Accelerating Hash Joins on GPUs for OLAP Workloads

Siyuan Lin (slin3), Zhiping Liao (zhiping2)

April 29, 2025

1 Summary

We implemented and iteratively refined a CUDA-based hash-join operator for TPC-H datasets [2], leveraging DuckDB [10] for query planning, metadata extraction, and format conversion. Using AWS `g5-2xlarge` instances, we evaluated our implementation on select-project-join (SPJ) queries extracted from TPC-H queries. Benchmarking across various synthetic distributions showed that our solution outperformed a CPU implementation by 21.5x on TPC-H SPJ queries.

2 Background

We built a well-functional database query execution engine that can load data, parse query, and execute select, project, or join database operators. For the overall engine, the input is a query, and the output is the query results. But we put great focus on the join operator and consider hash join as the physical operator implementation.

2.1 Hash Join

Join operators are among the most common physical operators in database management systems [9]. They are also among the most computationally expensive operators, which has driven significant interest in making them more efficient. A join operator combines rows from two or more tables based on specified predicates, enabling efficient data retrieval and problem-solving for complex business queries.

Most joins are equi-joins, which use only equality comparisons in the join predicate. Hash joins are the most important equi-join physical implementations in analytical processing and are among the most well-studied join algorithms. This motivated us to implement efficient hash joins by leveraging GPU architecture.

A hash join operator takes two inputs: build-side tuples and probe-side tuples. Each tuple consists of hash keys and their row number, i.e., `([key_1, key_2, ...], row_idx)`. Hash join consists of two main steps: the build phase and the probe phase.

- Build phase constructs the hash table by inserting keys from the build table.
- Probe phase searches the hash table for matches using keys from the probe table, emitting matching tuples when found.
- Both phases involve hash computations.
- The key data structure used is hash table.

Below pseudocodes Listing.1 shows a serial version of the hash join algorithm, where we first build a hash table, compute hash keys for the join columns, and insert `(hash_key, row_idx)` into the hash table. Then we probe the hash table, and retrieve the matched index pair `(r_idx, s_idx)` for joining table R and S.

```
function HashJoin(R, S):
    H = empty hash table

    // Build Phase
    for each tuple ([r_key1, r_key2...], r_idx) in R do
        r_key = hash([r_key1, r_key2...]
        H.insert(r_key, r_idx)

    Result = empty list

    // Probe Phase
    for each tuple ([s_key1, s_key2...], s_idx) in S do
        s_key = hash([s_key1, s_key2...]
        if H.contains(s_key) then
            [r_idx1, r_idx2...] = H.lookup(s_key)
            for r_idx in [r_idx1, r_idx2...]
                Result.append((r_idx, s_idx))

    return Result
```

Listing 1: Hash Join Pesudocodes

2.2 GPU Hash Join Algorithm

While joins have been extensively studied in CPU-based mainstream databases, GPUs offer great potential for throughput via massive parallelism and high memory bandwidth. Using the SIMT execution model, GPUs can launch hundreds of thousands of threads simultaneously. This allows both the build and probe phases to be implemented using parallel threads for hash table construction and probing, with appropriate synchronization mechanisms.

Computing the hash keys, building the hash table, and probing the hash table are generally computationally expensive. However, they can all be exploited in a data-parallel way because each element is independent and can be processed in parallel. Nonetheless, they present challenges in fully exploiting GPU performance and achieving optimal throughput.

Parallelizing the hashing process is straightforward and yields good performance. We map each tuple `(keys: [...], row_id)` to a CUDA thread, where hashing is computed locally and results are stored in sequential order. To maximize performance, we implemented a simple hashing algorithm that uses only basic bit operations. Each thread performs coalesced memory reads and writes, thus locality is good, memory bandwidth is fully exploited, and there is barely any thread divergence.

The main challenge lies in table building and probing, which involve random memory accesses that are poorly suited to GPU architecture. Although they can be handled in a naive implementation where each thread is mapped to a probing element, threads attempting to access hash slots and write to global buffers in global memory simultaneously can break the coalesced memory access pattern and data locality, overwhelming the memory bandwidth. For example, each thread probes a key `key` in the hash map, and due to the randomness of hashing, each thread will access random memory locations in the hash table, exhibiting no spatial data locality. CUDA-specific optimizations will be introduced in the following section.

This GPU SIMT execution model is also amenable to SIMD execution because each thread can be mapped to the same instruction (e.g., inserting into the hash table or probing elements in the hash table) over different data. The data is easy to gather: for insertion, the data used is each element in the building table array; for probing, the data used is each element in the probing table array.

3 Approach

We have implemented a simple but functional database execution pipeline that could parse query into query plan, load data, and execute select, project, and join OLAP queries. This execution engine framework is necessary and

helpful for us to evaluate and benchmark on different queries on different data instead of building customized hash join.

Given that we focus on implementing an efficient GPU hash join operator, we have leveraged several open-source projects for query parsing and data processing demands: DuckDB for query analysis and physical plan generation, DuckDB's TPC-H extension for generating Parquet files, and Apache Arrow for loading Parquet files on demand.

Here's a detailed breakdown of our system workflow (Figure.1):

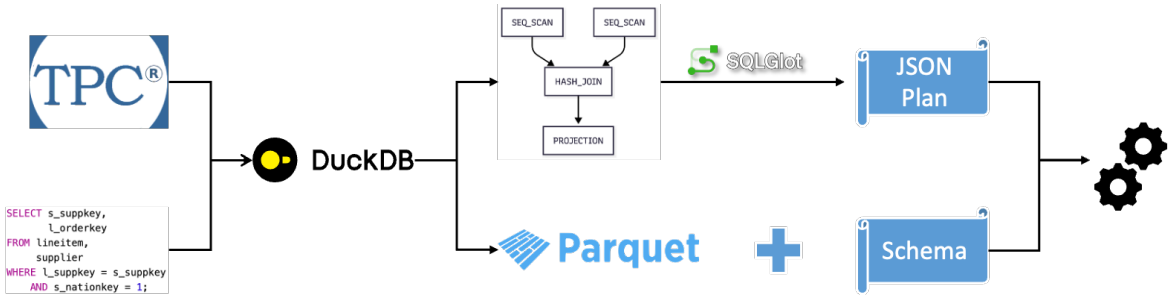


Figure 1: Workflow

1. DuckDB generates a physical plan in JSON format, while SQLGlot [8] parses the expressions in select and join predicates.
2. DuckDB also converts TPC-H datasets into Apache Parquet files and generates their schemas (including column names and data types) in a JSON file.
3. Our program loads both plan and schema JSON files using `nlohmann::json` [4]. It normalizes the plan, binds columns and expressions to the schema, and constructs an internal execution plan with appropriate physical operators (`FilterOperator`, `HashJoinOperatorGPU`, etc.) and expression evaluators.
4. Next, it loads only the necessary tables and columns based on our analysis, minimizing memory footprint. This step is crucial since our implementation lacks a buffer manager and uses a materialization model, which requires that all data be loaded into memory upfront.
5. Finally, the program executes the query, evaluating each operator recursively. The results are collected in a materialized Table.
6. Specially, the join operator is our main focus. The GPU join illustration diagram is shown in Figure.2. To elaborate `HashJoinOperatorGPU` works as below:
   (a) Move columns and data of both building table and probing table that reside on CPU to the GPU side using `cudaMemcpy`.
   (b) Compute hash values of building table's join keys and insert building table's `(hash, row_idx)` into hash table.
   (c) Compute the hash values of the join keys of the probing table and the probe hash table for the matched buckets.
   (d) Probe the hash map using probing table's hash keys, and materialize the matched `(row_idx, row_idx)` results from probing kernels that represents the matched row index pairs of join results from two tables.
   (e) Move the result matched index results from GPU to the CPU sides.
   (f) Materialize the final join results for the table join on CPU.

