ECEC 413: Parallel Computer Architecture
Programming Assignment 7
CUDA: Counting Sort

Tri Pham (EE), Dinh Nguyen (CE), Manh Cuong Phi (CE)

Instructor: Prof. Naga Kandasamy

ECE Department

Drexel University

March 5, 2021

Contents

Contents……………………………………………………………….………………………2

Introduction…………………………………………………………………………………3

Overview…...…………………………………………………………………………. .3

Project Description

Implementation……………………………………………………………………..3-5

Performance Analysis………………………………………………………..6-8

Conclusion……………………………………………………...…………………8

Introduction

Overview

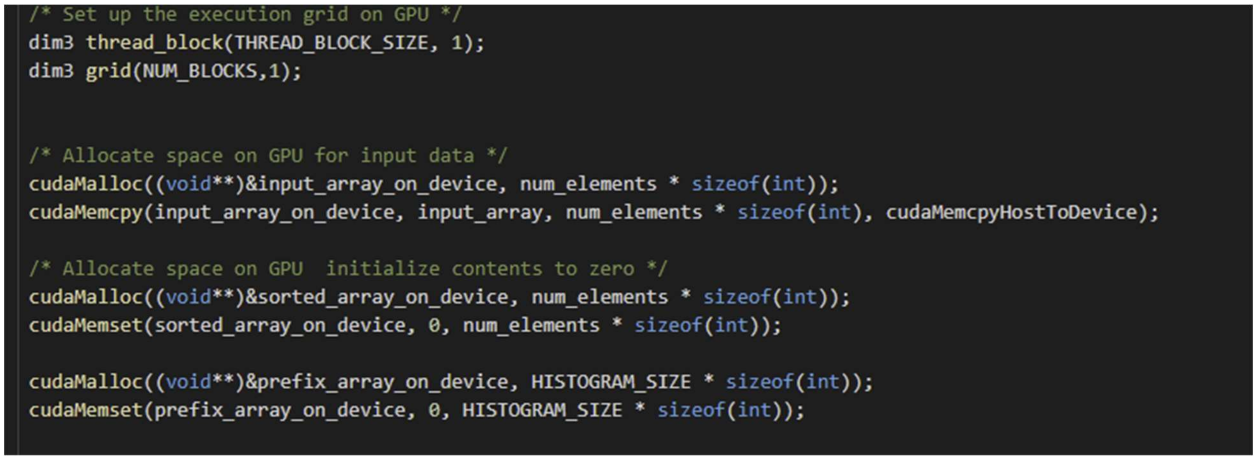
Counting sort is an efficient non-comparison based algorithm to sort an array of integers in the range 0 to r for some integer r. In this assignment we are going to implement the counting sort algorithm with a multithreading approach. We will be testing out the performance of multi-threading CUDA implementation on GPU to compare with the CPU performance.

Project Description

Implementation

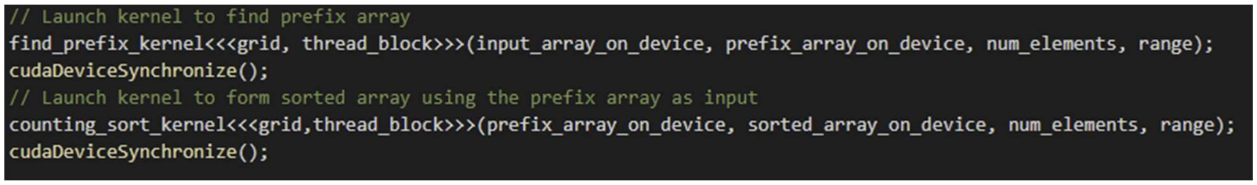
CUDA Deployment

We first start by setting up thread block, grid and allocate memory for sorted arrays according to the histogram size and number of elements given from the user.



```
/* Set up the execution grid on GPU */
dim3 thread_block(THREAD_BLOCK_SIZE, 1);
dim3 grid(NUM_BLOCKS,1);


/* Allocate space on GPU for input data */
cudaMalloc((void**)&input_array_on_device, num_elements * sizeof(int));
cudaMemcpy(input_array_on_device, input_array, num_elements * sizeof(int), cudaMemcpyHostToDevice);

/* Allocate space on GPU  initialize contents to zero */
cudaMalloc((void**)&sorted_array_on_device, num_elements * sizeof(int));
cudaMemset(sorted_array_on_device, 0, num_elements * sizeof(int));

cudaMalloc((void**)&prefix_array_on_device, HISTOGRAM_SIZE * sizeof(int));
cudaMemset(prefix_array_on_device, 0, HISTOGRAM_SIZE * sizeof(int));
```

