papergrid
Date: / /

Rank Sort

- Rank Sort is an algorithm for sorting a vector based on the rank of each element in the vector. The rank of an element represents how many numbers in the vector are smaller than it.

=> Steps:

- Count how many elements in the input vector are smaller than each each element for every element, and store this in another vector called 'rank'.
- Then the sorted vector would have it's indices given by the new vector rank.

⇓

=> Pseudocode Serial:

```
1  for   i=0  to  N
2      for  j=0  to  N
3          if  v[i] > v[j]
4              rank[i] ++
5
6  for  i=0  to  N
7      results[rank[i]] = v[i]
```

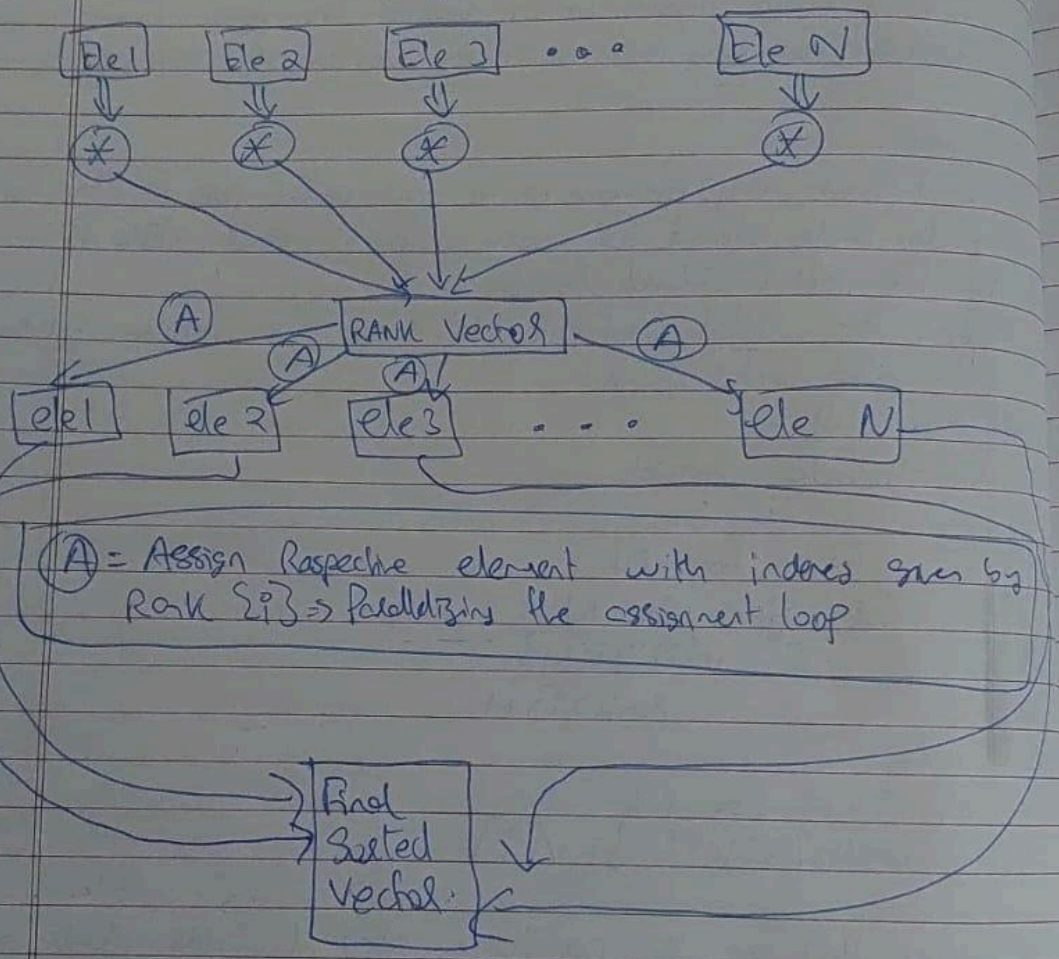
* Notice that we can parallelize step 1, as rank of each element can be calculated in parallel, meaning each thread will compute ranks for the elements within its interval.

* Also line 6 can be executed in parallel, as we are simply assigning values to the array predetermined from the rank array.

=> Hence, these are the optimizations that I could identify (5)

(*) = Run inner loop $j = 0$ to N

(a) Task dependency graph:



(A) = Assign respective element with indexes given by Rank $[i]$ => Parallelizing the assignment loop

Note that for simplicity I have assumed that the numbers present in the array are all unique.