We have implemented two versions of hash join operator with C++/CUDA:

1. a single-threaded CPU version using `std::multiset` for correctness, debugging and comparison;

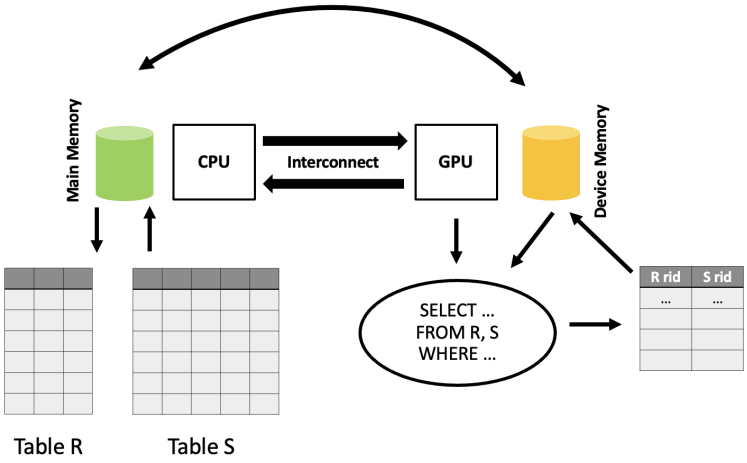


Figure 2: GPU Join Illustration

2. a GPU version using `cuco::static_multiset` for the performance and benchmarking, which is also our focus of the project.

Single-threaded CPU hash join implementation is trivial, one thread executes the algorithm described in Listing.1 iteratively for each element from the building table to insert into the hash table, and for each element from the probing table to look up against the hash table. Below section is a detailed description of implementations for GPU hash join operator.

3.1 GPU Hash Join

For GPU hash join overall, we use `cuco` [5] library as the hash table infrastructure implementation. We use NVIDIA T4 GPU for development, and NVIDIA A10G for benchmarking. `cuCollections` (abbreviated as `cuco`) is a header-only CUDA C++ library that provides GPU-optimized hash maps, sets, bloom filters, and other concurrent data structures. `cuco::static_multiset` and `cuco::static_multimap` are two key parallel data structures that we utilize to perform hash table operations.

A naive and intuitive mapping scheme used for CUDA programming is to map each element to a thread. In our project, for hash table building, the naive mapping is to map each thread to a unique element to insert into the hash table. And similarly for hash table probing, each thread is mapped to a unique element to look up against the hash table. But this naive implementation incurs random memory access due to hash key randomness. Non-coalesced reads and writes will lead to high latency. To mitigate this issue, `cuco` tries to reduce random memory access and thread divergence by exploiting block-level or tile-level synchronization, mapping each element to a group of threads. For `cuco`, open addressing is used for the hash table implementation where each element is hashed to a bucket. When there is a collision for key `key1`, it moves forward to the next available bucket index `key1 + 1` unless there is an empty bucket.

`cuco` applies tile-level mapping where each tile of threads (CUDA `cooperative_groups`) is mapped to the same element `key1` to insert or probe. For insertion, a tile of threads collaboratively iterates over possible buckets (`key1`, `key1 + 1`, `key1 + 2`, ...) inside a window (window size is the tile size) until it finds an available bucket and emits it. This gives good memory access patterns by exploiting spatial L1/L2 cache locality and cheap intra-tile synchronization primitives. For probing, the mapping is similar, and each tile of threads tries to find the matched `key1` inside the tiled window and emit it to the final output. This mapping schemes greatly helps improve the memory access patterns without losing too much efficiency when hash table load balance factor is not too low(in other word, the hash table is not so empty. 0.5 means half full). An illustration to insert `key = 4` is shown below in Figure.3.

Besides the mapping techniques, there are also some noticeable optimizations [7] used for probing:

1. `cuco` uses a shared memory buffer where each thread writes results to shared memory first. The shared buffer is spilled to main memory output when it is full, reducing random global memory writes.

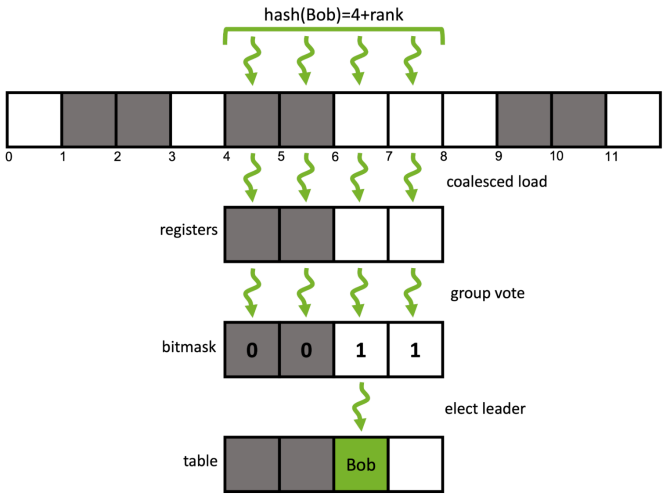


Figure 3: `cuco` cooperative group mapping scheme: Bob is hashed to `key = 4` where 4 is the hash table bucket index. Tile size is set to be 4 where 4 threads responsible for Bob hash key insertion visit bucket index 4, 5, 6, 7 respectively. Bucket index 6 is the first available empty bucket and it is used for store inserted Bob.

2. Warp-level primitives like `Ballot`, `Shfl` are used for warp synchronous operations with lower latency and reduced divergence.

3. Blockwisely local variables used for synchronization within the same block brings less contention against other blocks for global variables. Only update the global variable as kernels are done.

These optimizations make `cuco` kernels very robust and high-performance, saving us much trouble in rewriting the kernels, as the API is easy to use.

With `cuco` support, our GPU hash join operator underwent a number of evolutions. We first started with the `static_multimap` data structure that `cuco` provides, which sounded like an intuitive choice where it stores `(key, row_idx)` pairs, with `row_idx` as the payload.

Nevertheless, the `static_multimap` API limitations made it hard for us to achieve very efficient implementation, and we eventually switched to using `static_multiset` for improved performance.

3.1.1 Multimap Implementation

For database hash join, an intuitive idea is to build a hash table where the key is the hash value of the join columns (if there are multiple join columns, then the hash key would be the hash value of these join column values), and the payload is the `row_idx`.

Under this scenario, if we use the `static_multimap` implementation from `cuco`, the left table is built into the hash table, and the right table is probed against the hash table. When there is a match in the hash table, we emit the matched join pair `(left_row_idx, right_row_idx)`, which is used later to materialize the join results.

However, the current `cuco` hash table API exposes limited usability. For example, the `retrieve` host API used for probing the hash table provides the matched keys and payloads from the hash table, but it does not provide the matched probing `row_idx`, which makes it difficult to retrieve the pair `(left_row_idx, right_row_idx)`. An illustrating example is shown in Figure.4. The retrieved results are the matched key and the payload (`row_idx`). But this limitation in API is also understandable because this is what map lookup should do, look up the key and retrieve the value stored in it.

One workaround is that we can write our own kernels for hash map `retrieve` (lookup). Whenever there is a match in the hash table, emit probing `row_idx` and payload as a pair. However, Due to the existence of multikeys (where multiple rows have the same key), the output size is indeterministic for each probing key (the output size would only be 0 or 1 for non-multikey maps). The device API of `retrieve` is also difficult to use directly because it involves many optimizations specific to a block of threads with cooperative groups and block-wise reduction, making it hard to use and ensure correctness. Thus this workaround doesn't help.