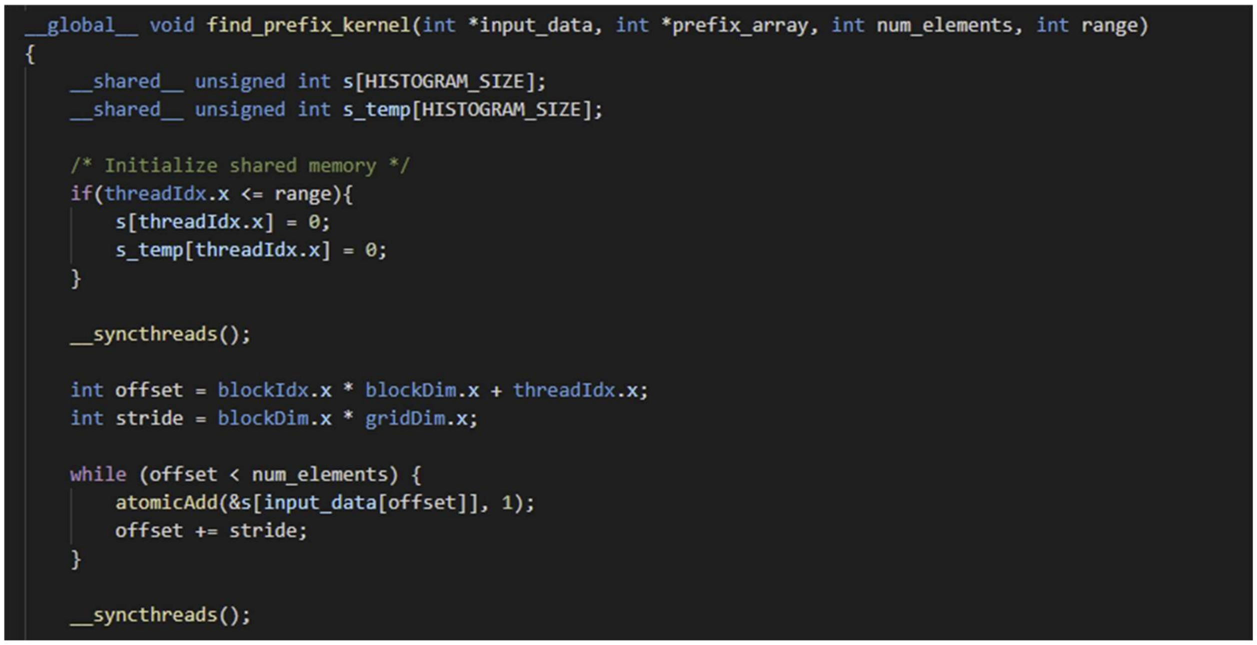
CUDA memory and array allocation code



```
// Launch kernel to find prefix array
find_prefix_kernel<<<grid, thread_block>>>(input_array_on_device, prefix_array_on_device, num_elements, range);
cudaDeviceSynchronize();
// Launch kernel to form sorted array using the prefix array as input
counting_sort_kernel<<<grid,thread_block>>>(prefix_array_on_device, sorted_array_on_device, num_elements, range);
cudaDeviceSynchronize();
```