Algorithm implemented using CUDA:

Compile the code using: *nvcc -o ranksort ranksort.cu*

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// CUDA runtime
#include <cuda_runtime.h>

// Kernel function to perform rank sort on GPU
__global__ void rankSortKernel(int *input, int *output, int N) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N) {
        int rank = 0;
        for (int j = 0; j < N; ++j) {
            if (input[tid] > input[j]) {
                rank++;
            }
        }
        output[rank] = input[tid];
    }
}

// Function to perform rank sort on CPU
void rankSortCPU(int *input, int *output, int N) {
    for (int i = 0; i < N; ++i) {
        int rank = 0;
        for (int j = 0; j < N; ++j) {
            if (input[i] > input[j]) {
                rank++;
            }
        }
        output[rank] = input[i];
    }
}
```

```c
// Function to shuffle an array using Fisher-Yates algorithm
void shuffleArray(int *array, int n) {
    srand(time(NULL));
    for (int i = n - 1; i > 0; --i) {
        int j = rand() % (i + 1);
        // Swap array[i] and array[j]
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

int main() {
    int N;
    printf("Enter size of array (N): ");
    scanf("%d", &N);

    // Generate unique random numbers
    int *h_A = (int *)malloc(N * sizeof(int));
    int *uniqueNumbers = (int *)malloc(N * sizeof(int));
    srand(time(NULL));
    int index = 0;

    while (index < N) {
        int randomNum = rand() % 100; // Generate random numbers between 0
and 99
        int isUnique = 1;
        // Check if randomNum is unique
        for (int i = 0; i < index; ++i) {
            if (uniqueNumbers[i] == randomNum) {
                isUnique = 0;
                break;
            }
        }
        if (isUnique) {
            uniqueNumbers[index++] = randomNum;
        }
    }

    // Shuffle the unique numbers
```

```
    shuffleArray(uniqueNumbers, N);

    // Copy shuffled unique numbers to h_A
    for (int i = 0; i < N; ++i) {
        h_A[i] = uniqueNumbers[i];
    }

    // Print shuffled array
    printf("Randomized & Shuffled Array:\n");
    for (int i = 0; i < N; ++i) {
        printf("%d ", h_A[i]);
    }
    printf("\n");

    // Allocate memory on GPU
    int *d_A, *d_B;
    cudaMalloc(&d_A, N * sizeof(int));
    cudaMalloc(&d_B, N * sizeof(int));

    // Copy input array from host to device
    cudaMemcpy(d_A, h_A, N * sizeof(int), cudaMemcpyHostToDevice);

    // Determine block and grid dimensions
    int T, B;
    printf("Enter number of threads per block (T): ");
    scanf("%d", &T);
    printf("Enter number of blocks in grid (B): ");
    scanf("%d", &B);

    // Ensure that T * B is at least N
    if (T * B < N) {
        printf("Error: Number of threads * blocks must be >= number of
elements in the array (N)\n");
        return 1;
    }

    // Launch GPU kernel and time GPU execution
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
```

```
    cudaEventRecord(start);

    rankSortKernel<<<B, T>>>(d_A, d_B, N);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float gpuTime = 0;
    cudaEventElapsedTime(&gpuTime, start, stop);

    // Copy sorted array from device to host
    int *h_B = (int *)malloc(N * sizeof(int));
    cudaMemcpy(h_B, d_B, N * sizeof(int), cudaMemcpyDeviceToHost);

    // Perform rank sort on CPU and time CPU execution
    int *h_C = (int *)malloc(N * sizeof(int));
    clock_t cpuStart, cpuEnd;
    cpuStart = clock();
    rankSortCPU(h_A, h_C, N);
    cpuEnd = clock();
    float cpuTime = ((float)(cpuEnd - cpuStart)) / CLOCKS_PER_SEC * 1000;

    // Display results and timing
    printf("GPU Sorted Array:\n");
    for (int i = 0; i < N; ++i) {
        printf("%d ", h_B[i]);
    }
    printf("\n");
    printf("CPU Sorted Array:\n");
    for (int i = 0; i < N; ++i) {
        printf("%d ", h_C[i]);
    }
    printf("\n");

    printf("GPU Execution Time: %.8f ms\n", gpuTime);
    printf("CPU Execution Time: %.8f ms\n", cpuTime);

    // Compute and display speed-up factor
    float speedUp = cpuTime / gpuTime;
    printf("Speed-Up Factor (CPU vs GPU): %.8f\n", speedUp);
```