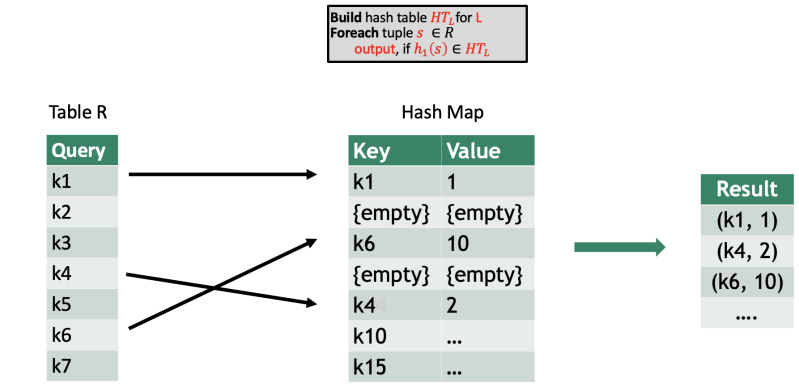


Figure 4: Multimap build and probe illustration

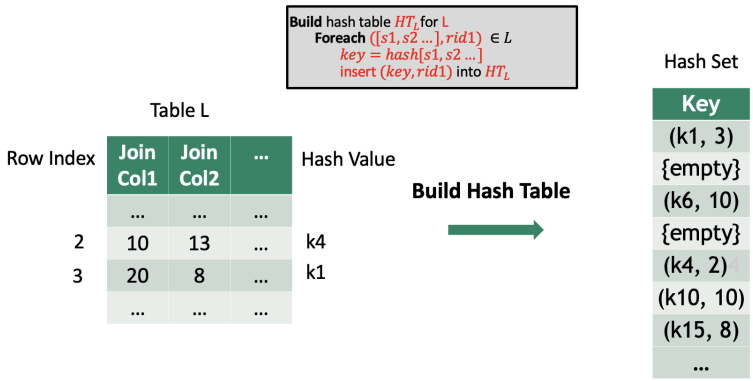


Figure 5: Multiset Build Step

We are left with another workaround, which is to first retrieve all the matched keys, and then iterate over the probing table again to retrieve the matched row indices because there is no direct mapping between row indices and join keys. This approach requires launching an extra kernel to compute the matching probing table row indices, which is inefficient.

3.1.2 Multiset Implementation

The output result `(key, payload)` from hash table lookup motivates us to encode payload into the key so that `payload` could be retrieved directly without extra materialization. Thus we turn to `cuco::static_multiset` implementation as we encode the payload into the key and don't need to store payload.

For multiset, the hash key and row index are encoded together as a pair, forming the element as the key stored in the `static_multiset`. However, the hash value computation of the element in the `static_multiset` is purely based on the hash key (the first element of the pair) by defining customized hash functions.

This method is very powerful when using the host `retrieve` API. When we call `retrieve` from the host side, we can retrieve both the probing side matched keys and the building side matched keys. Since the key itself includes the `row_idx`, it can directly be used to materialize the join columns. The API we use is shown in Listing.2. It gives great facility and has high performance.

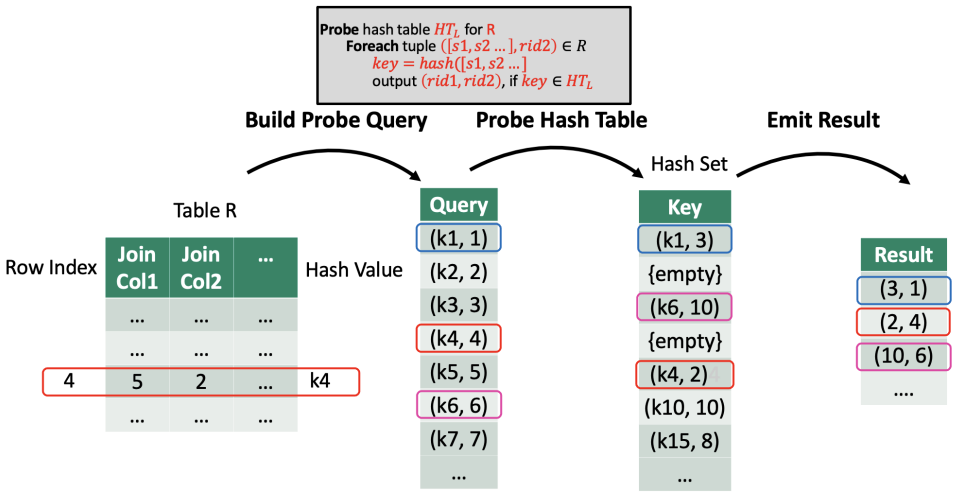


Figure 6: Multiset Probe Step

```cpp
/**
 *
 * @param first Beginning of the input sequence of keys
 * @param last End of the input sequence of keys
 * @param probe_equal The binary function to compare set keys and probe keys
 * for equality
 * @param probe_hash The unary function to hash probe keys
 * @param output_probe Beginning of the sequence of keys corresponding to
 * matching elements in `output_match`
 * @param output_match Beginning of the sequence of matching elements
 * @param stream CUDA stream this operation is executed in
 *
 * @return Iterator pair indicating the the end of the output sequences
 */
template <class InputProbeIt,
          class ProbeEqual,
          class ProbeHash,
          class OutputProbeIt,
          class OutputMatchIt>
std::pair<OutputProbeIt, OutputMatchIt> retrieve(
    InputProbeIt first,
    InputProbeIt last,
    ProbeEqual const& probe_equal,
    ProbeHash const& probe_hash,
    OutputProbeIt output_probe,
    OutputMatchIt output_match,
    cuda::stream_ref stream = {}
) const;
```

Listing 2: `cuco::static_multiset::retrieve` API

This approach that adopts multiset saves the need for an extra kernel computation to find the matched `(left_row_idx, right_row_idx)` pair, significantly improving efficiency. The illustration of how multiset build and probe is shown in Figure.5 and Figure.6 respectively. As we can see, the row index is encoded into the hash set key, then when emit lookup results, it is easy to retrieve the row index.

4 Optimizing GPU Hash Join Operator

GPU hash join operator can be broken down into multiple key components:

- Data movement between CPU and GPU.
- Hash key computation based on the join columns.
- Hash table building.
- Hash table probing.

where data movement is the serial part and it is constrained by the hardware interconnect speed between CPU and GPU. The other parts can all be performed in parallel. We optimize different components in the below section to make the end-to-end execution as fast as possible.

4.1 Moving Hashing to GPU

Initially, we performed hash key computation on the CPU while keeping hash table building and probing on the GPU. We assumed that moving data to the GPU for hash computation would be expensive, especially when join predicates involved multiple columns on each table, as this would increase bandwidth consumption. We were concerned this might create a bottleneck.

However, our profile results told a different story. After moving the hash computations (for both build-side and probe-side keys) to the GPU to leverage its massive parallelism, we saw dramatic improvements. This proved to be the right decision, as the results below show our implementation now consistently outperformed the CPU version across all test cases. The hash key computation is computed based on join columns, and we apply simple mapping scheme based on the GPU hardware: we map each thread to one row, and each thread is responsible for computing the hash key for the join cols of that row; then each thread is responsible for writing the results back to the output buffer. Coalesced memory reads and writes are enabled by this kernel and this kernel runs very fast.

Later, we profiled this solution in Nsight Systems [1] shown in Figure.7 to 1) confirm the elimination of CPU-side hash computing and 2) look for any inefficiencies to improve:

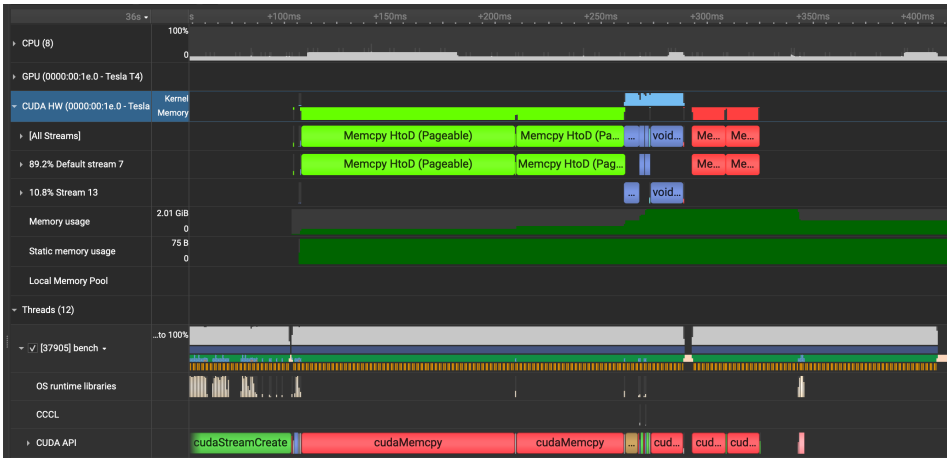
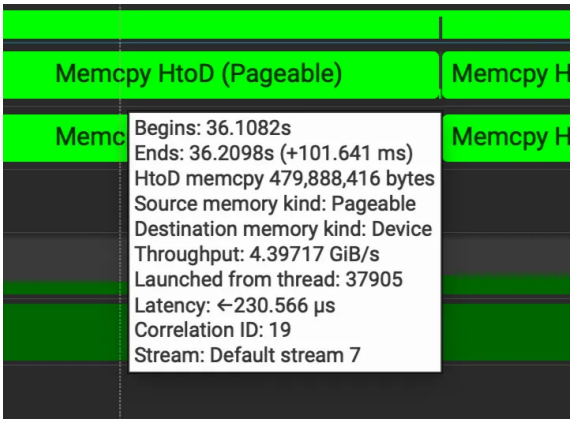