Kernel execution code

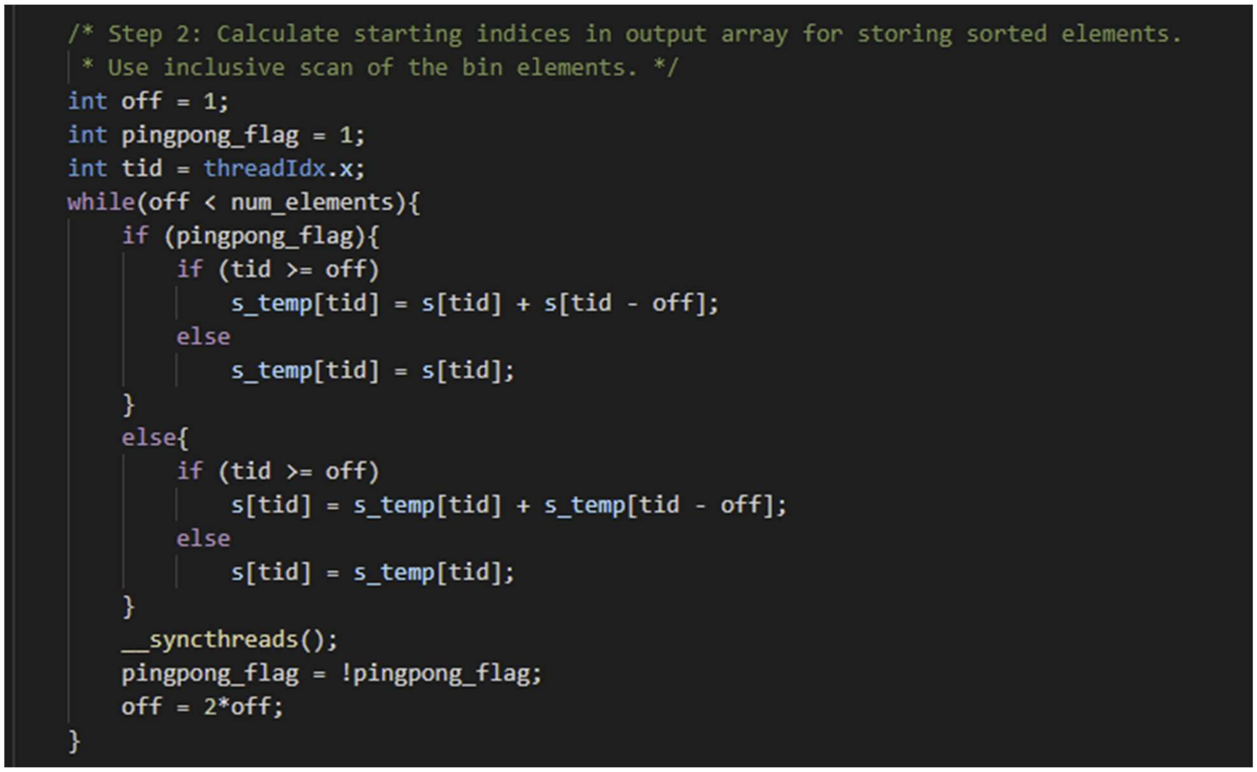
There are two kernels: one is used to execute the prefix scan including histogram generation and inclusive scan. The other one will perform the processing of the sorted array to be returned.The scanned value associated with each bin serves as an index into the output array to place the sorted elements.



```
__global__ void find_prefix_kernel(int *input_data, int *prefix_array, int num_elements, int range)
{
    __shared__ unsigned int s[HISTOGRAM_SIZE];
    __shared__ unsigned int s_temp[HISTOGRAM_SIZE];

    /* Initialize shared memory */
    if(threadIdx.x <= range){
        s[threadIdx.x] = 0;
        s_temp[threadIdx.x] = 0;
    }

    __syncthreads();

    int offset = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    while (offset < num_elements) {
        atomicAdd(&s[input_data[offset]], 1);
        offset += stride;
    }

    __syncthreads();
```

