Hadoop/Cuda Lab Assignment - 5
CSE 328

1 April 2022

Gyanendra Kumar Shukla
CSE 1
191112040

Department of Computer Science
NIT, Bhopal

Contents

1 Comparing Execution time of serial and parallel algorithms

Consider we have at least 20 elements in an integer array. Compare execution time for serial and parallel execution of the following sorting algorithms:

- Selection Sort
- Merge Sort
- Quick Sort
- Heap Sort

All the programs were compiled on nvidia gtx 1650 with smi information -

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 471.11       Driver Version: 471.11       CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name            TCC/WDDM | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ... WDDM  | 00000000:01:00.0  On |                  N/A |
| N/A   46C    P8     2W /  N/A |    872MiB /  4096MiB |      5%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

For compilation, use -

```
nvcc .\programname.cu -o outname -ccbin "C:\Program Files (x86)\Microsoft
Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30037\bin\Hostx64\x64"
```

1.1 Selection Sort

1.1.1 CODE

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

#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <curand_kernel.h>
#include <math_functions.h>

void _CheckCudaError(const cudaError_t cudaError, const char* file, const int line)
{
    if (cudaError != cudaSuccess) {
        std::cout << "[CUDA ERROR] " << cudaGetErrorString(cudaError) << " (" << file
        ↪  << ":" << line << ")\n";
        exit(EXIT_FAILURE);
    }
}
#define CheckCudaError(call) _CheckCudaError((call), __FILE__, __LINE__)

#define N       1024
#define blkSize 1024
#define grdSize 1

__global__ void init(unsigned long long seed, curandState_t* states)
{
    unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
    curand_init(seed, id, 0, &states[id]);
}

__global__ void build(int* d_data, curandState_t* states)
{
    unsigned int id = blockIdx.x * blockDim.x + threadIdx.x;
    d_data[id] = ceilf(curand_uniform(&states[id]) * N);
}

// Finds the maximum element within a warp and gives the maximum element to
// thread with lane id 0. Note that other elements do not get lost but their
// positions are shuffled.
__inline__ __device__ int warpMax(int data, unsigned int threadId)
{
    for (int mask = 16; mask > 0; mask /= 2) {
        int dual_data = __shfl_xor(data, mask, 32);
        if (threadId & mask)
            data = min(data, dual_data);
        else
            data = max(data, dual_data);
    }
    return data;
```

```
}

__global__ void selection32(int* d_data, int* d_data_sorted)
{
    unsigned int threadId = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int laneId = threadIdx.x % 32;

    int n = N;
    while(n-- > 0) {
        // get the maximum element among d_data and put it in d_data_sorted[n]
        int data = d_data[threadId];
        data = warpMax(data, threadId);
        d_data[threadId] = data;

        // now maximum element is in d_data[0]
        if (laneId == 0) {
            d_data_sorted[n] = d_data[0];
            d_data[0] = INT_MIN; // this element is ignored from now on
        }
    }
}

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selection_sort_seq(int* arr, int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}


int main()
{
    int* d_data;
    int* d_data_sorted;
    int* h_data;
```

```
int* h_data_sorted;
curandState_t* states;

// allocate host and device memory
CheckCudaError(cudaMalloc(&d_data, sizeof(int) * N));
CheckCudaError(cudaMalloc(&d_data_sorted, sizeof(int) * N));
h_data = (int*)malloc(sizeof(int) * N);
h_data_sorted = (int*)malloc(sizeof(int) * N);
CheckCudaError(cudaMalloc(&states, sizeof(curandState_t) * N));

// build random data
init<<<grdSize, blkSize>>>(time(0), states);
build<<<grdSize, blkSize>>>(d_data, states);

// print random data
CheckCudaError(cudaMemcpy(h_data, d_data, sizeof(int) * N,
↪  cudaMemcpyDeviceToHost));
printf("Testing the performance of selection sort on CUDA \nand comparing it with
↪  CPU running time\n");
printf("N: %d\n", N);

printf("Running Sequential Selection Sort on CPU\n");
printf("\n");