Figure 7: Nsight Systems: moving hashing to GPU

Our analysis revealed that memory copies—especially host-to-device transfers—dominated the join computation time, while table building and probing appeared minimal. Figure.8a showed that using pageable source memory slowed down the transfer speed. Based on **TODO: fill the references**, we believed that using pinned memory could improve throughput.

4.2 Failed Attempt to Utilize Pinned Memory

When `cudaMemcopy`'ing an arbitrary (pageable) host memory region to VRAM, there are actually two copies under the hood: host paged memory → host pinned memory → GPU memory. `cudaMallocHost` allows



(a) Pageable `cudaMemcpy` hurts throughput

```
Device: Tesla T4
Transfer size (MB): 16

Pageable transfers
  Host to Device bandwidth (GB/s): 4.431851
  Device to Host bandwidth (GB/s): 4.549689

Pinned transfers
  Host to Device bandwidth (GB/s): 6.236401
  Device to Host bandwidth (GB/s): 6.560652
```

(b) NVDIA T4 Memory Transfer Benchmark

us to allocate pinned memory, and data movements between pinned memory and VRAM are expected to be much faster. Moreover, with pinned memory, data movement between CPU and GPU can be performed in an asynchronous way using `cudaMemcpyAsync` and can overlap with kernel execution. However, we realized that in our use case, moving data from paged memory to pinned memory cannot be eliminated. The hash-join operator accepts results from its dependent operators, which work with smart pointers and dynamically managed containers. We could not easily change their underlying allocator. In addition, `cudaMallocHost` that allocates pinned memory or `cudaHostRegitser` that pins the pageable memory comes with a huge overhead, since it takes time to pin memory pages and it often involves OS activities.

Still, we wanted to be clever and exploit its properties. We noticed that many wasted data were copied into VRAM because we `cudaMemcpy` the whole necessary columns and active row numbers. It came to us that pinned memory could be an excellent place for buffering active rows: Only active rows were materialized into pinned memory. Although the total transfer numbers were the same (2 times), the amount of data transferred was less: fewer rows and no row ids were copied.

Unfortunately, this 'optimization' turned out to hurt performance a lot. We had two explanations for it:

1. Pinned memory only brought minimal improvement in terms of throughput. We benchmarked NVIDIA T4 memory transfer throughput using the methods proposed by [3], which indicated only an improvement 40% (Figure.8b). What made it worse was that `cudaMallocHost` was very expensive, shadowing the improvements of a higher transfer bandwidth.

2. Materialization does not actually save time. Based on our profile result in Figure.9, materialization (or column compaction in the figure) introduced a significant penalty due to poor memory locality (indirections), shown in Listing.3.

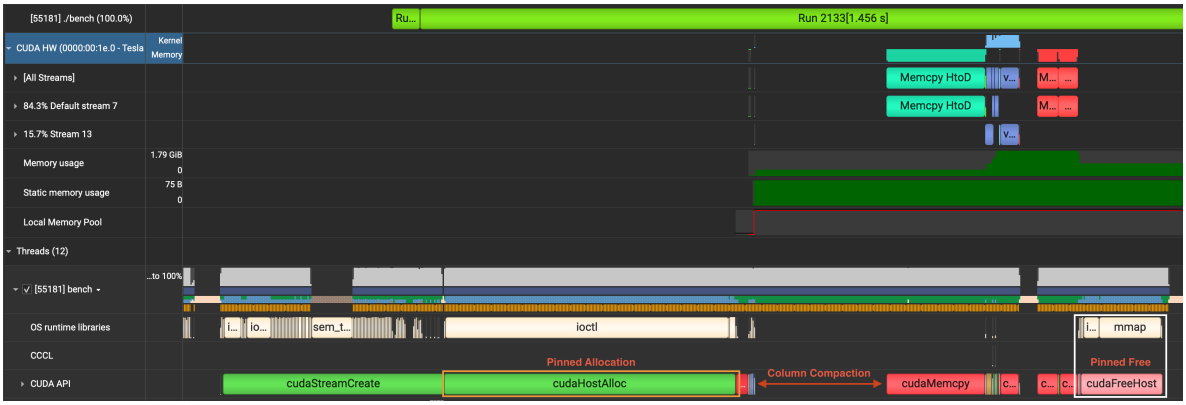


Figure 9: Failed Attempt

```
for (size_t j = 0; j < num_tbl_rows; j++) {
  pinned[j] = column.int_data[tbl.row_ids[j]];
}
cudaMemcpy(d_int_data, pinned, num_tbl_rows * sizeof(int),
           cudaMemcpyHostToDevice);
```

Listing 3: Column Compaction

4.3 Kernel Fusion

Previous implementation has a separate kernel for computing hash join keys for table join columns. This hash key computation is necessary because if we have multiple columns to join, we need to encode the multiple columns into one hash value as the key could not be a vector of values as required by `cuco` libraries, and indeed it is also a common practice to encode it into some hash value.

The Nsight Systems profiling for scale factor 10 is shown in Figure.10. It can be observed that the hash key computation kernel `compute_hash_key_kernel` is actually non-trivial compared to the hash table probing kernel `retrieve`. `compute_hash_key_kernel` is relatively lightweight while The `retrieve` kernel is a key function and involves many memory reads and writes, but `compute_hash_key_kernel` generates non-trivial overhead, which motivates use to try to make it as minimal and have fewer overheads.

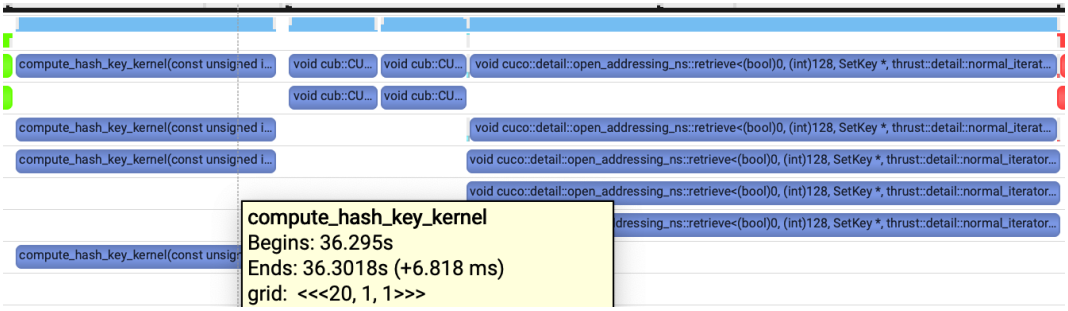


Figure 10: Nsys Profiling without Kernel Fusion

A new optimization point is to fuse the hash key kernels and hash table probing kernels, so that there are no extra materialization costs of the hash keys. Otherwise, we need to first compute the hash keys for each row, write them into global memory, and load them during `retrieve` kernel. Kernel fusion introduces fewer kernel calls and saves extra data writes and reads.

And we handle this hash key computation as a lazy evaluation where we transform the iterator as illustrated in Listing.4, but it is evaluated only when `first` is accessed, in other words, the hash key is computed only inside the `retrieve` kernel when then iterator is dereferenced. This gives so much facility, in both readability and performance.