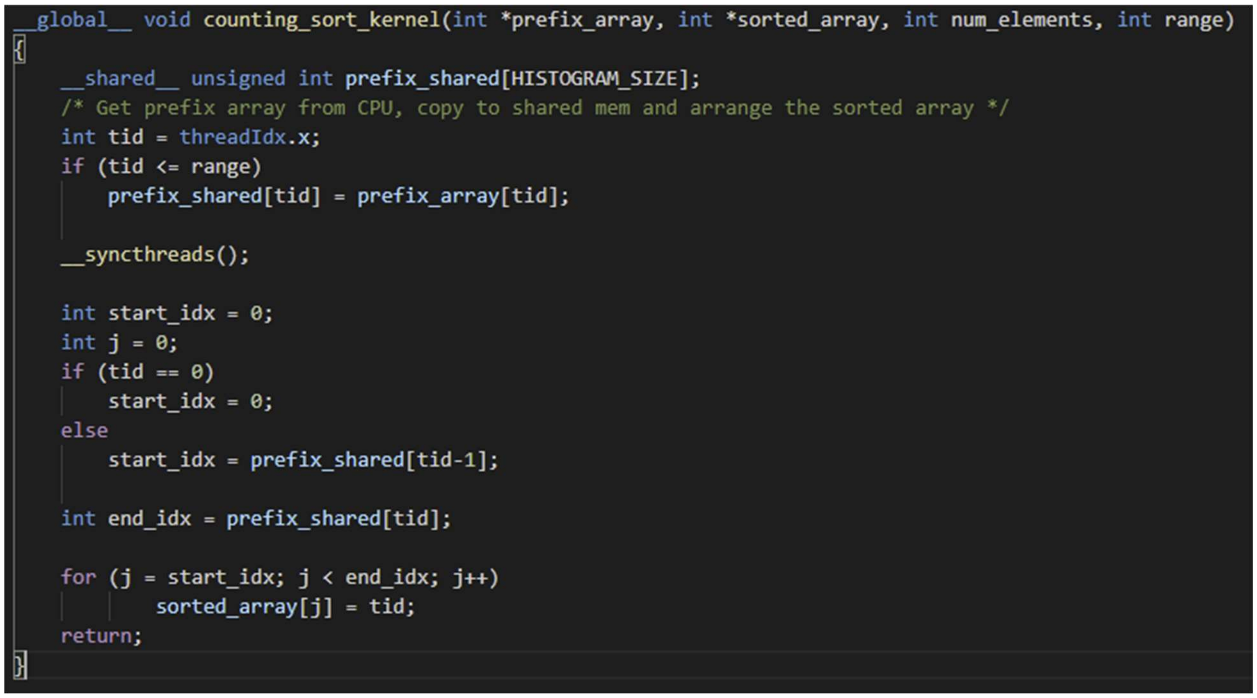
Prefix kernel code

To increase the overall performance, we use ping pong implementation to speed up by using two buffers. One store the element position of the previous counting sort iteration, and another to store new element indices for the current iteration.



```
/* Step 2: Calculate starting indices in output array for storing sorted elements.
 * Use inclusive scan of the bin elements. */
int off = 1;
int pingpong_flag = 1;
int tid = threadIdx.x;
while(off < num_elements){
    if (pingpong_flag){
        if (tid >= off)
            s_temp[tid] = s[tid] + s[tid - off];
        else
            s_temp[tid] = s[tid];
    }
    else{
        if (tid >= off)
            s[tid] = s_temp[tid] + s_temp[tid - off];
        else
            s[tid] = s_temp[tid];
    }
    __syncthreads();
    pingpong_flag = !pingpong_flag;
    off = 2*off;
}
```

