Evaluating String Equi-Join on GPUs

Shivraj Jamgadepatil
21600
Computer Science and Automation

Hrushikesh Salunke
21407
Computer Science and Automation

I. Abstract

The work described here primarily discusses and implements an algorithm to compute equi join of two tables with joining attributes being strings. It evaluates different join algorithms and demonstrates that GPU joins have capability of outperforming state of the art CPU joins.

Results obtained are validated against results obtained using postgreSQL server. Multiple optimizations like multi-threading, zero copy memory, minimal thread locking are used to speedup GPU algorithms.

Keywords: GPU, Databases, Equi-join

II. Introduction

Conventionally database manufacturers have targeted CPUs as their choice of compute hardware to perform database operations. Decades of work has made these operations very optimized on multi-threaded, multi-socket CPUs. But the advent of massively parallel processors like GPUs provides an opportunity to futher improve performance of these operations. One operation that is particularly prevalent in database usage is Equi-join. [1] [2] demonstrate the improved performance obtained using GPUs on Equi Join with integer attributes.

These works motivates us to look into optimizations that can be achieved for Equi join of string attributes. The problem with performing string equi joins is: string sizes can vary, making it difficult to pass them to GPU memory for performing computations. This problem is addressed here by hashing strings before passing them to GPU memory. The hashing allows us to obtain a fixed size integer values for the strings.

The hashing introduces problem of hash collision making it difficult to obtain exact join outputs. The hash function needs to be robust so as number of collisions is minimized. Still no hash function can guarantee zero collisions so while materializing results care must be taken to verify the join outputs obtained are actually valid.

This work demonstrates a GPU based string equi join algorithm which performs better than postgreSQL for computing joins. Strings attributes are hashed using city-hash function to get a packed array of {id,string_hash}. GPU memory resident hash table is used to perform hash join on the tables. The results are brought back to CPU memory to materialize them in main memory.

III. Design Decisions

This section provides a general reasoning about design choices made while finalizing the approach mentioned in next section. This section is divided into two sections which addresses scenarios when join output size is less than available GPU memory and when join output size is greater than available GPU memory but less than total memory available on the system.

A. Join output fits in available GPU memory

1) Choice of Join algorithm: Three different join algorithms were considered to perform in GPU join of given inputs those are:

- Naive Nested Loop join
- Block Nested Loop join
- Hash Join

When evaluating their run times on small tables (100,000 tuples) all algorithms performed almost equally on small number of tuples and all performed worst than postgreSQL. Using `nvprof` it was observed that Hash join computed join faster than any other approach but the memory overhead of data transfer was actual bottle neck in the program. Block nested kernel execution time was particularly bad owing to it's sub-optimal use of parallelism. So only Naive Nested loop join and Hash Join was considered to finalize the join algorithm.

Evaluating both the algorithms on a table with 1,000,000 tuple (fig 1) made it apparent that hash join performed way better (almost 40x faster) than the naive algorithm. Again the major bottleneck in hash join was memory transfer, so it made it apparent that hash join can be scaled without worsening the join performance.

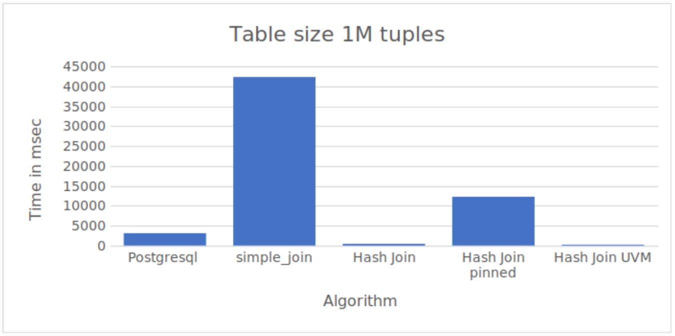

Fig. 1: Performance of Join algorithms over 1M tuples

Also the read only nature of hash join algorithm allows concurrent access to the hash table for performing hash look ups to GPU threads, allowing higher level of parallelism without introducing the need for any inter-thread synchronization. So Hash join is chosen as the algorithm to join two tables in GPU.

2) Linear Probing instead of chaining: While creating a hash table to resolve collision linear probing is used instead of quadratic probing or chaining. The reason is linear probing allows very efficient lookup as shared memory utilization is higher in this case and the access stride is sequential instead of being quadratic or random as in quadratic probing, chaining. Also many of the collisions are avoided by over-allocating a little more memory to store the hash table. This resulted in lower number of collisions and shorter average probing length.

3) Calculating Join Size Upper bound before actual join computation: Before actual join output is obtained a different kernel computes an upper bound on Join output size. This uses same computation as a kernel that performs actual join with removing the false positives present cause of hash collisions. Reasoning behind performing this seemingly unnecessary computation that might introduce inefficiency is:

- The kernel does not come in critical path of the program: It uses same data that kernel computing join needs so no unnecessary large memory transfers and the actual kernel running time observed is negligible as is observed using `nvprof`
- It allows frugal memory allocation: It avoids the need of allocating unnecessarily large GPU memory to store join output which might be the case when using some estimation models. This results in lower unused memory. In the worst case unused memory is 0.004%.
- Allows identification of method needed to store join output: This allows us to determine if join output size might be bigger than the available GPU memory which in-turn makes us easy to choose which type of memory allocation needs to be done to store the join output as is explained further in next subsection

4) Calculating string hash on CPU: Before the table is passed to GPU memory hash of strings present is calculated. Though this is very much a parallel workload which can leverage GPU power it is done on CPU in our case.

