Map-Reduce in CUDA

Team 20

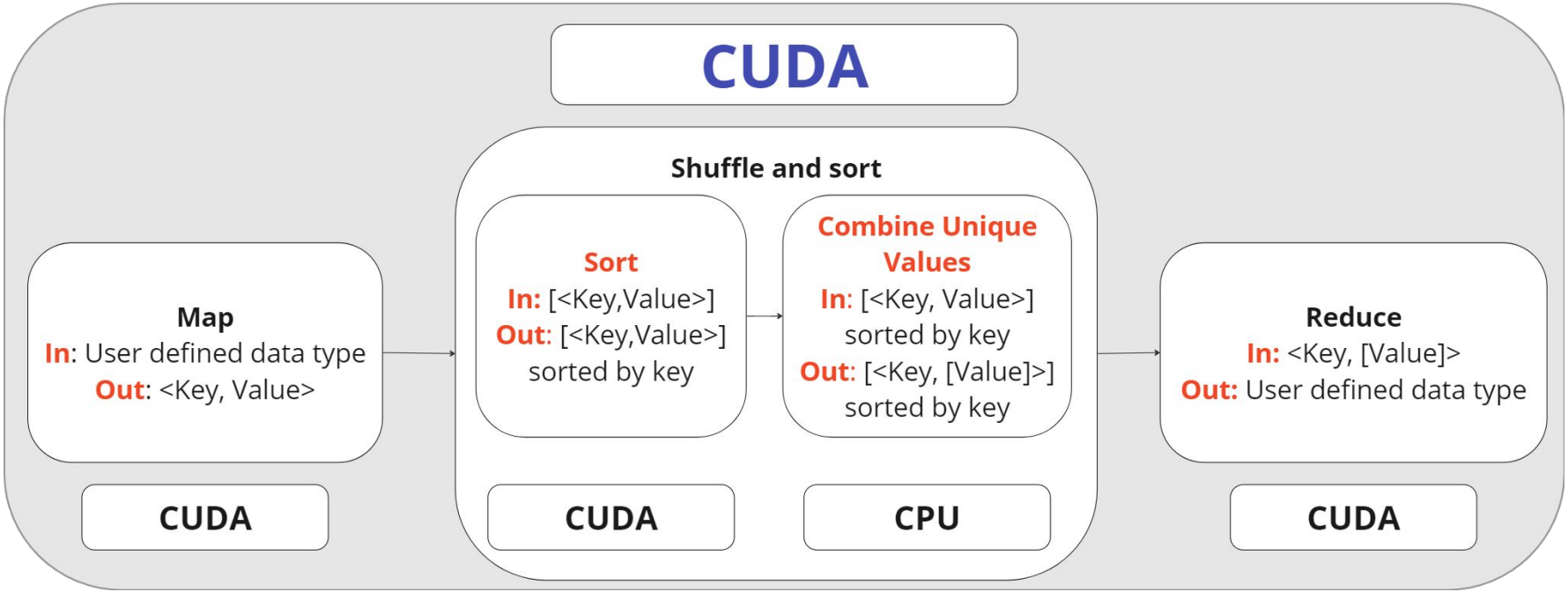
Name	Sec	BN	ID	Email
بموا عريان عياد	1	17	9202391	bemoi.tawadros00@eng-st.cu.edu.eg
مارك ياسر نبيل	2	14	9203106	mark.ibrahim00@eng-st.cu.edu.eg
بيتر عاطف فتحي	1	18	9202395	peter.zaki00@eng-st.cu.edu.eg

Implementation

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.

Block Diagram



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.
- Also, we used streaming with the map kernel to make use of the copying time.

Map Kernel: Calling the user defined device function

```
__global__ void mapKernel(const input_type *input, MyPair *pairs, output_type *dev_output, int *NUM_INPUT_D, int *NUM_OUTPUT_D)
{
    size_t threadId = blockIdx.x * blockDim.x + threadIdx.x; // Global id of the thread
    // // Total number of threads, by jumping this much, it ensures that no thread gets the same data
    if (threadId < *NUM_INPUT_D)
    {
        // Input data to run mapper on, and the starting index of memory assigned for key-value pairs for this
        mapper(&input[threadId], &pairs[threadId * NUM_PAIRS], dev_output, NUM_OUTPUT_D);
    }
}       bemoierian, 6 days ago • init
```

