Project Title: GPU-Accelerated Joins on CSV Files Using CUDA

To reiterate on the project, I am planning on leveraging GPU parallelism using CUDA to speed up the process of performing "SQL-like" joins on CSV files, focusing on efficiency for large-scale data.

Public Github: https://github.com/kappavi/cuda_accelerated_csv_joins/tree/main

(reiterating) Goals:

- 75% Goal: A basic INNER JOIN will be implemented using CUDA to join two CSV tables on a common key column.
- 100% Goal: This implementation will be extended to support both LEFT and RIGHT JOINS using CUDA, where the LEFT JOIN will include all rows from the left CSV file, with NULLs where there's no match in the right file (and vice versa for the RIGHT JOIN).
- 125% Goal: This implementation will support full OUTER JOIN operations or complex multi-join operations that support joining 3+ CSV tables at once.

Goals met: 125%

Accomplished:

- Implemented inner join with CPU
- Implemented inner join with GPU
- Implement left join with CPU/GPU
- Implemented right join with CPU/GPU
- Implemented full outer join with CPU/GPU
- Successfully leveraged GPU/CUDA to join tables faster than CPU (up to 14x speedup)
- Confirmed identical CSV files for CPU/GPU output over 8 test cases (from test sizes 10 to 1,000,000)

Implementation:

Firstly, I had to implement joins with CPU as a base metric to compare GPU implementation. Note that the images of code in this report will often be just the method header in the case they are too long. The actual code can be see in the public github repository, linked at the top of this document.

```
void joinCPU(const vector<vector<string>> &table1,
             const vector<vector<string>> &table2,
             const string &column1,
             const string &column2,
             const string &outputFilename,
             const string &joinType) {
    // Open the output file
    ofstream outFile(outputFilename);
    if (!outFile.is_open()) {
        cerr << "Failed to open output file: " << outputFilename << endl;
        return;
    }
```

This is the general join function. Note that we must pass a parameter for the two tables we are joining, the two columns we are to join on, an output file name, and the join type (left, right, or outer).

The inner join was the first function that was coded as a performance metric, which is basically the same function except there was no input for kind of join. The large-size csv file tests were performed using inner join:

```
    // Perform the join
    for (size_t i = 1; i < table1.size(); ++i) {
        for (size_t j = 1; j < table2.size(); ++j) {
            if (table1[i][colIndex1] == table2[j][colIndex2]) {
                // Print the matching row
                for (const string &val : table1[i]) outFile << val << ",";
                for (size_t k = 0; k < table2[j].size(); ++k) {
                    if (k != colIndex2) outFile << table2[j][k] << ",";
                }
                outFile << "\n";
            }
        }
    }
    outFile << "\n";
    outFile.close();
```

```
void innerJoin(const vector<vector<string>> &table1,
               const vector<vector<string>> &table2,
               const string &column1,
               const string &column2,
               const string &outputFilename) {


    ofstream outFile(outputFilename);
    if (!outFile.is_open()) {
        cerr << "Failed to open output file: " << outputFilename << endl;
        return;
    }
```

This inner join function basically finds rows in two tables where the values in the specified columns match, and writes the combined rows to an output file. If no matches exist for a row in either table, it is excluded from the result. There are a couple of different methodologies to implement join logic, and this uses nested for loops to do so.

After I confirmed that these were working with a few initial tests, I had to implement the joins with CUDA, which were much more complex. I started with implementing inner join with CUDA. This involved a two step process: creating a kernel for comparing values and marking matches and then creating the actual inner join function for batching the comparison and handling memory transfers between the host (CPU) and device (GPU). Some code samples for those (the kernel on the left, and some batch logic for the actual function on the right):

```
__global__ void innerJoinKernel(const char *table1Keys, const char *table2Keys,
                                int *matches, int table1Size, int table2Size, int keyLength) {
    int tid1 = blockIdx.x * blockDim.x + threadIdx.x;
    int tid2 = blockIdx.y * blockDim.y + threadIdx.y;

    if (tid1 < table1Size && tid2 < table2Size) {
        // Compare the keys
        bool isMatch = true;
        for (int i = 0; i < keyLength; i++) {
            if (table1Keys[tid1 * keyLength + i] != table2Keys[tid2 * keyLength + i]) {
                isMatch = false;
                break;
            }
        }

        // If match, record the indices
        if (isMatch) {
            atomicAdd(&matches[tid1 * table2Size + tid2], 1);  // Use atomic operation for thread safety
        }
    }
}
```

```
for (int batchStart = 0; batchStart < table1Size; batchStart += batchSize) {
    int currentBatchSize = min(batchSize, table1Size - batchStart);

    // Allocate memory for current batch
    char *d_table1Keys;
    cudaMalloc(&d_table1Keys, currentBatchSize * keyLength);
    cudaMemcpy(d_table1Keys, h_table1Keys + batchStart * keyLength, currentBatchSize * keyL

    cudaMalloc(&d_matches, currentBatchSize * table2Size * sizeof(int));
    cudaMemset(d_matches, 0, currentBatchSize * table2Size * sizeof(int));

    // Launch kernel for this batch
    dim3 blockSize(16, 16);
    dim3 gridSize((currentBatchSize + blockSize.x - 1) / blockSize.x,
                  (table2Size + blockSize.y - 1) / blockSize.y);
    innerJoinKernel<<<gridSize, blockSize>>>(d_table1Keys, d_table2Keys, d_matches,
                                             currentBatchSize, table2Size, keyLength);
```