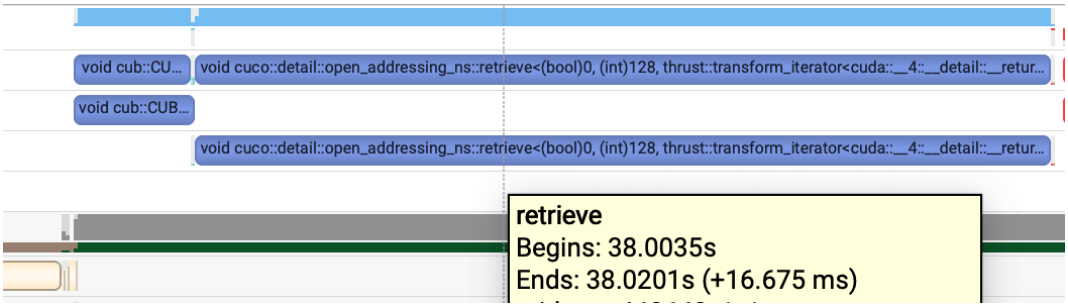


Figure 11: Nsys Profiling with Kernel Fusion

```
auto first = thrust::make_transform_iterator(
    d_row_ids,
    cuda::proclaim_return_type<SetKey>(
        [data_ptr = d_columns, num_left_cols] __device__(RowID idx) {
          HashType seed = num_left_cols;
#pragma unroll
          for (int j = 0; j < num_left_cols; ++j) {
            seed ^= data_ptr[j].get_hash(idx) + 0x9e3779b9 + (seed << 6) +
                    (seed >> 2);
          }
          return SetKey(seed, idx);
        }));

auto last = first + num_ltbl_rows;
```

Listing 4: Hash Value Computation Kernel Fusion

Using Nsight System profiling in Figure.11, it could be found that the compute hash kernel doesn't exist, and there is only retrieve kernel left. Originally, the compute kernel is 7ms, and retrieve kernel 15.7ms. And now the retrieve kernel increases only by 1ms due to hash key calculation (16.7ms now), but it reduces the overall kernel time for probing by 30%, which is a huge improvement overall.

4.4 GPU Memory Pool

Memory allocation costs are non-trivial in both CPU and GPU, especially when allocating large memory. For GPU, `cudaMalloc` is actually very time-consuming because it involves syscalls to the CUDA driver and possible OS page allocation, which has high latency. From the profiling below for scale factor 10, it could be found that the `cudaMalloc` and `cudaFree` take up a non-trivial portition of join time, especially when large memory is allocated and deallocated, the nsight system profile is shown in Figure.12a.

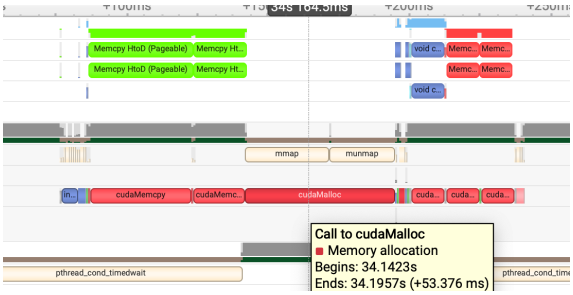
A way to tackle this is that we can use a memory pool for GPU memory, and this memory pool is created at the start of the program and only destroyed when the program is finished. It is beneficial in that memory allocation and deallocation would be very fast as no CUDA driver and OS page allocation is involved anymore. Below is the improvement by using the memory pool, and around 10% improvement is witnessed in Table.1 by running TPCH simple query in Listing.5 wit different scale factors. The memory pool we use is NVIDIA Rapid Memory Management (RMM) [6].

Implementation	SF=1	SF=2	SF=3	SF=5	SF=10
Cuda Malloc (s)	0.022	0.031	0.031	0.059	0.122
Memory Pool (s)	0.018	0.027	0.030	0.053	0.096

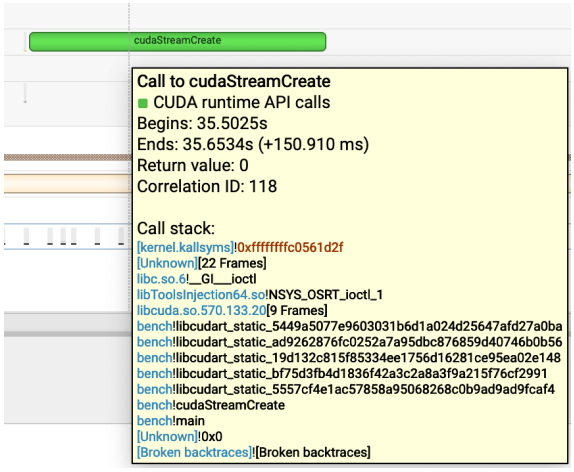
Table 1: Performance comparison between `cudaMalloc` and memory pool on A10G.

4.5 GPU Global Context

GPU kernels replies on specific stream for execution, and kernels within the same stream is executed in order. But from the Nsight Systems profile results as below, the stream creation is a non-trivial part in Figure.12b.



(a) No memory pool is used on GPU side



(b) Nsight Systems profiling for `cudaStreamCreate`.

A way to workaround this is to create a `cudaStream` globally stored in execution context, and it could get reused to amortize the creation and destruction costs. And it improves the performance by around 10%.

5 Results & Analysis

The hardware specification of the machine we used for benchmark is shown in Table.2 for GPU and in Table.3 for CPU. It is a fairly good machine that we have the budget to afford. And it has better performance than the machine T4 we used for development and test.

Model	Memory Size	Memory Bandwidth	PCI-e Bandwidth	SM Count	L1 Cache	L2 Cache
NVIDIA A10G	24 GB	600GB/s	16 GB/s	72	128 KB	6 MB

Table 2: Benchmark Machine GPU Specification

Model	Memory	Cores (Threads)	L1 Cache	L2 Cache	L3 Cache
AMD EPYC 7R32	30GB	4 (8)	128 KB (L1d) + 128 KB (L1i)	2 MB	16 MB

Table 3: Benchmark Machine CPU Specification

5.1 TPC-H

We initially planned to support most of the TPC-H queries. However, we discovered that most of them required aggregation and complex string pattern matching, which was not the focus of our project. Moreover, such queries would usually result in mild plans, as the optimizer could push down filters or choose a good join order, reducing the number of rows handled by a join operator. We designed two (Listing.5, Listing.6) simple but representative queries on TPC-H datasets:

```
select s_suppkey, l_orderkey
from   lineitem, supplier
where  l_suppkey = s_suppkey and s_nationkey = 1
```

Listing 5: `LineitemSupplierSimple`

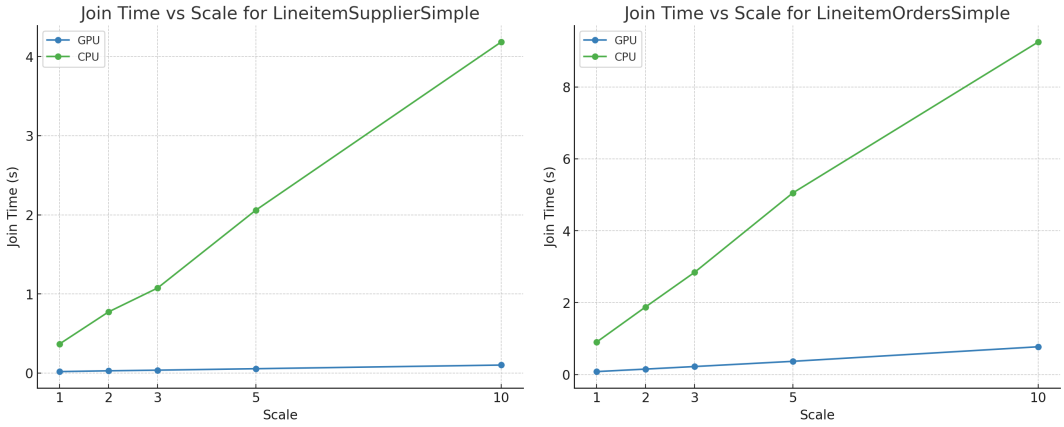
```
select l_orderkey
from   lineitem, orders
where  l_orderkey = o_orderkey
```

Listing 6: `LineitemOrdersSimple`

We benchmarked both our CPU and GPU implementation with scale factor ranging from 1 to 10. The size of each table at different scale factor is listed in Table.4. Our CUDA implementation showed massive leadership across all settings, outperforming single-threaded CPU implementation by 21.5x. The advantage between CPU and GPU continued as the problem size (scale factor) grew.