Map Kernel: Streaming

```
for (int s = 0; s < numberOfStreams; s++)
{
    int start = s * segment_size;
    int end = (start + segment_size) < NUM_INPUT ? (start + segment_size) : NUM_INPUT;
    int segment_input_size = end - start;
    std::cout << "Stream: " << s << " Start: " << start << " End: " << end << " Segment size: " << segment_input_size << std::endl;
    int *segment_input_size_d;
    if (iter == 0)
    {
        // Copy the size of the segment to device
        cudaMallocAsync(&segment_input_size_d, sizeof(int), streams[s]);
        cudaMemcpyAsync(segment_input_size_d, &segment_input_size, sizeof(int), cudaMemcpyHostToDevice, streams[s]);
    }
    cudaMemcpyAsync(&dev_input[start], &input[start], segment_input_size * sizeof(input_type), cudaMemcpyHostToDevice, streams[s]);

    // ================= MAP =================
    temp = runMapKernel(&dev_input[start], &dev_pairs[start], dev_output, segment_input_size_d, NUM_OUTPUT_D, streams[s]);
    // Print the time of the runs
    std::cout << "\n\nIteration:" << iter << " Stream:" << s << " Map function GPU Time: " << temp << " ms" << std::endl;
    mapGPUTime += temp;
    iterationTime += temp;
}
cudaDeviceSynchronize();
```

Sort Kernel

- Merge Sort
- Bitonic Sort

=> There is a limitation for our implementation that it works only for input of powers of 2 (16,32,64,...).
=> Bitonic Sort by nature works for only data sizes of powers of 2 only but it’s very efficient. Also, our implementation of merge sort works for only data size of powers of 2.
=>We will explain how we solved this problem in the following slide

Sort Kernel

- We solved the problem by checking if the size of the data is power of 2 using the following function: Ex: if num = 16 = 10000 => num - 1 = 15 = 1111 so num & (num -1) = 0
-
```
// Function to check if given number is a power of 2
bool isPowerOfTwo(int num)
{
    return num > 0 && (num & (num - 1)) == 0;
}
```
-
- If the size of the data isn't power of 2 we will get the next power of 2 using the following function:
- `nextPowerOfTwo = isPowerOfTwo(size) == false ? pow(2, ceil(log2(size))) : size;`
- We add the needed validations inside the kernels.

Merge Sort

In our implementation, we assumed that each kernel will sort (wid * 2) elements. So the first kernel will sort only consecutive elements. The second kernel will sort 4 consecutive elements. We did that to mimic the execution of recursion.

```
for (int wid = 1; wid < nextPowerOfTwo; wid *= 2) // O(log size)
{       You, 3 days ago • finish sorting
    mergeSortGPU<<<threadsPerBlock, blocksPerGrid>>>(gpuArrmerge, gpuTemp, size, wid * 2); // O(n)
}
```

Merge Sort

```
// GPU Kernel for Merge Sort
__global__ void mergeSortGPU(MyPair *arr, MyPair *temp, int n, int width)
{
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    // start index
    int left = tid * width;
    // middle index
    int middle = left + width / 2;
    // end index
    int right = left + width;
    if (left < n && middle < n)
    {
        mergeSequential(arr, temp, left, middle, right);
    }
}
```

Merge Sort

```
// Device function for recursive Merge
__device__ void mergeSequential(MyPair *arr, MyPair *temp, int left, int middle, int right, int n)
{
    int i = left;
    int j = middle;
    int k = left;                   // index for temp array
                                    // the first array is arr[left..middle-1]
                                    // the second array is arr[middle..right-1]
    right = right < n ? right : n; // to make sure that the right is not out of bounds      You, 1 se

    while (i < middle && j < right)
    {
        if (PairCompareLessEql()(arr[i], arr[j]))
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    while (i < middle)
        temp[k++] = arr[i++];
    while (j < right)
        temp[k++] = arr[j++];

    for (int x = left; x < right; x++)
        arr[x] = temp[x];
}
```

Bitonic Sort

One of the solutions to solve the problem of data size is to append the max value in the array to the original array till the size of the array become the next power of 2 but this solution waste time and memory so we will use bitonic sort as it is.