Regarding CUDA logic, we are trying to perform the following optimizations:

- Using shared memory to minimize global memory access latency,
- leveraging coalesced memory access patterns to ensure threads access memory contiguously,
- using pinned memory for faster host-to-device and device-to-host transfers,
- and batching large computations to prevent GPU memory overflows while maintaining efficient parallel execution.

We need to utilize a kernel for using shared memory. Based on my work in my GPU class (CS 8803), you have to create a kernel for processing a subset of rows in parallel by loading chunks of data into shared memory. Specifically in the context of inner join, the kernel compares rows from one table to rows in another table, row by row, in parallel. Shared memory helps by storing part of the second table locally for each block of threads, reducing expensive global memory accesses.

The kernel logic was by far the most complicated implementation but the most effective (as I have also seen in my GPU projects this semester). The other optimization techniques were more straightforward, for example, using cudaMemcpy to transfer data efficiently between the host (CPU) and device (GPU) allowed us to load tables into the GPU memory for parallel processing. Additionally, batching in sizes of 1,000 rows ensured that memory usage remained within the limits of the GPU's memory capacity, which was critical for avoiding segmentation faults (this was an issue I will discuss later in the challenges section).

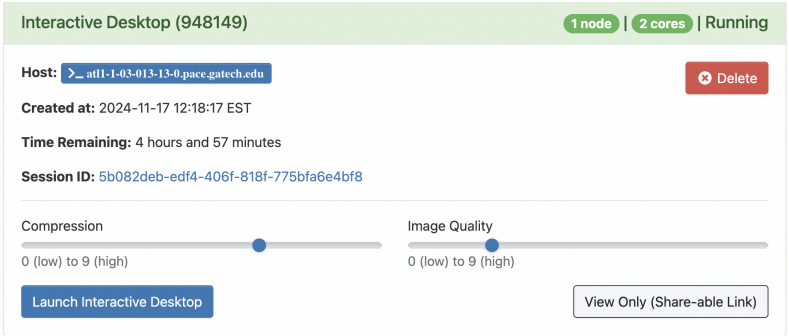
Coalescing memory access was another key optimization; this ensured that threads in the same warp accessed contiguous memory addresses, reducing memory access latency and making global memory operations significantly faster.

I had to play around with the batch size to find the optimal balance between maximizing parallelism and avoiding segmentation faults due to excessive memory allocation. For example, smaller batch sizes resulted in underutilization of the GPU, while larger batch sizes sometimes exceeded the available memory, leading to crashes. Initially, I was performing no batching at all, but this caused a segmentation fault at a CSV testing size of 100,000, making me realize I would need to batch the input. I understood at this point that the space complexity was proportional to the square of the size of the CSV file since for each batch, the GPU kernel compares every row from one batch of table1 with all rows of table2, resulting in a matrix of matches with dimensions dependent on the sizes of the batches. After testing different values (from 1,000 to 10,000), I determined that a batch size of 1,000 worked well for most scenarios while maintaining efficiency.

The CUDA logic for the general join function was very similar, so I won't go into detail about it here.

Environment and Testing:

While creating the initial inner join function and testing on CPU, I was in a local environment, using a CMakesList.txt file to compile my c++ files. However, once I started adding GPU implementation, I utilized a GPU on the PACE cluster. I used a NVIDIA GPU H100 HGX with 1 node, 2 cores per node, 1 GPU per node, and 8 GB of memory per core. Naturally, this led me to log onto the GT VPN to access the pace cluster:



When starting CUDA development, I converted my file from .cpp to .cu. On the pace cluster, I loaded a gcc

module of version 12.3.0 and used NVCC, a NVIDIA CUDA compiler, to compile the code. I would then run the output.

