Hash-Join Implementation on GPU

Chitrabhanu Gupta, Krithiga Murugavel

Abstract:

In this project, we are implementing a left join of two input tables of different sizes using the hash-join algorithm on a GPU. We used the cuckoo hashing concepts to avoid collision in our hash table. Finally, we did some performance analysis by running our hash-join in different machines, with different table size to see how it scales horizontally.

Motivation

The database join is an operation that is performed at almost every organization nowadays and given how much data they have at their disposal in recent times, it can actually become a bottleneck in the process flow. We noticed that the join operation is a good example of a highly parallelizable operation and began to investigate different algorithms for implementing a parallel join algorithm in the GPU. We drew inspiration from existing work on joins on the GPU and chose a hash-join algorithm over merge sort and one of our primary reasons for doing so lies in the fact that we intend this join algorithm to be a Proof of Concept for a GPU computing API for Apache Spark, and a simple merge join would probably require some additional reduce operations and extra data flows across nodes in the cluster, thus becoming considerably slow. We also considered the fact that building the hash table would result in O(1) complexity accesses during the actual join, which is highly desirable. We anticipated that the tradeoff would lie in the amount of memory space we require, since we are storing an extra table (the hash table).

Implementation

In order to implement the hash-join, there are two main steps involved:

- Building a Hash Table
- Perform the join between two input tables using the hash table.

we built two separate kernels for the two steps of the process.

Building a hash table.

We decided to hash the smaller table, to keep the memory used up by the hash table to minimum. To build the hash table we used the Cuckoo hash algorithm. Cuckoo hashing uses multiple hash functions to give keys a fixed restricted set of insertion locations. [john's paper]. We selected two hash functions for our cuckoo hashing algorithm.

We created a 64-bit entry from the key and value of the table, by left shifting the key by 32 and adding the value to it. [John's paper], the benefit of doing this is we can make a one-dimensional hash table instead of having to store both the key and value separately.

Each thread in the GPU is assigned to a row in the table. The key element of the row is hashed using the first hash function and the entire 64-bit entry is stored in the hash table index found from the hash function, if there was any other entry previously present in that index of the hash table, the entry is evicted and the second hash function is used to find a new location for the evicted entry, this process is repeated for a min number of iteration (in our case it was 10) unless we decide the hash function we chose was bad and hence we need to build the whole hash table again using some re-hash mechanism.

Hash functions

We used a simple hash function [john's paper] which uses two fixed constants 'a' and 'b' and modulo it with a big prime number p = '334214459' to create large set of unique locations and then finally modulo it with the table size "*TABLESIZE"* to make sure the location is a valid index to the hash table,

(((ai*key+bi)%p)%TABLESIZE);

In-order to re-hash we just have to pass new set of a and b constants to the hash functions.

Performing the Join

In a new kernel, we launched a thread for every row in the bigger table which is to be joined with the smaller table that was hashed in the previous step. Each thread has the task of applying the hash functions on the key of the row it is operating on and retrieve the index to go to in the hash table in-order to fetch the key and value of the first table. If a corresponding entry is found at the index, the 64-bit entry is split to extract the 32-bit key and value. These, and the value of the row of the probe table are put together into a 3-element array, forming a single row of the output table. Each thread stores the output row they produce at an index depending on its unique Tid (ThreadIdx.x + blockIdx.x*blockdim.x) in a static output table, which is our final joined output table.

We feel that this is a good strategy for a join operation since irrespective of the table size, every thread will just pick a row and perform an operation that does not depend on any other threads value or operation, so it can scale horizontally incredibly well, thus fitting very nicely into the Spark programming model, where we might have a cluster of GPUs.

Potential Problems and Possible Solutions/Improvements

We have kept the size of the hash table big enough to avoid collisions and the cuckoo hashing scheme further reduce the chances of a total hash miss. However, if we were on a space crunch and wanted to reduce the address space or memory used for the hash table, we would run into higher chances of collisions and misses. Potential ways of addressing this would be to add further hash functions and implement an efficient re-hash algorithm to efficiently calculate new hash functions.

If the table size to be joined is too big, coalesced reads could improve performance, but for that, when we are reading the table (for this to be spark compliant, when we are converting the spark data frame to "numpy" arrays and then to our array structure) we have to store the data in a structure that would be favorable for coalesced reads.

Our join implementation works on integer join columns only. To make it more generalized, we would have to devise hash functions for other data types, and potentially forsake the idea of punching the key-value pairs into a 64-bit entry in the hash table.

We could assign smaller block sizes so that //why we should assign smaller blocks from the paper// //lets skip this because its in the paper.

Since we are performing a left join (Test-Table Left Join Hashing-Table), we know the output size, and can assign a static array to store our results. But if it was an inner join, where we would not know the number of rows in the output, we would have to use a counter that is incremented atomically and the mutex feature for insulation during the writes of each thread, thus effectively serializing the write to the output table, and making the operation slower as a result.

Since this algorithm is meant to be Spark compliant, a natural problem is the sharing of the hash table between the different GPUs in the cluster, during the join phase. A good approach for sharing the hash table could be implementing a bucketing system based on the keys. The hash table will be bucketed and distributed according to the buckets, and during the join phase as well, the table would be bucketed and distributed so that inter GPU data transfer is minimized, thus speeding up the operation.

Comparison with CPU Implementation and other Tests

In this section, we will present the performance trends of our algorithm (the hashing kernel and the join kernel separately) in response to variations in parameters such as table size and block size. We will also present comparisons against CPU implementation of the same algorithm (we have also developed the CPU implementation ourselves) and discuss the results.