// sequential selection sort
int* d_data_seq = (int*)malloc(sizeof(int) * N);
memcpy(d_data_seq, h_data, sizeof(int) * N);

clock_t start, end;
double time_used;

start = clock();
selection_sort_seq(d_data_seq, N);
end = clock();

time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

printf("Sequential selection sort time: %f\n", time_used);


printf("\nRunning Selection Sort on GPU\n");
// parallel selection-sort
start = clock();
selection32<<<grdSize, blkSize>>>(d_data, d_data_sorted);
end = clock();

time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Parallel selection sort time: %f\n", time_used);
// print sorted data
CheckCudaError(cudaMemcpy(h_data_sorted, d_data_sorted, sizeof(int) * N,
↪  cudaMemcpyDeviceToHost));
```

```
    // free allocated memory
    CheckCudaError(cudaFree(d_data));
    CheckCudaError(cudaFree(d_data_sorted));
    free(h_data);
    free(h_data_sorted);
    CheckCudaError(cudaFree(states));

    return 0;
}
```

1.1.2 OUTPUT

```
Testing the performance of selection sort on CUDA
and comparing it with CPU running time
N: 1024
Running Sequential Selection Sort on CPU

Sequential selection sort time: 0.002000

Running Selection Sort on GPU
Parallel selection sort time: 0.000000
```

1.2 Merge Sort

1.2.1 CODE

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

// // // // // // // // // // // // // // // // // //
//  CPU Implementation                              //
// // // // // // // // // // // // // // // // // //
void merge(int *list, int *sorted, int start, int mid, int end)
{
    int ti=start, i=start, j=mid;
    while (i<mid || j<end)
    {
        if (j==end) sorted[ti] = list[i++];
        else if (i==mid) sorted[ti] = list[j++];
        else if (list[i]<list[j]) sorted[ti] = list[i++];
        else sorted[ti] = list[j++];
        ti++;
    }

    for (ti=start; ti<end; ti++)
        list[ti] = sorted[ti];
}

void mergesort_recur(int *list, int *sorted, int start, int end)
{
    if (end-start<2)
        return;

    mergesort_recur(list, sorted, start, start + (end-start)/2);
    mergesort_recur(list, sorted, start + (end-start)/2, end);
    merge(list, sorted, start, start + (end-start)/2, end);
}

int mergesort_cpu(int *list, int *sorted, int n)
{
    mergesort_recur(list, sorted, 0, n);
    return 1;
}

// // // // // // // // // // // // // // // // // //
//  GPU Implementation                              //
// // // // // // // // // // // // // // // // // //
__device__ void merge_gpu(int *list, int *sorted, int start, int mid, int end)
{
    int k=start, i=start, j=mid;
    while (i<mid || j<end)
    {
        if (j==end) sorted[k] = list[i++];
        else if (i==mid) sorted[k] = list[j++];
        else if (list[i]<list[j]) sorted[k] = list[i++];
```

```
        else sorted[k] = list[j++];
        k++;
    }
}

__global__ void mergesort_gpu(int *list, int *sorted, int n, int chunk){

    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int start = tid * chunk;
    if(start >= n) return;
    int mid, end;

    mid = min(start + chunk/2, n);
    end = min(start + chunk, n);
    merge_gpu(list, sorted, start, mid, end);
}

// Sequential Merge Sort for GPU when Number of Threads Required gets below 1 Warp
↪  Size
void mergesort_gpu_seq(int *list, int *sorted, int n, int chunk){
    int chunk_id;
    for(chunk_id=0; chunk_id*chunk<=n; chunk_id++){
        int start = chunk_id * chunk, end, mid;
        if(start >= n) return;
        mid = min(start + chunk/2, n);
        end = min(start + chunk, n);
        merge(list, sorted, start, mid, end);
    }
}


int mergesort(int *list, int *sorted, int n){

    int *list_d;
    int *sorted_d;
    int dummy;
    bool flag = false;
    bool sequential = false;

    int size = n * sizeof(int);

    cudaMalloc((void **)&list_d, size);
    cudaMalloc((void **)&sorted_d, size);

    cudaMemcpy(list_d, list, size, cudaMemcpyHostToDevice);
    cudaError_t err = cudaGetLastError();
    if(err!=cudaSuccess){
        printf("Error_2: %s\n", cudaGetErrorString(err));
        return -1;
    }

    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
```

```
// vaues for sm_35 compute capability
const int max_active_blocks_per_sm = 16;
const int max_active_warps_per_sm = 64;

int warp_size = prop.warpSize;
int max_grid_size = prop.maxGridSize[0];
int max_threads_per_block = prop.maxThreadsPerBlock;
int max_procs_count = prop.multiProcessorCount;

int max_active_blocks = max_active_blocks_per_sm * max_procs_count;