Inclusive scan code

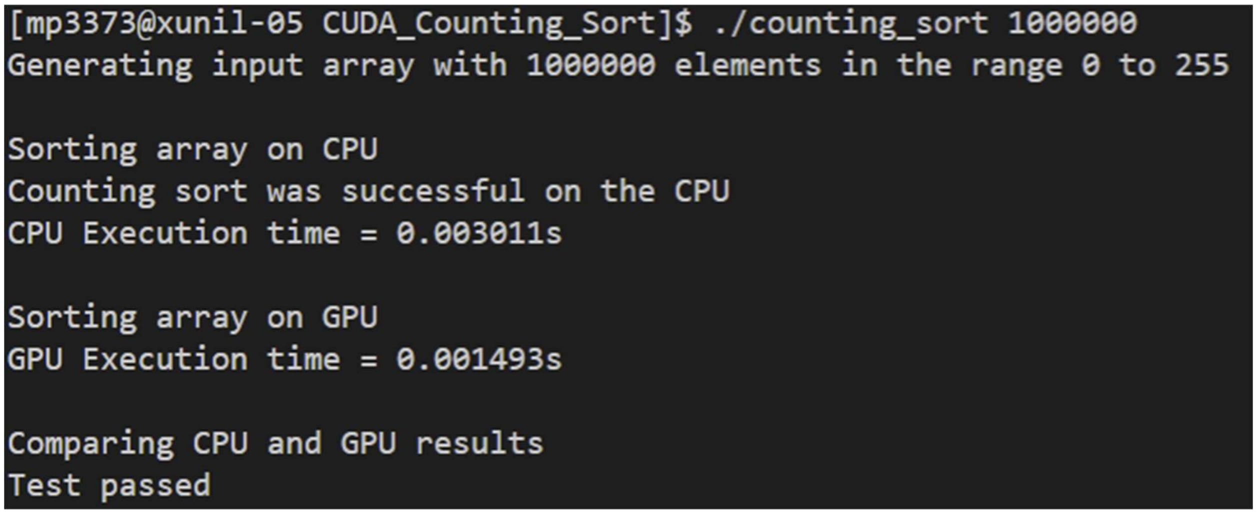


```
__global__ void counting_sort_kernel(int *prefix_array, int *sorted_array, int num_elements, int range)
{
    __shared__ unsigned int prefix_shared[HISTOGRAM_SIZE];
    /* Get prefix array from CPU, copy to shared mem and arrange the sorted array */
    int tid = threadIdx.x;
    if (tid <= range)
        prefix_shared[tid] = prefix_array[tid];

    __syncthreads();

    int start_idx = 0;
    int j = 0;
    if (tid == 0)
        start_idx = 0;
    else
        start_idx = prefix_shared[tid-1];

    int end_idx = prefix_shared[tid];

    for (j = start_idx; j < end_idx; j++)
            sorted_array[j] = tid;
    return;
}
```

Sorted array generation code

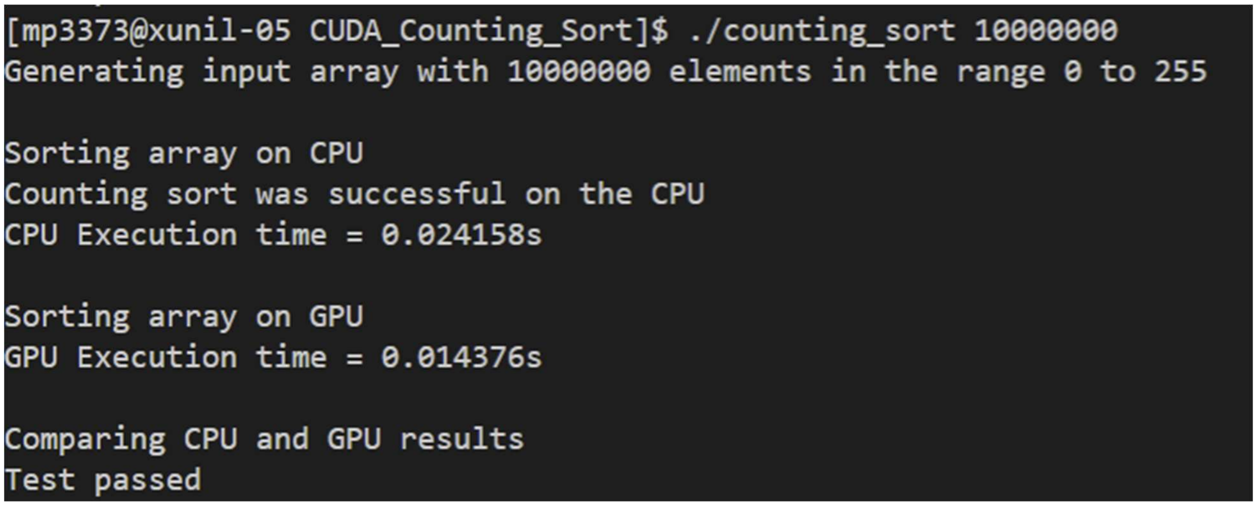
Performance Analysis

To observe the true performance of CUDA implementation on GPU vs CPU, we need to run the program with a large number of elements. The performance difference can be seen in the below figures as we tested each method with the number of elements 10^6, 10^7 and 10^8 elements. Below are the results for each case:



