Aaron Nightingale
Christopher Patterson
Jersin Nguetio
Menglu Liang
Tonglin Chen

Rapids Group-by GPU Implementation

I. Introduction

Group-by is a very common operation used in computer databases. It has applications ranging from authenticating forms to calculating the average sales of a product in a store. The performance of in-memory database systems has a relatively well-developed technique for processing complicated queries. However, an ever-faster performance is still required by users, especially in 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 result in major increases in performance.

There are many features to consider when implementing the group-by operation in 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 had to be optimized as well. This paper will provide insight on the challenges associated with creating a GPU group-by application.

In 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. By increasing the performance of the algorithm by a factor of 100, there would be many positive effects in real-world applications. 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 1080 GPU. CUDA programming techniques, optimization methods, and concepts learned during this course were applied during the development of this GPU group-by algorithm.

II. Design Overview

The basic idea of the group-by function is shown in the figure below. The group-by function takes an array of keys, values, and reduction operations as inputs. The group-by function 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.

Input Data Set Example

Key[0]	Key[1]	Key[2]	Key[3]	Key[4]	Key[5]	Key[6]
0	1	1	4	9	8	4
1	2	2	5	2	9	5
2	3	3	6	4	1	6
Value[0]	**Value[1]**	**Value[2]**	**Value[3]**	**Value[4]**	**Value[5]**	**Value[6]**
0	1	3	5	7	1	8
1	2	4	9	3	1	9

Output Data Set Example

Key[0]	Key[1]	Key[2]	Key[3]	Key[4]
0	1	4	9	8
1	2	5	2	9
2	3	6	4	1
Value[0]	**Value[1]**	**Value[2]**	**Value[3]**	**Value[4]**
0	3	5	7	8
1	2	1	1	2

Figure 1 - Example group-by input and output data sets.

III. Implementation

Many of the parallel algorithms were implemented using the Thrust library included with CUDA. This library provides an API to create device pointers and vectors similar to the functionality provided through STL. Thrust is highly optimized and provides many common parallel algorithms such as reduce, scan, and copies from host to device memory. Another benefit of Thrust is that many of these APIs allow customizable functors to map to atypical data structures. The drawbacks of using Thrust in our implementation are that we do not have control over many of the advanced CUDA features as we would if we had implemented our own kernels. This includes when we can use shared or constant memory and using streams. These limitations can be mitigated by pairing Thrust with custom kernels.

Our group-by implementation takes several inputs, including a key matrix of size MxN, a value matrix of size MxK, and a reduction operation array of size K. Each row in the key matrix corresponds to a key and maps directly to a row in the value matrix. Furthermore, each element in the reduction array corresponds to a column in the value matrix. Our algorithm reduces the key and value matrices to P rows where P is the number of unique input keys. The value matrix reduction is done using the operation specified for a given column. Early versions of the algorithm used Thrust's copy API to transfer data from host to device. Now our group-by data transfers use pinned memory to transfer data. Before using pinned memory for host to device transfers, we saw a data overhead of up to 1.3 seconds, regardless of input size. With pinned memory, the overhead has been decreased to about 0.007 seconds.

We tried two different methods for implementing the group-by functionality. The first method, shown in figure 2, begins by sorting the key matrix to group all of the matching keys together. This is done using Thrust-sort with a custom functor which makes comparisons between columns in each row. The sort also keeps track of the original key row indices while sorting so we can copy data from the value array in sorted order. After sorting, a kernel, identify_bound() is called. This kernel compares two neighboring keys. If they do not match, a one is placed in a new array at the index of the second key being compared. Once the kernel is finished, our new array contains ones at every index where a new key was found. Performing an inclusive scan on this new array gives us a new, single value representing each key. with this new sorted representation and the original row indices, we use the

Thrust function “unique_by_key()” to create a new array of indices corresponding to the unique keys in our input key matrix. This allows us to make a permuted iterator on the original key matrix and copy only unique keys to the key output matrix in the host’s memory.

Figure 2 - A visual representation of obtaining unique keys from the input key matrix.

After obtaining the unique keys and the array of sorted indices, the next step of our algorithm is to perform reductions on each column of the input value matrix based on the corresponding reduction operation in the reduction_ops input array. There are five different reduction types: max, min, sum, count, and mean. Similar to the permuted copy of unique keys to the host mentioned above, we copy a sorted column from the host input value matrix to the device. Once we have that data, we check the corresponding reduction operation. Based on the signified operation, we perform a reduction using Thrust’s reduce_by_key() method with the sorted keys and sorted column data as inputs. This method directly results in a device vector of output values for each unique key for that column. This reduction is performed for each column in the input value matrix, where the operation performed may change between columns. The results are copied back to the host into the output value matrix. This method is depicted in figure 3:

