Parallel Computing

Big Assignment - Final Report

Map-Reduce

Team 20

Name	Sec	BN	ID	Email
بموا عريان عياد	1	17	9202391	bemoi.tawadros00@eng-st.cu.edu.eg
مارك ياسر نبيل	2	14	9203106	mark.ibrahim00@eng-st.cu.edu.eg
بيتر عاطف فتحي	1	18	9202395	peter.zaki00@eng-st.cu.edu.eg

Final system description

Map-Reduce

Map Kernel

The Map kernel applies a specified operation or function to each element of an input array in parallel. Utilizing the massively parallel architecture of GPUs, this kernel efficiently distributes the workload across numerous threads, enabling simultaneous computation on multiple data elements. Output is a <key, value> pair.

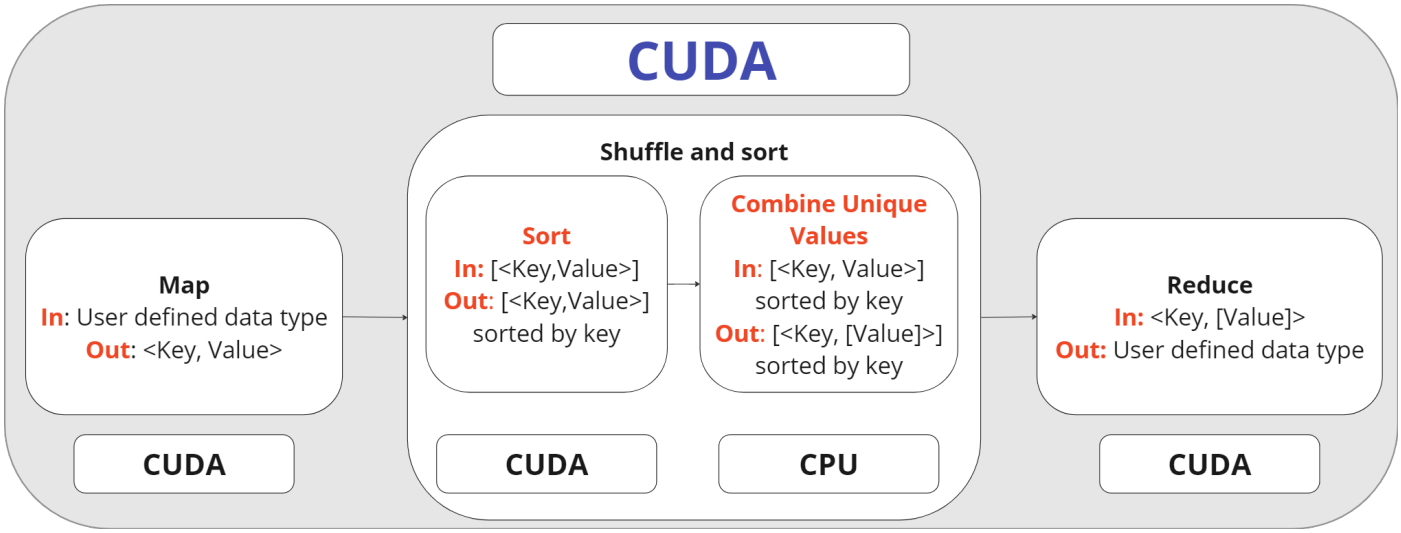
Shuffle and Sort Kernel

The Shuffle and Sort kernel arranges the processed data elements into a desired order, typically sorting them by key based on a specified criterion. By employing parallel sorting algorithms optimized for GPU architectures, this kernel organizes the data efficiently, facilitating subsequent analysis or retrieval operations. The output is <key,[values]>.

Reduce Kernel

The Reduce kernel aggregates the results obtained from the Map kernel by performing a reduction operation, such as summation or finding the maximum value. Through parallel reduction techniques, this kernel efficiently combines intermediate results to produce a single output value for each key.

