**Hash Join Implementation on GPU**

**Chitrabhanu Gupta, Krithiga Murugavel**

**Background**

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

The first kernel is for building the hash table. // Write details of hashing//

The second kernel is for performing the actual join operation with the help of the hash table. We launch a thread for every row in the table. Each thread has the task of applying the hash functions on the key of the row it is operating on and retrieve the location to go to in order to fetch the key and value of the first table. If a corresponding entry is found at the location, 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 the thread id in a static array, which is our 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 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 even provided a cuckoo hashing scheme to further reduce the chances of a total hash miss. However, if we were on a space crunch and wanted to reduce the address space 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 into our cuckoo scheme and arrange for a rehashing mechanism where a random seed regenerates a new prime number for our hash function.

If a table size 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 dataframe 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//

Since we are performing a left join (TestTable Left Join HashingTable), 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 has 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.