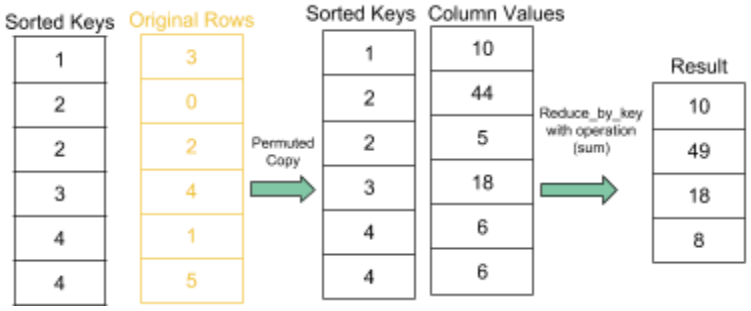

Figure 3 - Example of the value reduction (sum) operation on one value matrix column.

Inspired by similar work done by Karnagel et al. [1], we attempted a hash-based implementation of a group-by operator to further improve performance gains. In essence, a hash-based implementation relies on a hash table residing in device memory and composed of *hash buckets* with fields designed to serve as *aggregate* fields. Mapping each input row to a hash bucket by means of its hash value eliminates multiple traversals of input data that is inevitable in sorting-based algorithms. After each input row has been added to its corresponding bucket, aggregate data from hash buckets are extracted

and compiled into a format suitable for delivery. Despite much effort, we were not successful in generating valid results. We have nevertheless included source code and header files for review.

IV.Verification

In the early development stages of this project, we verified our CPU group-by implementation manually. The two pictures below show a typical input data set (on the left) and the corresponding group-by output (on the right). By doing some spot checks, we became fairly confident in our CPU implementation. Later we also exported the input and output data sets to a .CSV file so that we can check our implementation in excel. This process worked well for us.

Once the CPU implementation was verified to be robust and correct, we implemented a function that compares the GPU result to the CPU result. 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 threshold in the comparison.

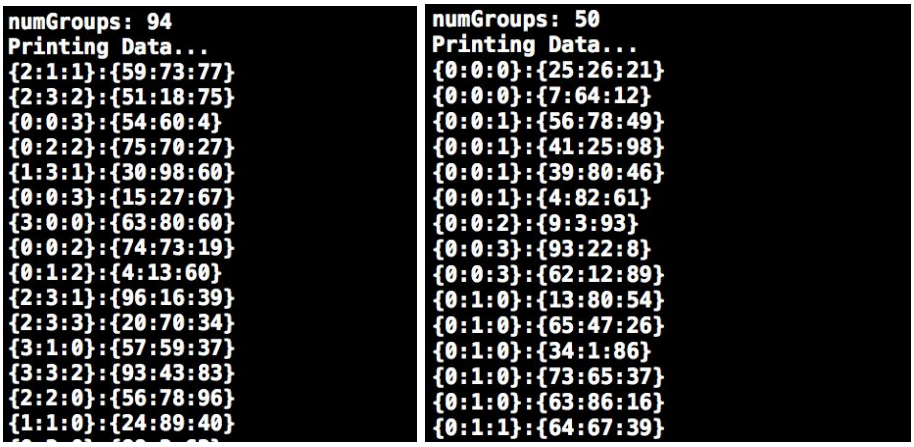

Figure 4 - Example group-by input and output data sets for manual verification.

V .Performance

The group-by operation and many of its reduction operations are inherently serial. For this reason, we were originally expecting a speedup on the order of 100x. In our implementation, we saw that the GPU implementation was up to 22 times faster than the CPU implementation. Although our implementation wasn't as fast as we originally expected, we are pleased with the results. We learned how easy it is to implement custom (yet parallelized) sorting and reduction functions using Thrust. In addition, we learned more about optimizing real-world CUDA applications.

The figure below shows two graphs that summarize the performance of our group-by GPU kernel. The bar graph on the left shows the speedup of the GPU implementation for input data sets with various rows and key columns. As you can see, the GPU implementation gets its best performance for data sets with one million rows or more. The picture on the right features a log-log plot of the execution time of each implementation for a given number of rows in the input data set. As you can see, there is a bit of overhead in the GPU kernel. This makes the CPU implementation faser for data sets with ~80,000

rows or less. For larger data sets, the GPU implementation is much faster than the CPU implementation. One thing to note in the plot is that the slope of the CPU execution time is roughly equal to the slope of the GPU execution time for large input data sets. This indicates that we are not exploiting the full potential of the GPU's parallel architecture. Further work could be done to optimize out GPU group-by implementation.