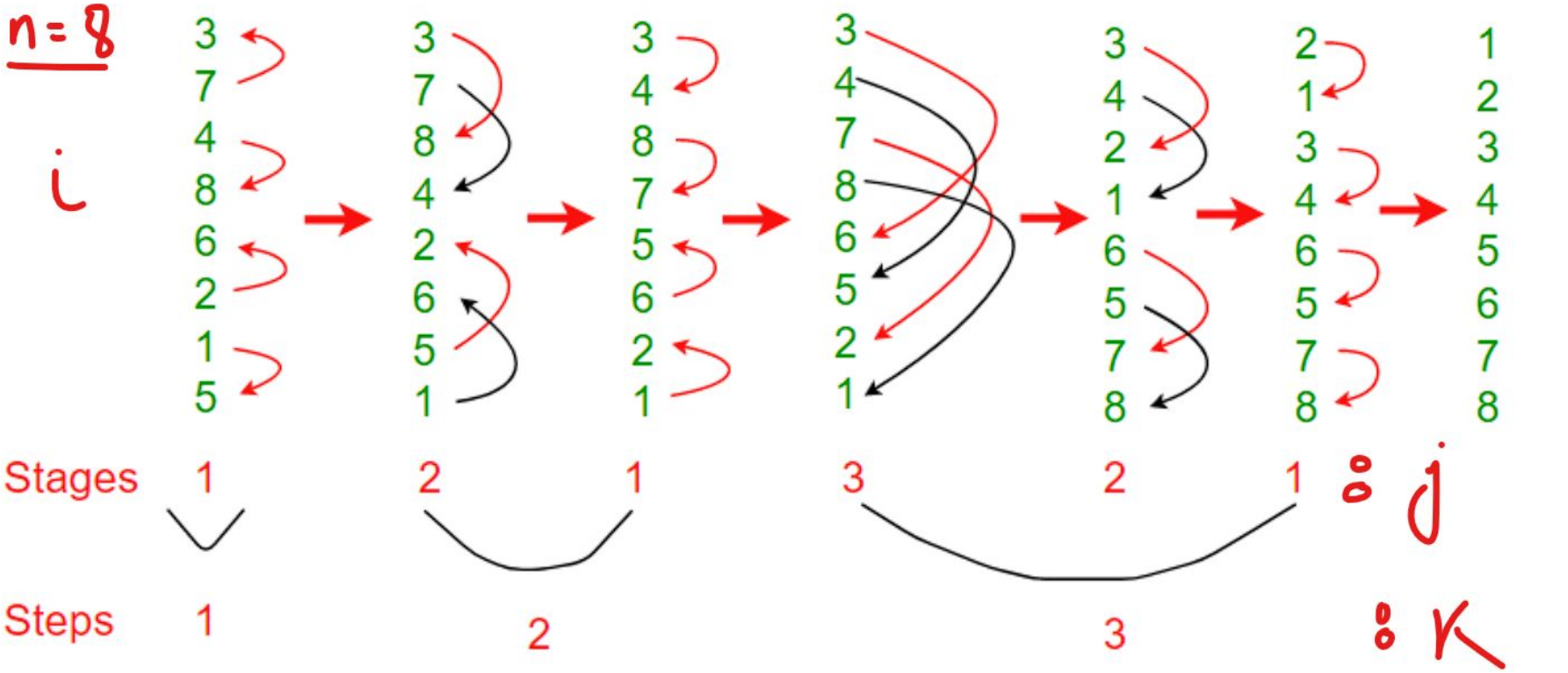
```
// O(log^2 n)
// The outer loop starts with k=2 and doubles each iteration
for (k = 2; k <= NUM_INPUT; k <<= 1) // O(log n)
{
    // The inner loop starts with j=k/2 and divides each iteration till it becomes 1
    for (j = k >> 1; j > 0; j = j >> 1) // O(log n)
    {
        bitonicSortGPU<<<blocksPerGrid, threadsPerBlock>>>(gpuArrbiton, j, k);
    }
}
```

Bitonic Sort

```
// GPU Kernel Implementation of Bitonic Sort
__global__ void bitonicSortGPU(MyPair *arr, int j, int k)
{
    unsigned int i, ij;

    i = threadIdx.x + blockDim.x * blockIdx.x;
    // Ensures that each pair is compared only once as if they are the same element the result will be 0;
    ij = i ^ j;
    // if the ij is greater than i, then the comparison is valid
    // 34an hakon babos 31a element ali ba3di
    if (ij > i)
    {
        // Determines the sorting direction for the current bitonic sequence.
        if ((i & k) == 0)
        {
            // if (i & k) == 0, then sort in ascending order
            // if greater than
            if (PairCompareGreater()(arr[i], arr[ij]))
            {
                MyPair temp = arr[i];
                arr[i] = arr[ij];
                arr[ij] = temp;
            }
        }
        else
        {
            // if (i & k) != 0, then sort in descending order       You, 1 second ago • Uncommitted changes
            // if less than
            if (PairCompare()(arr[i], arr[ij]))
            {
                MyPair temp = arr[i];
                arr[i] = arr[ij];
                arr[ij] = temp;
            }
        }
    }
}
```

Bitonic Sort



Combine Unique Keys

- Input: sorted Key-Value pairs (sorted on key) coming from Sort Kernel.
- Output: <key,[values]>, where key is unique.
- Complexity: Since the list is already sorted, we need only 1 pass on the list of key-value pairs to combine unique keys.