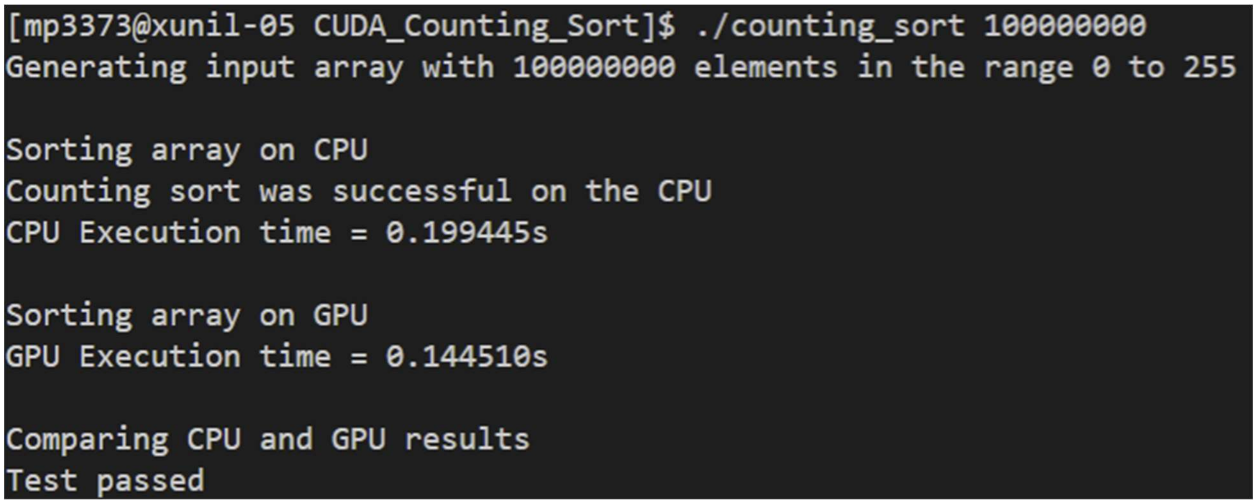
```
[mp3373@xunil-05 CUDA_Counting_Sort]$ ./counting_sort 1000000
Generating input array with 1000000 elements in the range 0 to 255

Sorting array on CPU
Counting sort was successful on the CPU
CPU Execution time = 0.003011s

Sorting array on GPU
GPU Execution time = 0.001493s

Comparing CPU and GPU results
Test passed
```

10^6 integers input array



```
[mp3373@xunil-05 CUDA_Counting_Sort]$ ./counting_sort 10000000
Generating input array with 10000000 elements in the range 0 to 255

Sorting array on CPU
Counting sort was successful on the CPU
CPU Execution time = 0.024158s

Sorting array on GPU
GPU Execution time = 0.014376s

Comparing CPU and GPU results
Test passed
```

10^7 integers input array



```
[mp3373@xunil-05 CUDA_Counting_Sort]$ ./counting_sort 100000000
Generating input array with 100000000 elements in the range 0 to 255

Sorting array on CPU
Counting sort was successful on the CPU
CPU Execution time = 0.199445s

Sorting array on GPU
GPU Execution time = 0.144510s

Comparing CPU and GPU results
Test passed
```

10^8 integers input array

Comparing the run time between the CPU and GPU, the performance increase over the serial version was very noticeable. For the input array of 10^6, the parallel version from the GPU was about 101% faster than the serial version on the CPU. As the input array grew larger, the performance on the GPU decreased noticeably. Taking a look at the 10^7 and 10^8 integers input arrays, the performance increase was reduced to 68% and 38 % respectively. This reduction in performance was due to the overhead which is the increase in communication delay between CPU/GPU. As the size of the array increased in respect to 10x time, the time taken to transfer the input and output arrays back and forth from the CPU would also increase.

Conclusion

We have successfully implemented the counting sort algorithm on the GPU. The two kernels were created for the implementation. One is used to execute the prefix scan including histogram generation and inclusive scan. The other is to perform the processing of the sorted array to be returned. To further increase the performance, the ping pong buffer was implemented using two buffers. One stores the element position of the previous counting sort iteration, the other stores the new element indices for the current iteration. With the above implementation, the performance has been improved drastically compared to the serial version on the CPU. Up to 100% improvement of input array of 10^6 elements. As the input array gets larger, the communication delay between CPU/GPU also increases. This decreased the overall performance on the GPU quite significantly, however, the run time was still about 30% to 60% faster than the CPU for larger data size.