Table 4: Number of Rows per Table at Different Scale Factors

Table Name	SF = 1	SF = 2	SF = 3	SF = 5	SF = 10
customer	150000	300000	450000	750000	1500000
lineitem	6001215	11997996	17996609	29999795	59986052
nation	25	25	25	25	25
orders	1500000	3000000	4500000	7500000	15000000
part	200000	400000	600000	1000000	2000000
partsupp	800000	1600000	2400000	4000000	8000000
region	5	5	5	5	5
supplier	10000	20000	30000	50000	100000



(a) `LineitemSupplierSimple` Join Time

(b) `LineitemOrdersSimple` Join Time

Figure 13: TPC-H Dataset Performance

We investigated the performance of our GPU implementation on `LineitemSupplierSimple` when scale factor = 10 (Figure.14). We discovered that data movement, particularly host-to-device `cudaMemcpy` was the bottleneck. We believe that it is difficult to shorten or hide the latency here unless the entire workflow is done on the GPU. Still, there remains a great question about when to move from or to the GPU. We think such a decision should be determined by a cost-based query optimizer, which is beyond the scope of this project. Currently as we are performing GPU in-memory join, thus larger scale factor will lead to larger tables that could not fit on GPU main memory and would require out-of-core execution, which is outside this project's scope. Thus the largest scale factor we benchmark is scale factor = 10.

5.2 Multi-key Join Analysis

Multi-key joins are crucial in modern OLAP workloads. We want to investigate how our CUDA implementation responds to different join key sizes. TPC-H only supports up to 3 meaningful join keys, which might not be a

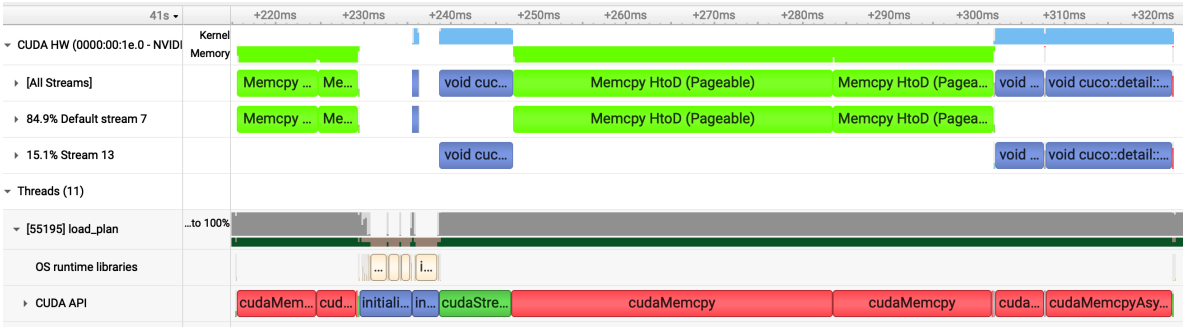


Figure 14: Nsight Systems Profile for TPC-H, `LineitemSupplierSimple`, Scale Factor = 10

great choice for this experiment. We generate a synthetic data set with five join keys defined in the Listing.7. To make the data comparable across different key sizes, we made sure that all queries in Listing.7 output the same number of rows regardless of key sizes. In this case, we can isolate how key size affects performance.

```
keys = np.random.permutation(num_rows)

left_df = pd.DataFrame({
    'l1': keys,
    'l2': keys % 1000,
    'l3': keys % 100,
    'l4': keys % 10,
    'l5': keys % 5,
    'payload': keys
})

right_df = pd.DataFrame({
    'r1': matching_keys,
    'r2': matching_keys % 1000,
    'r3': matching_keys % 100,
    'r4': matching_keys % 10,
    'r5': matching_keys % 5,
    'payload': matching_keys
})
```

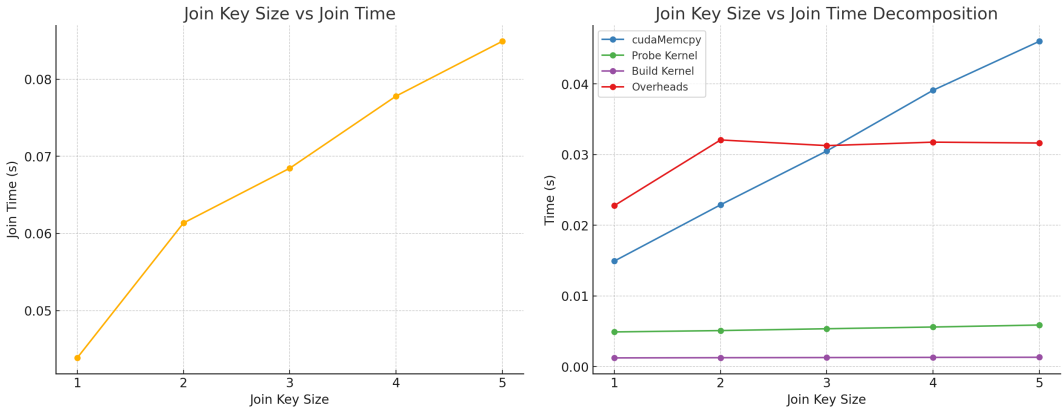
Listing 7: Multi-key Join Table Generator

```
SELECT *
FROM left_table
JOIN right_table
ON l1 = r1 AND l2 = r2 AND l3 = r3 AND [...] -- depending on the key size
```

Listing 8: Multi-key Join Queries

The results showed that increasing the size of the join key led to more data movement and therefore increased the overall join time. This is expected because hash computations are done on GPU. The more join keys are, the greater the number of columns that will be sent to the GPU. Our inner hash representation is a 32-bit integer, independent to the key size. Therefore, the time consumed by other parts of the algorithm (probe kernel, build kernel) remained the same.

Overheads in our plot primarily came from host memory allocation. Since we intentionally kept the result row numbers the same, it did not change much as the join key size grew.



(a) Multi-key Join Time

(b) Multi-key Decomposition

Figure 15: Multi-key Join

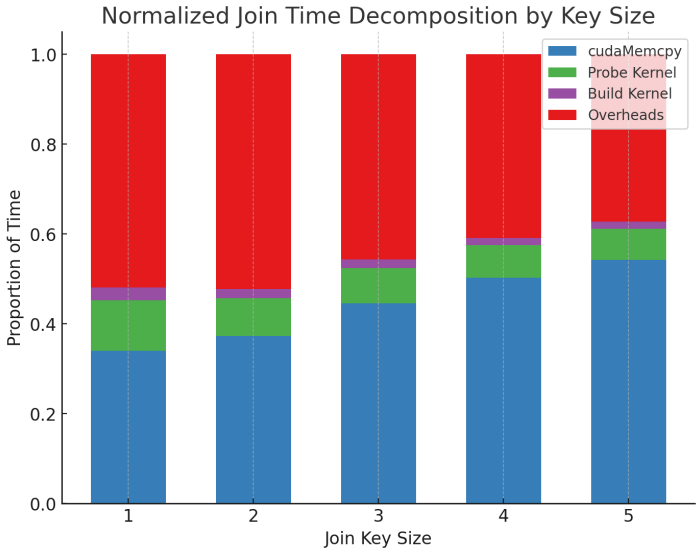


Figure 16: Multi-ket Join Normalized Decomposition

5.3 Skew Join Analysis

Skewed key distribution is another major challenge for a join operator. It happens when either of the join keys contains a large number of duplicated values, generating a huge amount of rows (in the worst case, $|\text{left\_table}| \times |\text{right\_table}|$ rows, which is prohibitive when the table size is large).