Strings are variable sized so passing it to GPU memory needs to over-allocation of memory to achieve constant stride. Even if constant stride is not desired indexing a string becomes very complicated cause of the variable size.

Storing strings also requires a lot more storage than storing the hash. So the data transfer costs become non-negligible in case of large tables.

Instead we calculate hash on CPU. We calculate it as the data is being read. This offsets the cost of hash calculation. Also multiple threads are used to obtain the hash which further reduced the hash computation overhead.

B. Join output size greater than available GPU memory size

Here we assume that though join may not fit in GPU memory, it fits in combined memory available on the system. In the case of output size exceeding GPU memory NVIDIA provides different mechanisms for memory over-subscription:

- Pinned Memory
- Unified Virtual Memory (UVM) [3]

Both of them provide zero-copy GPU to CPU memory. We evaluated both of them for the case when join output size exceeded available GPU memory which makes normal `cudaMalloc` failing to allocate memory.

We observed that UVM performs better than pinned memory. This is largely attributed to sequential nature of write performed. This leverages prefetching done by UVM mechanism.

The use of zero-copy also eliminates the need to copy back results from GPU to CPU improving performance further.

IV. Methodology

Below a general overview of the approach used is provided.

1) Read the tables into CPU memory.

2) Compute Hash of strings using cityhash algorithm and store results in a new table.
3) Generate hash table for table R on joining attribute.
4) Compute upper bound for join output cardinality.
5) If output size is greater than available GPU memory allocate memory using UVM else allocate memory in default way.
6) Perform hash join in GPU using GPU resident hash-table.
7) Materialize the results leveraging multi-threading and remove false positives.

V. Experiments

A. Experimental Setup

- CPU: AMD Ryzen 5 3600X, 6-core 12 threads
- GPU: NVIDIA RTX 2060 super, 8GB
- DRAM: 16GB, 3200MHz

The tables are generated randomly from a list of 447K words [4] as TPC-H has no string attributes that can be joined. For higher sized tables this word list is duplicated and sampled randomly to obtain required number of words.

B. Table Sizes

The table sizes used for experimentation have 100K, 500K, 1 Million, 5 Million, 10 Million rows. 5th table with 10 Million rows had a join cardinality of 500 Million. This is assumed to be able to demonstrate scalability of the approach in case of larger tables. Limited DRAM restricted us from experimenting further with 20 Million tuples table, which had result cardinality of 2 Billion rows.

C. Emulating case when join output size greater than available GPU memory

To demonstrate that the approach works when output size is greater than available GPU memory, a chunk of memory is blocked. This blocked memory is chosen in a way that the output join size cannot completely fit in GPU memory. For the case of 10 Million tuples a total of 5GB out of 8GB GPU memory was blocked memory.

We needed to emulate this cause in case of 20 Million tuples, the join output was 2 Billion tuples. This translated to $>$ 16GB result size which was greater than total available memory making it difficult to obtain the results.

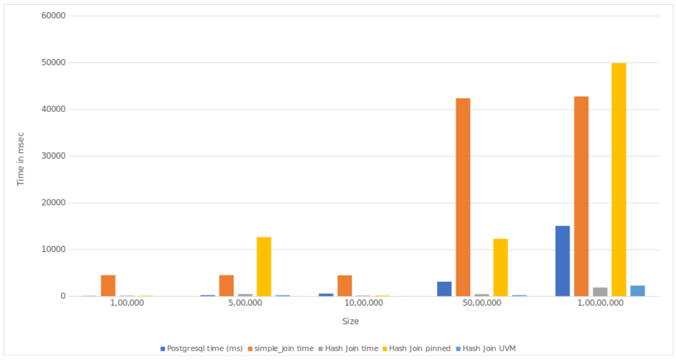

Fig. 2: Results

D. Results

In this section we provide results obtained. We considered 4 different approaches against the postgreSQL database engine.

- Naive GPU Join
- Hash Join on GPU
- Hash join using pinned memory
- Hash Join using UVM

We have blocked memory in case of 5 Million and 10 Million tuples to force our approach to use pinned memory and UVM to obtain the results.

This demonstrates that our approach provides a speedup of almost **8x** over the conventional postgreSQL approach even in worst case. The graph illustrates the same.

1) Materialize Time: To obtain exact results in memory we need to materialize the results. While materializing the results collisions needs to be resolved. The materialize time has considerable overhead as needs to be done in CPU. The materialization is done using multiple threads to reduce this overhead.

Even with materialize time added to our hash computation time we achieve a speedup of more than **2X** over postgreSQL for 10 Million tuples which is the worst case.

VI. Conclusions

- GPU computation presents an opportunity to speed up database operations.
- Pre-computation of join in case of non-partitioned join allows for efficient use of GPU memory without impacting the performance too much.
- Memory oversubscription techniques like UVM and pinned memory allow us to generate join outputs whose size might be bigger than available GPU memory

- Lock free hash maps allow for greater level of parallelism making join computation faster

References

[1] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious hash-joins on gpus," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 698–709.

[2] H. Doraiswamy, V. Kalagi, K. Ramachandra, and J. R. Haritsa, "A case for graphics-driven query processing," *Proc. VLDB Endow.*, vol. 16, no. 10, p. 2499–2511, jun 2023. [Online]. Available: https://doi.org/10.14778/3603581.3603590

[3] "Uvm guide," https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[4] "Word list," https://github.com/dwyl/english-words.