Block Diagram



Implementation

Map and Reduce functions are user-defined, with a given signature for both functions. The user is also able to specify his data structures based on his problem definition.
A data file path is provided via command line arguments, the file is automatically read and then fed to the pipeline specified in the block diagram.

CPU

Developed a multithreaded implementation for the Map-Reduce algorithm on the CPU. Data is distributed based on the number of threads provided; each thread executes either the map or the reduce function on its assigned subset of data. The shuffle and sort phases are applied during the process.
All tests are conducted using a single thread for both mapping and reducing to assess the potential improvements offered by the GPU version.

GPU

Developing a CUDA implementation for Map-Reduce on GPU.
Each thread executes a **map** function for a given input.
The resulting output is then fed into a CUDA implementation of either **Merge Sort or Bitonic Sort**, based on user preference.
Subsequently, unique keys are consolidated on the CPU.
Each key is then routed to the **reducer**, with options for either:

- Applying a parallel reduction kernel for each key.
- Assigning each thread to process one output element.
  As per the user's choice.

We also utilized **streaming, shared memory, and pinned memory, with coalescing memory access and control divergence in mind.**

Configurable parameters

1. User-defined map function.
2. User-defined reduce function.
3. Map block size (grid size is calculated automatically such that each thread handles one output element).
4. Reduce block size (grid size is calculated automatically based on the next choice).
5. Whether to make each thread calculate one output element at reduce or use reduction techniques (which automatically calculates block size and grid size based on the input size, and sends the size of the shared memory array).
6. Expected number of output elements (with the option to be automatically inferred from the number of unique keys).
7. Number of iterations.
8. The user-defined data type for map key-value pair.
9. The user-defined data type for shuffling and sorting key-value pairs.

10. The user-defined data type for reduced output.
11. User-defined operators (<,>,<=, etc.) for each data type.
12. The option to prepopulate output values before starting the algorithm.

These parameters are provided via a config.cuh (ex:kmeans.cuh, wordcount.cuh) file and included in the main map_reduce.cu file.

Experiments and results

Problems Tested

K-Means

Data size = 2^15 points
Dimensions = 2
Iterations = 2
K = 50
MAP_BLOCK_SIZE = 512
MAP_GRID_SIZE = (NUM_INPUT + MAP_BLOCK_SIZE - 1) / MAP_BLOCK_SIZE = (2^15 + 512- 1) / 512 = 64
REDUCE_BLOCK_SIZE = 32
REDUCE_GRID_SIZE = (NUM_OUTPUT + REDUCE_BLOCK_SIZE - 1) / REDUCE_BLOCK_SIZE = (8 + 32- 1) / 32 =1

	CPU	CUDA: sort=merge, reducer=single thread for each output, Streaming =false.	CUDA: sort=bitonic, reducer=single thread for each output, Streaming =false.	CUDA: sort=merge, Reducer=single thread for each output, Streaming =true.	CUDA: sort=bitonic, reducer=single thread for each output, Streaming =true.
Map	487ms	677.78us	681.46us	7.9673ms	8.7559ms

Sort	29ms	152.60ms	5.6998ms	148.66ms	5.6942m
Shuffle	7ms	207ms	245ms	195ms	184ms
Reduce	4ms	2.0655ms	2.4135ms	2.0753ms	2.1023ms
Total without copy	533ms	362.34ms	253.8ms	353.695ms	200.55ms
Total with copy	-	858ms	817ms	912ms	783ms

Word Count

Data size = 2 ^ 12
MAP_BLOCK_SIZE = 512
MAP_GRID_SIZE = (NUM_INPUT + MAP_BLOCK_SIZE - 1) /
MAP_BLOCK_SIZE = (2^15 + 512- 1) / 512 = 64
REDUCE_BLOCK_SIZE = 256
REDUCE_GRID_SIZE = (shuffle_output_size +
REDUCE_BLOCK_SIZE * 2 - 1) / (REDUCE_BLOCK_SIZE * 2), is different depending on the the number of values for each unique key.