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.
- We implemented 2 modes of operation:
  1. Applying a parallel reduction kernel for each key.
  2. Assigning each thread to process one output element.

Reduce Kernel Mode 1:
Applying a parallel reduction kernel for each key.

1. Loop on each <key, [Values]>
2. Launch a kernel for each <key, [Values]>

```
for (int i = 0; i < output_size; i++)
{
    int shuffle_output_size = host_shuffle_output[i].size;
    REDUCE_BLOCK_SIZE = 256;
    REDUCE_GRID_SIZE = (shuffle_output_size + REDUCE_BLOCK_SIZE * 2 - 1) / (REDUCE_BLOCK_SIZE * 2);
    ShuffleAndSort_KeyPairOutput *dev_shuffle_output;
    cudaMalloc(&dev_shuffle_output, sizeof(ShuffleAndSort_KeyPairOutput));
    cudaMemcpy(dev_shuffle_output, &host_shuffle_output[i], sizeof(ShuffleAndSort_KeyPairOutput), cudaMemcpyHostToDevice);
    temp = runReduceKernel(dev_shuffle_output, &dev_output[i], TOTAL_PAIRS_D, NUM_OUTPUT_D);
    cudaFree(dev_shuffle_output);
}
```

Reduce Kernel Mode 1:
Applying a parallel reduction kernel for each key.

Kernel call with shared memory size:

```
    reduceKernel<<<REDUCE_GRID_SIZE, REDUCE_BLOCK_SIZE, 2 *
REDUCE_BLOCK_SIZE * sizeof(MyOutputValue)>>>(dev_pairs, dev_output,
TOTAL_PAIRS_D, NUM_OUTPUT_D);
```

Reduce Kernel Mode 2:
Assigning each thread to process one output element.

Call a single kernel, let each thread calculates one output element

```
ShuffleAndSort_KeyPairOutput *dev_shuffle_output;
cudaMalloc(&dev_shuffle_output, output_size * sizeof(ShuffleAndSort_KeyPairOutput));
cudaMemcpy(dev_shuffle_output, host_shuffle_output, output_size * sizeof(ShuffleAndSort_KeyPairOutput), cudaMemcpyHostToDevice);
temp = runReduceKernel(dev_shuffle_output, dev_output, TOTAL_PAIRS_D, NUM_OUTPUT_D);
cudaFree(dev_shuffle_output);
```

Problems Implemented

1. K-Means

- Reduction mode = 1
- 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

2. Word Count

- Reduction mode = 2
- Data size = 2 ^ 12
- 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.

Experiments and Results

1. K Means

	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

2. Word Count

	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	364ms	511ms	437ms

Performance Analysis

Theoretical

CPU vs GPU Complexity

Format: CPU ⇒ GPU

	Map	Reduce
K Means	O(N*M) ==> O(M) N=dataset size, M = number of centroids.	O(M*L) ==> O(L) M = number of centroids, L = Number of points assigned to the centroid.
Word Count	O(N) ==> O(1) N = Dataset size.	O(N*M) ==> O(N*(M-1)/LogM) N = Number of unique keys, M = number of values in each unique key.

Sort	
Merge Sort	Bitonic Sort
O(NLog(N)) ==> O(Log(N)) N=dataset size	O(N(Log(N))^2) ==> O(Log(N)^2) N=dataset size

CPU vs GPU Speed Up (CPU/GPU)

	Map	Reduce	Sort	
			Merge Sort	Bitonic Sort
K Means	N = 2^15	M = 50	N = 2^15	N = 2^15
Word Count	N = 2^12	LogM = Log2^9 = 2.71	N = 2^12	N = 2^12

CPU vs GPU Speed Up Ratio ((CPU-GPU)/CPU)

	Map	Reduce	Sort	
			Merge Sort	Bitonic Sort
K Means	0.999	0.98	0.99	0.99
Word Count	0.999	0.63	0.99	0.99

Based on our implementation

CPU vs GPU Speed Up Ratio ((CPU-GPU)/GPU)

	Map	Reduce	Sort	
			Merge	Bitonic
K Means	0.998	0.4825	-4.262	0.8
Word Count	0.994	0.979	-2.1218	0.854

Comparison with peers

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

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

Thank You