the key are copied to the index of the threadId in the output matrix.

Similarly, to retrieve the value matrix from this Group-by operation, a kernel is launched where each thread is given an index into the hash table determined by the Thrust kernel. The difference is that all the data for the value matrix resides in the hash table so we do not need to reference any input data to obtain the output value matrix. We simply loop through each column of the hash table, copying the data to the output matrix. If a column's operation is to calculate the mean, at this point we divide the column value by a key count we stored for each unique key.

Once these kernels are complete, the host simply copies the data from device to host and the Group-by operation is done.

4 Verification

With our previous work on Group-by, we implemented and verified the results of the CPU version of our algorithm by exporting the input and output data into excel. Thus, verification was adapted to compare the CPU output (example output fig.2a) with the new GPU output (example output fig.2b). This required minor code modification to do as the previous sort based GPU Group-by algorithm sorted all the keys and output them in a similar way the CPU formulates the output matrices. The new hash based GPU implementation performs the operations and then copies back the data to the host without a guarantee that keys, and their data, will be in a sorted order. To make comparisons between the verified CPU output and the hashed based GPU output, we modified the routine to check that both outputs contain the same number of unique keys and then if that matches, we find CPU output key in the GPU output matrix and compare values.

Verification was done using integer values for various input sizes with 100k rows, 2 key columns, 10 unique keys, and 3 value column. Because we are using integers for all of the data types, a direct comparison is sufficient. Future implementations that use floating point numbers will need some kind of tolerance threshold in the comparison.

```
[patte539@ece-gpulab21 ~/RAPIDS-Groupby]$ ./groupby_hash
Operations:rsum rsum rmin
numGroups: 10
Printing Results...
{2365:12914}:{502561712:502321936:22}
{4810:51393}:{497691804:503088131:51}
{5608:43215}:{497787678:498785222:26}
{14565:23622}:{493472552:492995062:28}
{30393:4710}:{498388215:497913435:6}
{40926:32489}:{502067416:498752098:6}
{45465:26413}:{508616518:501343698:6}
{56357:13533}:{508680477:497572000:7}
{56467:44642}:{496834250:492435478:0}
{59209:43560}:{493476177:495341131:9}
End Printing Results
```

(a) CPU results.

```
Printing GPU Results...
{59209:43560}:{493476177:495341131:9}
{4810:51393}:{497691804:503088131:51}
{30393:4710}:{498388215:497913435:6}
{5608:43215}:{497787678:498785222:26}
{56357:13533}:{508680477:497572000:7}
{40926:32489}:{502067416:498752098:6}
{14565:23622}:{493472552:492995062:28}
{2365:12914}:{502561712:502321936:22}
{45465:26413}:{508616518:501343698:6}
{56467:44642}:{496834250:492435478:0}
End GPU Printing Results
CPU time: 0.0158778 s
GPU time: 0.0140272 s
PASSED - CPU data == GPU data
```

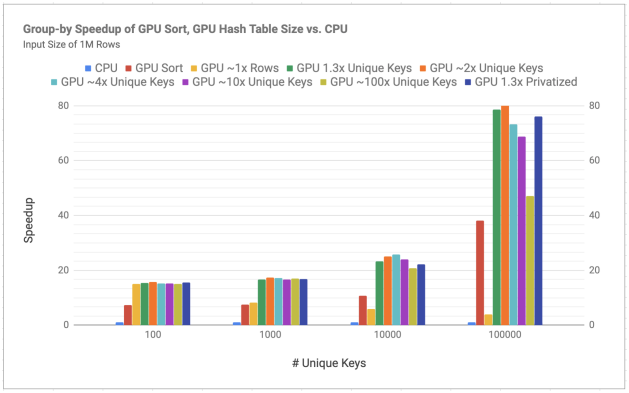
(b) GPU results.

Figure 2: GPU and CPU results compared. 10 unique keys, 2 key columns, 3 value columns, ops:{sum, sum, min}.

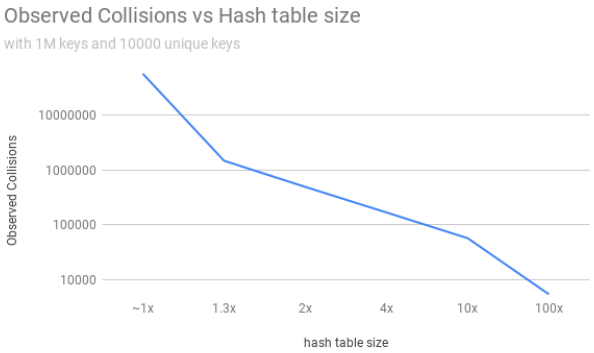
5 Performance

With our hash-based Group-by implementation, we can see that that the GPU implementation can achieves up to a 78X speedup over the CPU implementation for the input sizes we tested. With respect to our previous, Thrust sort-based implementation we saw improvements upwards of 2x speedup. This performance was closer than the sort-based design but still need not meet our performance goals and this is due to the high overheads our kernels have for copying data to the GPU. Figure 3 below shows a bar graph of the speedup of the GPU implementation for an input size of one million rows. This graph shows the speedup between all our tests and the serial CPU version (blue) as well as the original sort based implementation (red).