int max_active_warps = max_active_warps_per_sm * max_procs_count;

int chunk_size;
for(chunk_size=2; chunk_size<2*n; chunk_size*=2){
    int blocks_required=0, threads_per_block=0;
    int threads_required = (n%chunk_size==0) ? n/chunk_size : n/chunk_size+1;

    if (threads_required<=warp_size*3 && !sequential){
        sequential = true;
        if(flag) cudaMemcpy(list, sorted_d, size, cudaMemcpyDeviceToHost);
        else cudaMemcpy(list, list_d, size, cudaMemcpyDeviceToHost);
        err = cudaGetLastError();
        if(err!=cudaSuccess){
            printf("ERROR_4: %s\n", cudaGetErrorString(err));
            return -1;
        }
        cudaFree(list_d);
        cudaFree(sorted_d);
    }
    else if (threads_required<max_threads_per_block){
        threads_per_block = warp_size*4;
        dummy = threads_required/threads_per_block;
        blocks_required = (threads_required%threads_per_block==0) ? dummy :
        ↪  dummy+1;
    }
    else if(threads_required<max_active_blocks*warp_size*4){
        threads_per_block = max_threads_per_block/2;
        dummy = threads_required/threads_per_block;
        blocks_required = (threads_required%threads_per_block==0) ? dummy :
        ↪  dummy+1;
    }else{
        dummy = threads_required/max_active_blocks;
        // int estimated_threads_per_block = (dummy%warp_size==0) ? dummy :
        ↪  (dummy/warp_size + 1)*warp_size;
        int estimated_threads_per_block = (threads_required%max_active_blocks==0)
        ↪  ? dummy : dummy+1;
        if(estimated_threads_per_block > max_threads_per_block){
            threads_per_block = max_threads_per_block;
            dummy = threads_required/max_threads_per_block;
            blocks_required = (threads_required%max_threads_per_block==0) ? dummy
            ↪  : dummy+1;
        } else{
```

```
                threads_per_block = estimated_threads_per_block;
                blocks_required = max_active_blocks;
            }
        }

        if(blocks_required>=max_grid_size){
            printf("ERROR_2: Too many Blocks Required\n");
            return -1;
        }

        if(sequential){
            mergesort_gpu_seq(list, sorted, n, chunk_size);
        }else{
            if(flag) mergesort_gpu<<<blocks_required, threads_per_block>>>(sorted_d,
            ↪  list_d, n, chunk_size);
            else mergesort_gpu<<<blocks_required, threads_per_block>>>(list_d,
            ↪  sorted_d, n, chunk_size);
            cudaDeviceSynchronize();

            err = cudaGetLastError();
            if(err!=cudaSuccess){
                printf("ERROR_3: %s\n", cudaGetErrorString(err));
                return -1;
            }
            flag = !flag;
        }
    }
    return 0;
}

int main(int argc, char const *argv[]) {

    clock_t start, end;
    double time_used;
    int n_list[] = {10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000};
    int i, j;
    for(j=0; j<8; j++){
        printf("############ LENGTH OF LIST: %d ############\n", n_list[j]);
        int *sorted = (int *) malloc(n_list[j]*sizeof(int));
        int *list = (int *) malloc(n_list[j]*sizeof(int));
        int *sorted_s = (int *) malloc(n_list[j]*sizeof(int));
        int *list_s = (int *) malloc(n_list[j]*sizeof(int));
        for(i=0; i<n_list[j]; i++){
            list[i] = rand()%10000;
            list_s[i] = list[i];
        }
        start = clock();
        mergesort(list, sorted, n_list[j]);
        end = clock();
        time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("TIME TAKEN(Parallel GPU): %f\n", time_used);
```

```
        start = clock();
        mergesort_cpu(list_s, sorted_s, n_list[j]);
        end = clock();
        time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("TIME TAKEN(Sequential CPU): %f\n", time_used);

        for(i=1; i<n_list[j]; i++){
            if(sorted[i-1]>sorted[i]){
                printf("WRONG ANSWER _1\n");
                return -1;
            }
        }
        for(i=0; i<n_list[j]; i++){
            if(sorted_s[i]!=sorted[i]){
                printf("WRONG ANSWER _2\n");
                printf("P:%d, S:%d, Index:%d\n", sorted[i], sorted_s[i], i);
                return -1;
            }
        }
        printf("CORRECT ANSWER\n");

        free(sorted);
        free(list);
        free(sorted_s);
        free(list_s);
        printf("#####################################################\n");
    }
    return 0;
}
```