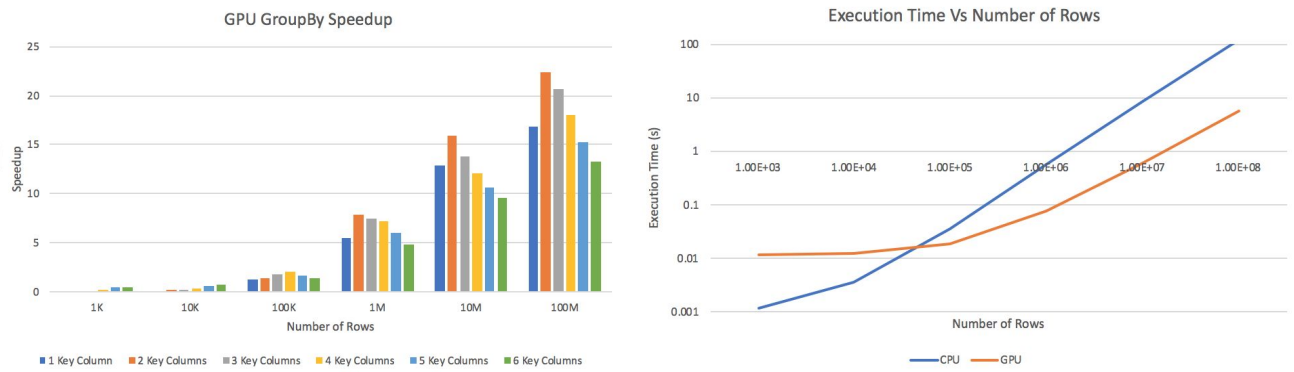

Figure 5 - GPU kernel performance results

To get a better understanding of the performance bottlenecks in our implementation, we profiled our program. Using "nvprof" we could see the runtime for different kernel functions and CUDA API calls. Notice that there might be overlap between kernel function and API calls since the Thrust library would utilize cuda streams to accelerate the performance. The data below is from a run with 10 million random elements and 16 distinct keys using the sorting based algorithm. In this run, GPU execution time is 0.36928s. (Allocating Pinned memory is not counted toward the GPU execution time).

Kernel Profile Result		
Percentage	**Time (ms)**	**Use**
74.22%	212.08	[CUDA memcpy HtoD]
19.12%	54.632	thrust sort
2.96%	8.4713	thrust permutation
0.94%	2.6887	[CUDA memcpy DtoD]
2.76%	~7.9	All other operations

API Profile Result		
Percentage	**Time**	**Use**
55.90%	477.46	cudaMallocHost
18.58%	158.66	cudaStreamSynchronize
8.86%	75.641	cudaFree
7.24%	61.858	cudaMemcpyAsync
6.88%	58.765	cudaFreeHost
1.48%	12.611	cudaMalloc
1.06%	~9.0	All other API calls

Figure 6 - GPU group-by profiling results.

The profiling result shows that the two limiting factors for the sorting based algorithm are memory transfers and the sorting operation itself. Memory operations are expensive in the nature of sorting based algorithm. To have all elements sorted on GPU, we need to copy all the elements to the global memory at once, and the program won't benefit from pipelined memory copy. Also, we would need to allocate pinned host memory that is used only once. cudaMallocHost is more expensive

compared to malloc, so it would generally be better to reuse the memory. Sorting is another bottleneck in this implementation because compare sort (merge sort) is O(nlogn). Using GPU sort helps reduce the time for a smaller number of items but for larger items the time complexity would be the same. Given more development time, we could fully use nvprof to trace all the calls and eliminate unwanted data transfers, thus enhancing the performance.

Early on in this project's timeline we actively pursued a hash-table approach. This is a well documented strategy that gives great performance results. However, we quickly learned that there were many challenges in this type of design. Selecting a hash function, dynamically adjusting device array sizes, handling hash collisions, etc. Given more time, we are confident that we could create a hash-table implementation much faster than our current program.

VI. Conclusion

Our group successfully implemented a GPU group-by kernel capable of handling an arbitrary number of input rows and columns. Our implementation was up to 22 times faster than the CPU implementation (which was using highly optimized STL functions). In addition, our implementation can process tables with 200 million data elements in about 1 second. Our implementation has been thoroughly tested for data sets ranging from 1,000 rows to 100,000,000 rows, and has proven itself to be robust and accurate.

In the early stages of this project, we read several papers about other GPU group-by implementations. By reading these papers we learned more about hash-table group-by implementations as well as the performance bottlenecks associated with the group-by procedure. We also gained valuable experience using Thrust, which is a commonly used library that provides a standard template for commonly used GPU operations. Furthermore, this project has helped us to learn more about the challenges involved with parallelizing real-world applications.

As a final note we want to mention that our GPU group-by implementation is not fully optimized. With more time and resources, there are plenty of changes that could be made to improve the performance of our code. For example, a future implementation could pursue the previously mentioned hash-table approach. This method is appears frequently in literature and has been shown to be quite fast. Other performance improvements may come from re-writing some of the Thrust functions to be optimized for our application. Each of us has learned a lot while working on this project, and considering the schedule of this project, we are pleased with the performance of our GPU group-by kernel.

VII. Reference

[1] T. Karnagel, R. Müller, and G. M. Lohman. Optimizing GPU-accelerated Group-By and aggregation. In *ADMS*'15.