I had 12 different tests created. To summarize each of them:

- Tests 1 – 3 ensured basic join functionality (columns matched, columns didn't match, columns matched but elements didn't, etc.)
- Tests 4 – 8 ensured compared CPU and GPU performance on CSV tables of sizes 10, 100, 1,000, 10,000, 100,000, and 1,000,000). I used the chrono library to measure the execution time for both CPU and GPU implementations. For each CSV table, I had created two tables, one that had from elements 1 to CSV_size and the other that had elements from CSV_size // 2 to 1 + CSV_size // 2, such that there would be an overlap of 50% of the rows for each pair of tables.
    - Test 7 took a ridiculous amount of time to run on CPU: 5 hours. This makes sense because of the CSV size being 1,000,000.
- Test 9 I could not run successfully because the CSV size was too large (10,000,000), and I ultimately commented it out.
- Test 10 – 12 tested left join, right join, and outer join functionality.

Some test examples:

```
// test 3: simple join with attributes of different name ------------------------
vector<vector<string>> data5 = {{"ID", "Name"}, {"1", "Alice"}, {"2", "Bob"}, {"3", "Charlie"}};
vector<vector<string>> data6 = {{"Real_ID", "Age"}, {"1", "23"}, {"2", "34"}, {"3", "45"}};

//print
std::cout << "Table 5:\n";
printTable(data5);
std::cout << "Table 6:\n";
printTable(data6);
createCSV("table5.csv", data5);
createCSV("table6.csv", data6);
auto table5 = readCSV("table5.csv");
auto table6 = readCSV("table6.csv");
// should output empty table
innerJoin(table5, table6, "ID", "Real_ID", "output_test3.csv");

// print the joined data
auto joinedTable3 = readCSV("output_test3.csv");
```

```
ID,Name,Age,
1,Alice,23,
2,Bob,34,
3,Charlie,45,
```

Since there were common IDs but different column names, the column name of the left was taken.

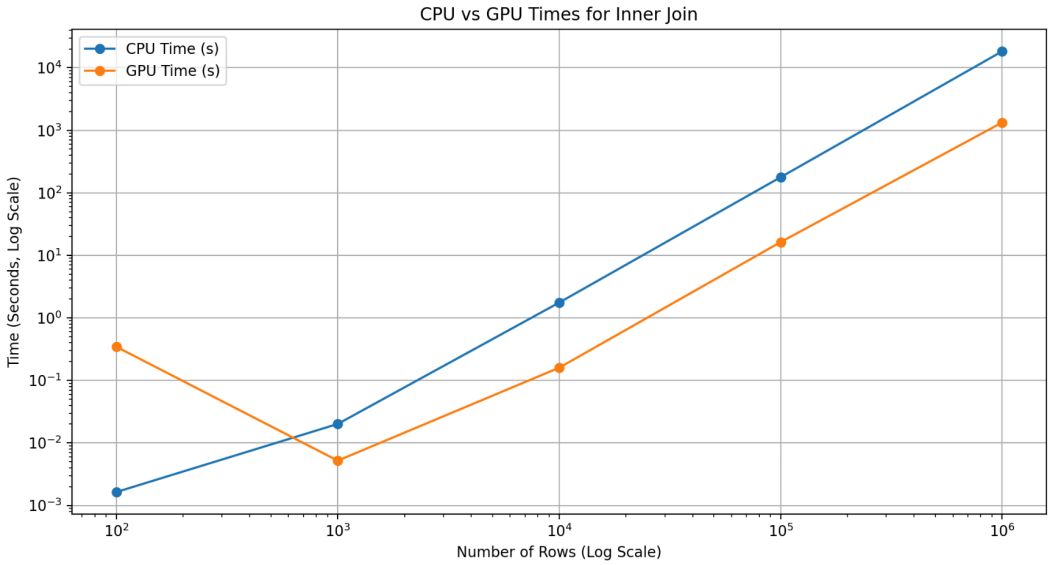
```
// test 5 --- join with CPU vs GPU size 1000 ------------------------
std::cout << "Test 5: Inner Join with CPU vs GPU size 1000\n";

// Read the massive CSV files into memory
table7 = readCSV("massive_tables/massive_table1000-1.csv");
table8 = readCSV("massive_tables/massive_table1000-2.csv");

// Time the CPU join
std::cout << "Massive Inner Join: beginning \n";

cpuTime = measureExecutionTime([&]() {
    innerJoin(table7, table8, "ID", "ID", "output_test5_cpu.csv");
});
std::cout << "CPU Inner Join Time: " << cpuTime << " seconds\n";

std::cout << "Massive Inner Join: finished \n";
// Time the CUDA join

std::cout << "Massive CUDA Inner Join: beginning \n";

cudaTime = measureExecutionTime([&]() {
    cudaInnerJoin(table7, table8, "ID", "ID", "output_test5_cuda.csv");
});
std::cout << "CUDA Inner Join Time: " << cudaTime << " seconds\n";

std::cout << "Massive CUDA Inner Join: finished \n";

// print GPU speed up factor
std::cout << "GPU Speedup Factor: " << cpuTime / cudaTime << "\n";
```

```
ID,Value,Value,
500,Value500,Value500,
501,Value501,Value501,
502,Value502,Value502,
503,Value503,Value503,
504,Value504,Value504,
505,Value505,Value505,
506,Value506,Value506,
507,Value507,Value507,
508,Value508,Value508,
509,Value509,Value509,
510,Value510,Value510,
511,Value511,Value511,
512,Value512,Value512,
513,Value513,Value513,
514,Value514,Value514,
515,Value515,Value515,
516,Value516,Value516,
517,Value517,Value517,
518,Value518,Value518,
519,Value519,Value519,
520,Value520,Value520,
521,Value521,Value521,
522,Value522,Value522,
523,Value523,Value523,
524,Value524,Value524,
525,Value525,Value525,
526,Value526,Value526,
527,Value527,Value527,
528,Value528,Value528,
```