1.2.2 OUTPUT

```
############ LENGTH OF LIST: 10 ############
TIME TAKEN(Parallel GPU): 0.283000
TIME TAKEN(Sequential CPU): 0.000000
CORRECT ANSWER
#################################################
############ LENGTH OF LIST: 100 ############
TIME TAKEN(Parallel GPU): 0.001000
TIME TAKEN(Sequential CPU): 0.000000
CORRECT ANSWER
#################################################
############ LENGTH OF LIST: 1000 ############
TIME TAKEN(Parallel GPU): 0.001000
TIME TAKEN(Sequential CPU): 0.000000
CORRECT ANSWER
#################################################
############ LENGTH OF LIST: 10000 ############
TIME TAKEN(Parallel GPU): 0.002000
TIME TAKEN(Sequential CPU): 0.002000
CORRECT ANSWER
#################################################
```

```
############ LENGTH OF LIST: 100000 ############
TIME TAKEN(Parallel GPU): 0.013000
TIME TAKEN(Sequential CPU): 0.029000
CORRECT ANSWER
################################################
############ LENGTH OF LIST: 1000000 ############
TIME TAKEN(Parallel GPU): 0.118000
TIME TAKEN(Sequential CPU): 0.328000
CORRECT ANSWER
################################################
############ LENGTH OF LIST: 10000000 ############
TIME TAKEN(Parallel GPU): 1.090000
TIME TAKEN(Sequential CPU): 3.247000
CORRECT ANSWER
################################################
############ LENGTH OF LIST: 100000000 ############
TIME TAKEN(Parallel GPU): 12.129000
TIME TAKEN(Sequential CPU): 43.976000
CORRECT ANSWER
################################################
```

1.3 Quick Sort

1.3.1 CODE

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

 void printArr( int arr[], int n )
{
    int i;
    for ( i = 0; i < n; ++i )
        printf( "%d ", arr[i] );
}
__device__ int d_size;

__global__ void partition (int *arr, int *arr_l, int *arr_h, int n)
{
    int z = blockIdx.x*blockDim.x+threadIdx.x;
    d_size = 0;
    __syncthreads();
    if (z<n)
      {
        int h = arr_h[z];
        int l = arr_l[z];
        int x = arr[h];
        int i = (l - 1);
        int temp;
        for (int j = l; j <= h- 1; j++)
          {
            if (arr[j] <= x)
              {
                i++;
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
              }
          }
        temp = arr[i+1];
        arr[i+1] = arr[h];
        arr[h] = temp;
        int p = (i + 1);
        if (p-1 > l)
          {
            int ind = atomicAdd(&d_size, 1);
            arr_l[ind] = l;
            arr_h[ind] = p-1;
          }
        if ( p+1 < h )
          {
            int ind = atomicAdd(&d_size, 1);
            arr_l[ind] = p+1;
```