We can see that the best performance gains are typically achieved by the runs where our hash table is set to 1.3x the number of unique keys matrix (green) and 2x the number of unique keys matrix (orange). Continually increasing the hash table size begins to see performance degradation with respect to 1.3x and 2x the size of the hash table. This is likely due to the overhead of allocating that much memory on device. Also when the length of hash table is too large, memory coalescing is significantly reduced. On the other hand, a hash table of 1x the size of the input matrix achieves worse performance than the sort implementation when there are many unique keys. This makes sense as when the number of elements stored in a hash table approaches the size of the hash table, the number of collisions increases dramatically, which causes lots of divergence and comparison overheads in our kernel code. Thus, careful consideration of hash table size is important. By using our implementation which samples and predicts the size of the hash table, the predicted table size, on average, get a hash table large enough to ensure minimal collisions while not incurring unnecessary cudaMalloc overhead.



(a) CPU vs sort based vs various GPU results.

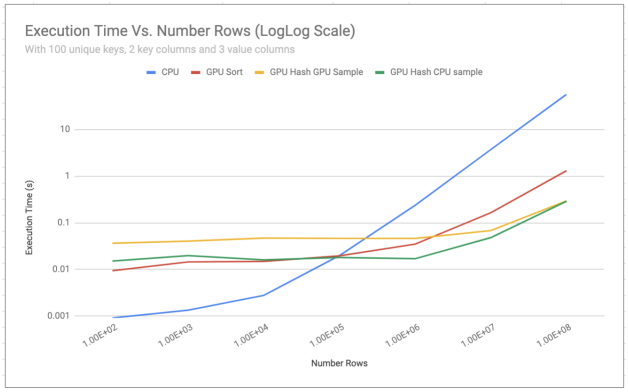


(b) Number of collision observed

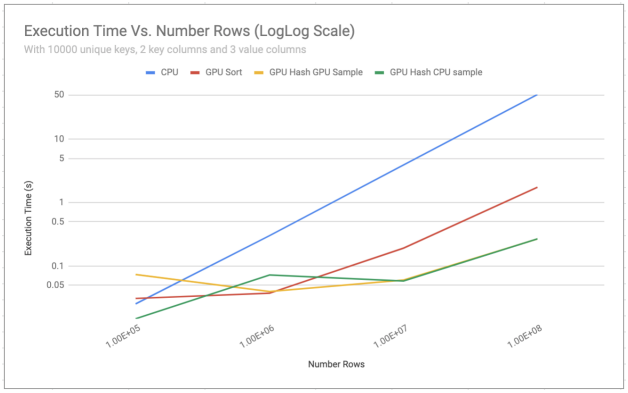
Figure 3: Comparison of execution time and number of collisions.

The graphs in figure 4 show a log-log plot of the execution time for each implementation for different input matrix sizes. As we can see in the graphs, there is a bit of overhead in launching the GPU kernels. This makes the CPU implementation faster for data sets with 100k rows or less. For larger data sets, the GPU implementations are much faster than the CPU implementation. It is also evident that when the number of unique keys are small, there is more overhead to do key sampling on the GPU than on

the CPU. Without prior knowledge of how the input data should look, determining to run the CPU sampling over GPU sampling is not trivial and would require some testing to determine which method to use. However, if the input data set is large, we can see the GPU and CPU sampling perform similar, thus consideration of which method to use may not be necessary.



(a) 100 unique keys, varying input matrix size.



(b) 10k unique keys, varying input matrix size.

Figure 4: Execution time of CPU vs. sort-based vs. hashed-based table size sampling, using Markov chain.

To get a better understanding of the performance bottlenecks in our implementation, we profiled our program. Using "nvprof" we could see the run-time for different kernel functions and CUDA API calls. The data below is from a run with 100 million rows and 10k distinct keys, 2 key columns, and 3 value columns using the hash-based algorithm with GPU sampling to predict the hash table size. In this run, GPU executiontime is 0.268734s and there were no relaunches.