	CPU	CUDA: sort=merge, reducer=using reduction techniques, Streaming =false.	CUDA: sort=bitonic, reducer=using reduction techniques, Streaming =false.	CUDA: Rort=merge, Reducer=using reduction techniques, Streaming =true.	CUDA: sort=bitonic, reducer=using reduction techniques, Streaming =true.
Map	859us	5.2810us	5.1200us	80.100us	80.323us
Sort	1.656 ms	5.1697ms	241.06us	5.1679ms	236.14us
Shuffle	195us	2ms	1ms	2ms	1ms
Reduce	953us	19.427us	19.841us	19.361us	19.520us

Total without copy	3 ms	7.194ms	1.266ms	7.267ms	1.33566ms
Total with copy	-	577ms(check again)	364ms	511ms	437ms

Performance analysis

What are the CPU benchmarks for your application (theoretical ones)?

Map and Reduce

Since map and reduce functions are both user-defined, the complexity will depend on the problem itself, let's divide them based on our 2 problems:

K Means

Map

Calculate the distance between each point and each centroid, and pick the centroid with the shortest distance. O(N*M)
N = Dataset size, M = number of centroids

Reduce

Calculates the new centroid given the old centroid and the new points that belong to it. O(M*L)
M = number of centroids, L = Number of points assigned to the centroid

Word Count

Map

Emits key-value pair for each input, O(N)

Reduce

Sum values for each unique key, O(N*M).
N = Number of unique keys, M = number of values in each unique key.

Sort

Using C++ std sort with complexity O(NlogN)

Combine unique keys

Since the list is already sorted, we need only 1 pass on the list to combine unique keys. O(N)
N = Dataset size.

What about their GPU counterparts (based on your implementation)?

Map and Reduce

Since map and reduce functions are both user-defined, the complexity will depend on the problem itself, let's divide them based on our 2 problems:

K Means

Map

Calculate the distance between each point and each centroid and pick the centroid with the shortest distance. O(M)

M = number of centroids.
Since each thread handles a datapoint, we make O(M) computations in parallel.

Reduce

Calculates the new centroid given the old centroid and the new points that belong to it. O(L)
L = Number of points assigned to the centroid.
Since each thread handles a centroid, we make O(L) computations in parallel.

Word Count

Map

Emits key-value pair for each input, O(1)
Since all happen in parallel.

Reduce

Sum values for each unique key, O(N*(M-1)/LogM).
N = Number of unique keys, M = number of values in each unique key.
(M-1)/LogM is the complexity of the sum reduction technique used in lecture.

Sort

We implemented two sorting algorithms, the first is merge sort and the second one is bitonic sort.
It's noticed from the previous results that the bitonic sort is faster than the merge sort but the bitonic sort has a limitation: it works only on data sizes that are powers of two only.

Sort

Merge Sort

- The total number of threads is about N so the final complexity is O(Log(N)).

Bitonic Sort

- The total number of threads is about N so the final complexity is O(Log(N)^2).

How much is the speedup of the GPU over the CPU?

K Means

Map

O(N*M) -> O(M)
N=dataset size, M = number of centroids.
Speedup = N = 2^15
Ratio =(2^15*50-50)/2^15*50= 0.999

Reduce

O(M*L) -> O(L)
M = number of centroids, L = Number of points assigned to the centroid
Speedup = M = 50
Average L = dataset size / number of centroids = 2^15/50
Ratio =(50*2^15/50-2^15/50)/50*2^15/50= 0.98

Word Count

Map

O(N) -> O(1)
N = Dataset size.
Speedup = N = 2^12
Ratio = (2^12-1)/2^12 = 0.999

Reduce

O(N*M) -> O(N*(M-1)/LogM) = O(N*(M)/LogM)
N = Number of unique keys, M = number of values in each unique key.
After solving the problem, number of unique keys = 8
Average M = dataset size / 8 = 2^12 / 2^3 = 2^9
Speedup = LogM = Log2^9 = 2.71
Ratio = (8*2^9-8*2^9/Log2^9)/8*2^9 = 0.63

Sort

Merge Sort

- The total number of threads is about N so the final complexity is O(Log(N)).
- CPU: O(NLog(N))
- Speedup = CPU / GPU = N = 2^15
- Ratio = (NLog(N) - Log(N)) / NLog(N) = (N - 1) / N = 0.99

Bitonic Sort

- The total number of threads is about N so the final complexity is O(Log(N)^2).