```
    // Clean up
    cudaFree(d_A);
    cudaFree(d_B);
    free(h_A);
    free(h_B);
    free(h_C);
    free(uniqueNumbers);

    return 0;
}
```

Sample Output for CUDA:

Enter size of array (N): 10
Randomized & Shuffled Array:
3 99 33 63 85 27 52 74 8 58
Enter number of threads per block (T): 5
Enter number of blocks in grid (B): 2
GPU Sorted Array:
3 8 27 33 52 58 63 74 85 99
CPU Sorted Array:
3 8 27 33 52 58 63 74 85 99
GPU Execution Time: 0.00771200 ms
CPU Execution Time: 0.00900000 ms
Speed-Up Factor (CPU vs GPU): 1.16701245

Algorithm implemented using openMP for GPU:

Compile using: *gcc -o ranksort ranksort.c -fopenmp*

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

// Function to generate an array of unique random numbers
void generate_unique_random(int *arr, int N) {
    int i, j, is_unique;
    for (i = 0; i < N; i++) {
```

```
        do {
            arr[i] = rand() % (N * 10); // Generate random number
            is_unique = 1;
            // Check uniqueness with previous elements
            for (j = 0; j < i; j++) {
                if (arr[j] == arr[i]) {
                    is_unique = 0;
                    break;
                }
            }
        } while (!is_unique);
    }
}

// Function to perform rank sort serially
void rank_sort_serial(int *v, int *results, int N) {
    int i, j, rank[N];

    // Initialize ranks to 0
    for (i = 0; i < N; i++) {
        rank[i] = 0;
    }

    // Calculate ranks
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (v[i] > v[j]) {
                rank[i]++;
            }
        }
    }

    // Place elements in results array based on ranks
    for (i = 0; i < N; i++) {
        results[rank[i]] = v[i];
    }
}

// Function to perform rank sort in parallel using OpenMP for GPU
```

```
void rank_sort_parallel(int *v, int *results, int N, int num_threads, int
num_blocks) {
    int i, j, rank[N];

    // Initialize ranks to 0
    #pragma omp target teams distribute parallel for
num_threads(num_threads) map(tofrom:rank)
    for (i = 0; i < N; i++) {
        rank[i] = 0;
    }

    // Calculate ranks in parallel
    #pragma omp target teams distribute parallel for collapse(2)
num_threads(num_threads) map(to:v[:N], rank[:N])
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            if (v[i] > v[j]) {
                rank[i]++;
            }
        }
    }

    // Place elements in results array based on ranks in parallel
    #pragma omp target teams distribute parallel for
num_threads(num_threads) map(to:v[:N], rank[:N], results[:N])
    for (i = 0; i < N; i++) {
        results[rank[i]] = v[i];
    }
}

int main() {
    int N, num_threads, num_blocks;
    printf("Enter the number of elements (N): ");
    scanf("%d", &N);
    printf("Enter the number of threads: ");
    scanf("%d", &num_threads);
    printf("Enter the number of blocks: ");
    scanf("%d", &num_blocks);

    // Allocate memory for arrays
```

```
    int *v = (int *)malloc(N * sizeof(int));
    int *results_cpu = (int *)malloc(N * sizeof(int));
    int *results_gpu = (int *)malloc(N * sizeof(int));

    // Generate random unique numbers
    generate_unique_random(v, N);

    // Print original array
    printf("Original Array:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", v[i]);
    }
    printf("\n");

    // Perform rank sort serially (on CPU)
    rank_sort_serial(v, results_cpu, N);

    // Print sorted array on CPU
    printf("Sorted Array (CPU):\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", results_cpu[i]);
    }
    printf("\n");

    // Perform rank sort in parallel (on GPU)
    rank_sort_parallel(v, results_gpu, N, num_threads, num_blocks);

    // Print sorted array on GPU
    printf("Sorted Array (GPU):\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", results_gpu[i]);
    }
    printf("\n");

    // Calculate speedup
    double start_cpu = omp_get_wtime();
    rank_sort_serial(v, results_cpu, N);
    double end_cpu = omp_get_wtime();

    double start_gpu = omp_get_wtime();
```

```
    rank_sort_parallel(v, results_gpu, N, num_threads, num_blocks);
    double end_gpu = omp_get_wtime();

    double time_cpu = end_cpu - start_cpu;
    double time_gpu = end_gpu - start_gpu;
    double speedup = time_cpu / time_gpu;

    printf("CPU Time: %f seconds\n", time_cpu);
    printf("GPU Time: %f seconds\n", time_gpu);
    printf("Speedup (CPU vs. GPU): %.2fx\n", speedup);

    // Free allocated memory
    free(v);
    free(results_cpu);
    free(results_gpu);

    return 0;
}
```

Sample Output:

Enter the number of elements (N): 10
Enter the number of threads: 5
Enter the number of blocks: 2
Original Array:
83 86 77 15 93 35 92 49 21 62
Sorted Array (CPU):
15 21 35 49 62 77 83 86 92 93
Sorted Array (GPU):
15 21 35 49 62 77 83 86 92 93
CPU Time: 0.000000959 seconds
GPU Time: 0.000392308 seconds
Speedup (CPU vs. GPU): 0.002444508x