Zipf's law is commonly used to model real-world data skewness [11]. Zipf(α) has a tunable parameter α which determines how much data are concentrated to a few numbers. α must be larger than one, and the greater it is, the more concentrated the data will be. We used `numpy` to sample values from a Zipf distribution. The tables and the query are defined in Listing.9.

```
-- Tables:
CREATE TABLE left_table (l1 int, left_payload int)
CREATE TABLE right_table (r1 int, right_payload int)
-- Query:
SELECT * FROM left_table JOIN right_table ON l1 = r1
```

Listing 9: Skewness Experiment Table Definition and Query

(a) Zipf α vs Result Rows (b) Zipf α vs Join Time (c) Skew Join Time Decomposition

Figure 17: Skew Join

Figure 18: Nsight Systems Profile, $\alpha = 2.5$

As the skewness increases, we observed a growing gap between the actual join time and the combined time of `cudaMemcpy`, table building, and probing, as Figure.17c suggested. This occurs because highly skewed data generate an enormous number of output rows. For example, when $\alpha = 2.5$, the output reaches 57% of $|\text{left\_table}| \times |\text{right\_table}|$, causing host memory allocation to become the dominant factor in execution time, shown in Figure.18.

5.4 Speedup Analysis

5.4.1 Overall Speedup

From the above execution time breakdown, it could be found that the speedup is bus transfer bound, in other words, data movement between CPU and GPU takes up the most of the time because it is very slow (16B/s) on our benchmark machine A10G. The time used for data movement is orders of magnitude larger than parallel kernel execution for hash map building and probing. It also implies that even if we can apply async memory transfer using `cudaMemcpyAsync` to hide kernel costs, the total data movement time still greatly dominates, as could be seen by Figure.16. There is no way that we could reduce data transfer, because we would need all the data to do hash table building and probing. Moreover, there are some extra overheads that include CPU memory allocation and API calls, and similarly there is nothing we could do to reduce the overheads because they are the necessary system overheads.

Ignoring the overheads, from Amdal's law, the overall execution speedup is bottlenecked by the sequential data movement between CPU and GPU. The speedup is greatly limited by the interconnect PCI-e bandwidth. The way the overall execution could be improved is to have powerful machine with faster CPU-GPU interconnect bandwidth. The latest H100 machine provides 450GB/s unidirectional bandwidth with NVLINK compared to 16GB/s on our benchmark A10G GPU, and we believe that H100 will greatly improve the execution time by reducing data transfer time, but whether data transfer is still a bottleneck needs further benchmarks. But unfortunately we don't have have access to more advanced machine due to budgets.

5.4.2 Kernel Speedup

But if we goes deeper into the kernel speedup, it could be found that the kernel has variant performance under different benchmark tests, but they all have very good performance. There are two profiling results for expensive

`retrieve` kernels shown respectively in Figure.19 and Figure.20. The profiling result are almost the same even if skewed join will have more output matches in the hash table and be suppose to bring more overheads. The compute utilization is high which means it makes a good use of the parallel cores.

Besides memory access patterns are very good with high cache hit rate (more than 70%) in Figure.21. It sounds weird at first glance why compute utilization is high because hash table lookup only needs to do simple hash computation and comparison but requires do a lot of random memory read and writes into output buffer. Moreover the hash table for scale factor = 10 would excced the size of L1 and L2 cache, how could it possible to still have good memory access behaviors. In fact, it makes sense because intra-warp computations, optimizations, and synchronizations (like reduce) are applied to reduce random and enhance coalesced memory access [5]. Thus more computations are incurred to improve memory access patterns. And the overall performance is greatly optimized so far.

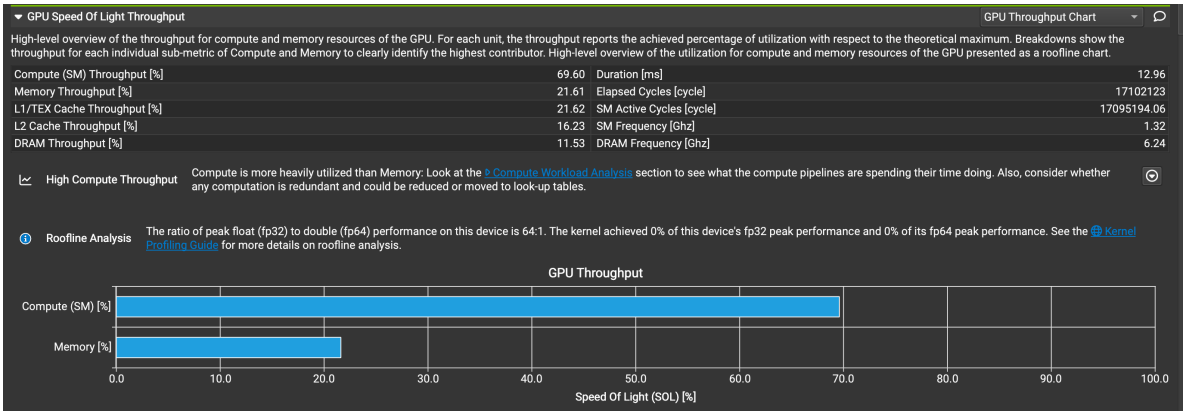


Figure 19: TPCH SF=10 LineitemSupplierSimple `retrieve` Kernel

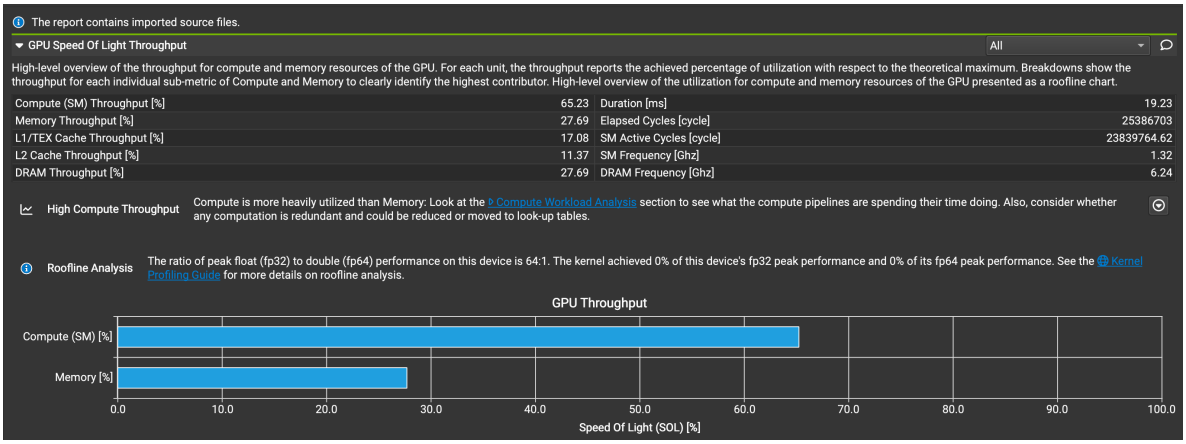


Figure 20: Skew Join `retrieve` Kernel

For hash table `insert` kernels in Figure.22, the situation is different where compute throughput is extremely low, and memory throughput dominates though still very low. This is because this kernel is fairly easy and not too much optimization could be done.

Moreover, for hash table lookup, as different tiles of threads are responsible for different data, there will be highly likely thread divergence because they will hit or miss in the hash table lookup and have different control branches, which depends on the data itself, which could be validated from Nsight Compute profiling result in Figure.23 where the thread states are mostly stalled for branch resolving. This is unavoidable latency overhead.

All in all, the kernel performance is very good and not too much optimization for speedup could be exploited so far. And since the `retrieve` kernel from `cuco::static_multiset` fits our need very well, we can use it very handy and have minimum overheads.