- CPU: O(N(Log(N))^2)
- Speedup = CPU / GPU = N = 2^15
- Ratio = (NLog(N)^2 - Log(N)^2) / NLog(N)^2 = (N - 1) / N = 0.99

How does this compare to the theoretical speedup?

K Means

Map

Speedup = (487ms - 677.78us) / 487ms = 0.998
99.8% faster than CPU

Reduce

Speedup = (4ms - 2.0753ms) / 4ms = 0.4825
48.25% faster than CPU.

Sort

Merge Sort

Speedup = (29 - 152.60) / 29 = -4.262
426.2% slower than CPU

Bitonic Sort

Speedup = (29 - 5.6942) / 29 = 0.8
80% faster than CPU

Word Count

Map

Speedup = (859us - 5.1200us) / 859us = 0.994
99.4% faster than CPU

Reduce

Speedup = (953us - 19.841us) / 953us = 0.979
97.9% faster than CPU

Sort

Merge Sort

Speedup = (1.656ms - 5.1697ms) / 1.656ms = -2.1218
212% slower than CPU

Bitonic Sort

Speedup = (1.656ms - 241.06*10^-3) / 1.656ms = 0.854
85.4% faster than CPU

If your speedup is below the theoretical one (this is mostly the case), how do you explain this, what could be changed to achieve a better one?

Factors like memory access patterns, thread synchronization overhead, and algorithmic bottlenecks can limit the extent of speedup achieved. To optimize the speedup, several strategies can be explored, including fine-tuning memory access patterns, optimizing kernel configurations, and leveraging advanced GPU features like shared memory and tiling.

How do your GPU results compare to open-source peers of the same features, if any?

Comparison our sort with thrust::sort

K Means

Thrust Sort

DeviceMergeSortBlockSortKernel = 616.57us
DeviceMergeSortMergeKernel = 414.80us
DeviceMergeSortPartitionKernel = 106.92us
Total Time = 1.138ms

Bitonic Sort

Time = 5.6998ms
Speedup = (1.138ms - 5.6998ms)/1.138ms = -4.01
400% slower than thrust

Merge Sort

Time = 152.60ms
Speedup = (1.138ms - 152.60ms)/1.138ms = -133.09
13309% slower than thrust

Word Count

Thrust Sort

DeviceMergeSortBlockSortKernel = 102.44us
DeviceMergeSortMergeKernel = 35.074us
DeviceMergeSortPartitionKernel = 14.752us
Total Time = 152.266 µs

Bitonic Sort

Time = 241.06us
Speedup = (152.266 - 241.06)/152.266 = -0.583
58.3% slower than thrust

Merge Sort

Time = 5.1697ms
Speedup = (152.266 - 5.1697ms*10^3)/152.266 = -32.94
3294% slower than thrust

Comparison of our Map-Reduce with spark

Using PySpark running in pseudo-distributed mode:

K Means

Map

Time = 93.69850 µs
Speedup (ratio vs spark)= (93.69850 - 5.1200)/93.69850 = 0.945
Our map is 94.5% faster than Spark

Reduce

Time = 1823.235 ms
Speedup (ratio vs spark)= (1823.235 - 19.841*10^-3)/1823.235 = 0.999
Our reduce is 99.9% faster than Spark

Word Count

Map

Time = 1.56188 ms

Speedup (ratio vs spark)= (1.56188 - 5.1200*10^-3)/1.56188 = 0.967

Our map is 96.7% faster than Spark

Reduce

Time = 18.17346 ms

Speedup (ratio vs spark)= (18.17346 - 19.841*10^-3)/18.17346 = 0.998

Our reduce is 99.8% faster than Spark