Here, the output table only has entries from 500 to 1,000, which were the common IDs among the tables.

Results

The GPU successfully performed much faster than CPU on CSV table joins, especially for tables of larger size, but they did not scale as well as I thought they would.



Note that these graph axis scales are LOGARITHMIC, so even though there are linear curves on the graph, the performance difference between the CPU and GPU increases significantly with larger datasets.

The speed up increased as the CSV sizes increased, up to 13.76x speed up when we tested a CSV of size 1,000,000.

We can attribute this performance to the GPU's parallelism contributed to this performance, allowing it to handle larger datasets more effectively than the CPU. However, memory transfer overhead and suboptimal scaling due to batching limitations reduced the efficiency gains for smaller datasets. The GPU's superior performance can be attributed to efficient parallelization, shared memory optimizations, and coalesced memory access.

I believe that the GPU performance can be much improved through further kernel optimizations and a redesign of the memory management workflow. Perhaps the nature of writing to CSV files is so sequential that it becomes a bottleneck when parallelism could otherwise be leveraged, but I believe that through other optimization techniques and improving the kernel, optimizing batch size based on the GPU's memory characteristics, and such as asynchronous memory transfers (maybe through the use of streams), that the GPU performance can further be increased.

Challenges

The CUDA logic and figuring out the kernel creation was at first very challenging because understanding how to parallelize nested loops into thread blocks and grids while maintaining correct logic required significant experimentation. Specifically, ensuring threads processed non-overlapping segments of data and handling edge cases (such as rows without matches in left, right, or outer joins) added complexity.

Dealing with limited memory was also an issue. Batching helped, but balancing batch size to avoid underutilization or memory overflows took trial and error. At one point, larger batch sizes caused segmentation faults, especially with tables larger than 100,000 rows.

Lastly, the sheer size of the CSV tables and running O(n^2) algorithms on them was very time consuming, both for CPU and GPU. For example, while the GPU reduced the runtime significantly, the fundamental quadratic complexity of nested for-loops limited the scalability for datasets like 10,000,000 rows, which contributes to a TODO I would like to accomplish.

TODO

One todo is to increase GPU performance such that even larger table sizes work. Right now, I was unable to obtain success on the table of size 10,000,000, so I would like to optimize the GPU code to be able to join such data sizes. Perhaps improving the way that the join is conducted (instead of nested for loops, maybe trying block-based matrix multiplication logic or hash-based joins) would be optimal.

Another todo is modifying input to allow for any number of files on the join. This should not be too difficult, but I wonder how optimizable multiple tables is (should we think about parallelizing different sets of tables at once, like a divide and conquer approach?). Additionally, implementing advanced techniques like hierarchical joins (joining two tables, then merging the result with another table) could lead to more efficient computation. It is a complex issue and leaves much to research about.

Conclusion

While the results were promising, further optimizations are needed to handle larger datasets effectively and reduce memory transfer overhead. Additionally, supporting multi-table joins would enhance the system's capability and scalability. Despite these challenges, this project demonstrates the potential of GPU computing for large-scale data processing tasks and provides a foundation for future work in this domain.

Through this project, I learned much about CUDA kernel creation, memory management, and parallelism, and that these concepts (especially based on my past projects) are transferable to many other GPU-accelerated applications, such as this one, making this a valuable exploration of GPU computing's capabilities and limitations.