It is worthwhile to mention, for join operator implementation, CPU would not be a better choice even with multi-threaded implementations because GPU have much more parallelism to exploit and demonstrate very

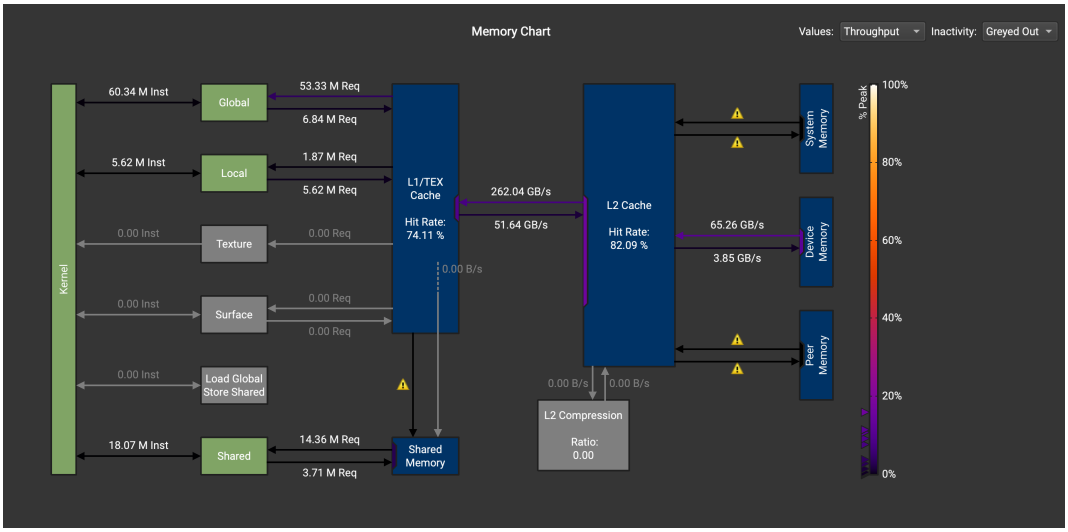


Figure 21: TPCH SF=10 LineitemSupplierSimple `retrieve` Kernel Memory Chart

good speedup even faced with GBs or TBs of data and have cheaper synchronizations within the same block and within the same warp. In contrast, CPU thread synchronization is much more expensive. Thus GPU sounds like a more sound target machine and it can have so many potentials to be explored.

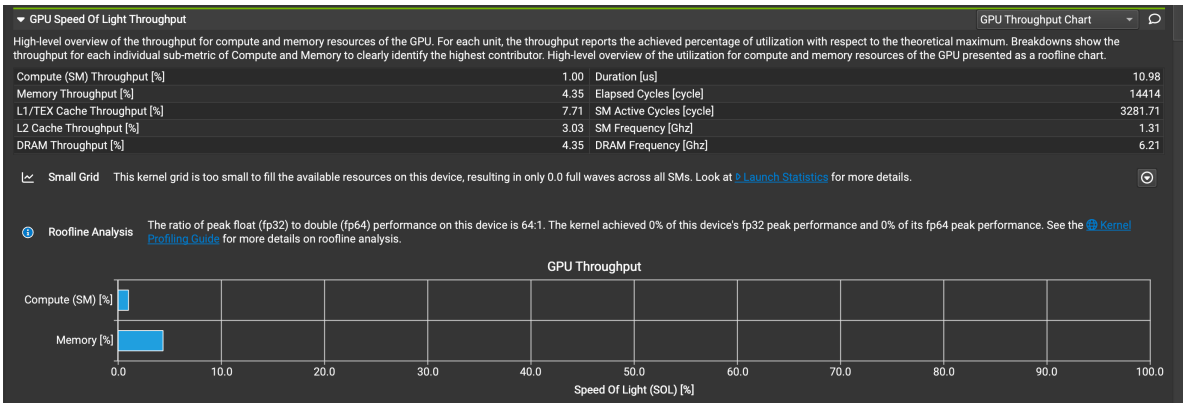


Figure 22: TPCH SF=10 LineitemSupplierSimple `insert` Kernel Throughput Profiling

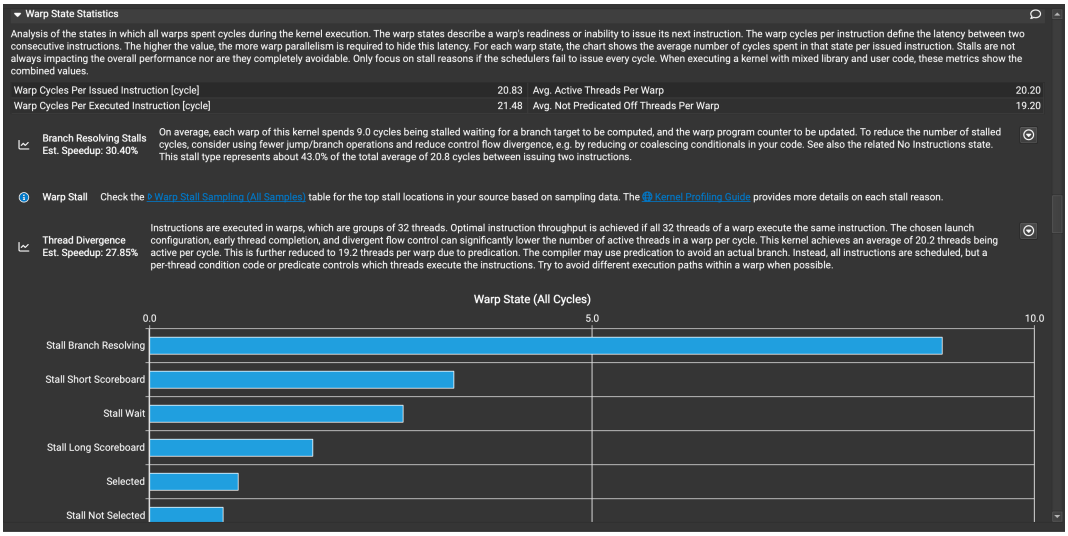


Figure 23: TPCH SF=10 LineitemSupplierSimple `retrieve` Kernel Warp State Statistics

6 Conclusion

In this research project, we have built a simple yet functional database query execution engine focused on accelerating query performance using a GPU-based hash join. We leveraged NVIDIA's high-performance cuCollections library to build a custom GPU hash join operator on top of its parallel hash table infrastructure.

Even with GPU parallelism, there are much optimization room that will reduce unnecessary overhead and make the end-to-end hash join more efficient. Along the road, we optimized and benchmarked our implementation across datasets of varying sizes and distributions, achieving significant speedups compared to a naive CPU implementation. However, profiling revealed that unavoidable overheads—such as CPU–GPU data movement and CPU-side memory allocations—still dominate the end-to-end execution time. These bottlenecks limit the overall speedup, even when the GPU parallel execution is highly optimized. In future work, we plan to explore techniques such as improving host-side memory management, trying on more powerful machines and pushing more components of the query execution pipeline onto the GPU to further minimize these overheads and unlock greater performance gains.

References

[1] NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[2] TPC Benchmarks Overview. https://www.tpc.org/information/benchmarks5.asp, 1993.

[3] How to optimize data transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/, December 2012.

[4] JSON for Modern C++. https://github.com/nlohmann, April 2013.

[5] NVIDIA/cuCollections. https://github.com/NVIDIA/cuCollections, April 2019.

[6] Fast, flexible allocation for cuda with rapids memory manager. https://developer.nvidia.com/blog/fast-flexible-allocation-for-cuda-with-rapids-memory-manager/, 2020. NVIDIA Developer Blog.

[7] Maximizing performance with massively parallel hash maps on gpus. https://developer.nvidia.com/blog/maximizing-performance-with-massively-parallel-hash-maps-on-gpus/, 2021. NVIDIA Developer Blog.

[8] Toby Mao. tobymao/sqlglot. https://github.com/tobymao/sqlglot, April 2021.

[9] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, March 1992.

[10] Mark Raasveldt and Hannes Mühleisen. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Yue Yang and Jianwen Zhu. Write skew and zipf distribution: Evidence and implications. *ACM Trans. Storage*, 12(4), June 2016.