The results of the profiling are shown below in figure 5. In this depiction, we can see 74.1% of the GPU time (161.95ms), was spent copying data from host to device. This is the biggest limitation our algorithm has as the data sets we work with are quite large. The second most intensive GPU activity is running the fillTable kernel. This takes up 19.02% (41.565ms) of the time because this is where the heavy lifting of our algorithm comes into play. In this kernel, we are performing our hashing algorithm to determine where to store keys, performing atomic operations in the hash table, and performing new calculations when there are hash collisions. The only other notable activity is the predictTableLength_GPU kernel which takes up 6.51% of the GPU run-time. This is due to the Markov model explained above in section 3 and is necessary for ensuring launch with a minimal but sufficient sized hash table for any input matrix sizes. This way we abstract away the implementation from the user so they do not have to worry about supplying a hash table size. Of course there are ways to predict the hash table size in other ways like histogramming the entire input space but this would incur larger overheads to launch the GPU. Most of the other GPU activities relate to copying back the Group-by results to the host and affect the run-time minimally.

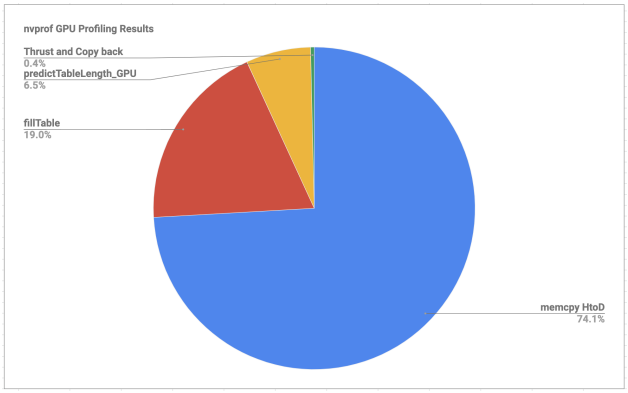


Figure 5: nvprof profiling run-time results for hash-based Group-by with 100M rows and 10k unique keys.

6 Future Work

In addition to the hash-based approach, we also tried to incorporate other techniques that we covered in the course to try and reap additional performance. Heterogeneous CPU-GPU computing seemed like a potential candidate, as the CPU was sitting idle while the GPU carried out all the operations. Task partitioning wasn't an option, as all the tasks involved in the operation need to be sequential in its execution flow. For data partitioning, the operations phase is the only task that would facilitate heterogeneous

computing since the operation can be preformed on a row of the value matrix, independent of the other rows.

Unfortunately, the task proved to be more difficult than we initially thought. The GPU thrust kernels are synchronous in nature, so in order to split the work between the CPU and GPU, we would have to spawn a new CPU thread that performs the computation. However, thrust does not offer too much flexibility and control in this regard. If a thread other than the original thread attempts to call any thrust function in the operations phase will cause the program to crash due to lack of memory access privileges. We would have to rewrite large portions of the code to facilitate heterogeneous computing.

It is also worth mentioning that the current data structure is stored in memory in column major format. However, due to the access patterns of many of the tasks involved in the function (including the actual operations themselves), the accesses to the matrices and hash table in memory are uncoalesced. The performance, as demonstrated in section 5, achieves significant speedup over prior implementations, but further work on the Rapids Group-by project will explore converting to row major and seeing if there is further room for improvement.

7 Conclusion

For this project, our group designed and implemented a improved group-by kernel based on our works last semester with a new hash-based approach. The performance improvement is substantial when the input matrix has unique keys more than 10% of its size. The new implementation obtains a 2x speed-up over our old sort-based GPU implementation, and up to 78x speed-up compared with the CPU implementation(which utilizes highly optimized STL functions). Our hash-based group-by kernel is also quite robust and accurate while highly efficient. It can generate correct results for all our test cases, ranging from 1,000 rows to 100,000,000 rows.

In order to improve performance, we tried various different approaches that incorporate the techniques and algorithms we tackled in lectures. We tested and modified the hash code function, developed a privatized version with sub hash tables in each warp to reduce the global accessing, tried data partitioning to take full advantage of the CPU resources, etc. We also made some effort to make sure our implementation is robust by designing a prediction algorithm that helps to allocate enough memory for the hash table. Even if the predicted size is incorrect, the kernel still has a safety consideration that will relaunch the whole kernel with a larger hash size once it runs out of space.

As we predicted last semester, a hash-based group-by kernel can provide better performance than the sort-based approach. We predict that there is still some room for improvement which could be tackled in the future. Possible optimizations include rewriting thrust functions to reduce overhead and provide more control and flexibility, or trying other memory alignment strategies which may lead to better performance.

References

[1] Hazar Harmouch and Felix Naumann. "Cardinality estimation: An experimental survey". In: *Proceedings of the VLDB Endowment*. Vol. 11. 4. VLDB Endowment, 2017, pp. 499–512.

[2] Tomas Karnagel, René Müller, and Guy M. Lohman. "Optimizing GPU-accelerated Group-By and Aggregation". In: *ADMS@VLDB*. 2015.