```
            arr_h[ind] = h;
          }
      }
}

void quickSortIterative (int arr[], int l, int h)
{
    int *lstack, *hstack;
    lstack = (int *)malloc(sizeof(int)*(h-l+1));
    hstack = (int *)malloc(sizeof(int)*(h-l+1));
    // int hstack[ h - l + 1];

    int top = -1, *d_d, *d_l, *d_h;

    lstack[ ++top ] = l;
    hstack[ top ] = h;

    cudaMalloc(&d_d, (h-l+1)*sizeof(int));
    cudaMemcpy(d_d, arr,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);

    cudaMalloc(&d_l, (h-l+1)*sizeof(int));
    cudaMemcpy(d_l, lstack,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);

    cudaMalloc(&d_h, (h-l+1)*sizeof(int));
    cudaMemcpy(d_h, hstack,(h-l+1)*sizeof(int),cudaMemcpyHostToDevice);
    int n_t = 1;
    int n_b = 1;
    int n_i = 1;
    while ( n_i > 0 )
    {
        partition<<<n_b,n_t>>>( d_d, d_l, d_h, n_i);
        int answer;
        cudaMemcpyFromSymbol(&answer, d_size, sizeof(int), 0, cudaMemcpyDeviceToHost);
        if (answer < 1024)
          {
            n_t = answer;
          }
        else
          {
            n_t = 1024;
            n_b = answer/n_t + (answer%n_t==0?0:1);
          }
        n_i = answer;
        cudaMemcpy(arr, d_d,(h-l+1)*sizeof(int),cudaMemcpyDeviceToHost);
    }
}


// A utility function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
```

```
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partitionSeq(int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element and indicates the right position of
    ↪  pivot found so far

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSortSeq(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partitionSeq(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSortSeq(arr, low, pi - 1);
        quickSortSeq(arr, pi + 1, high);
    }
}




int main()
{
    // int arr[5000];
    srand(time(NULL));
```

```
    // for (int i = 0; i<5000; i++)
    //    {
    //      arr[i] = rand ()%10000;
    //    }
    // int n = sizeof( arr ) / sizeof( *arr );
    // quickSortIterative( arr, 0, n - 1 );
    // printArr( arr, n );
    // return 0;

    clock_t start, end;
    double time_used;
    int n_list[] = {10, 100, 1000, 10000, 100000, 1000000};
    int i, j;
    for(j=0; j<6; j++){
      int *arr = (int *)malloc(sizeof(int)*n_list[j]);
      for (i = 0; i < n_list[j]; i++)
         {
           arr[i] = rand ()%10000;
         }
        printf("############ LENGTH OF LIST: %d ############\n", n_list[j]);
        start = clock();
        quickSortIterative( arr, 0, n_list[j] - 1 );
        end = clock();
        time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("TIME TAKEN(Parallel GPU): %f\n", time_used);


        start = clock();
        quickSortSeq( arr, 0, n_list[j] - 1 );
        end = clock();
        time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("TIME TAKEN(Sequential CPU): %f\n", time_used);

        printf("#####################################################\n");
    }

}
```

1.3.2 OUTPUT

```
############ LENGTH OF LIST: 10 ############
TIME TAKEN(Parallel GPU): 0.270000
TIME TAKEN(Sequential CPU): 0.000000
#################################################
############ LENGTH OF LIST: 100 ############
TIME TAKEN(Parallel GPU): 0.001000
TIME TAKEN(Sequential CPU): 0.000000
#################################################
############ LENGTH OF LIST: 1000 ############
TIME TAKEN(Parallel GPU): 0.003000
TIME TAKEN(Sequential CPU): 0.003000
#################################################
```

```
############ LENGTH OF LIST: 10000 ############
TIME TAKEN(Parallel GPU): 0.010000
TIME TAKEN(Sequential CPU): 0.212000
##################################################
############ LENGTH OF LIST: 100000 ############
TIME TAKEN(Parallel GPU): 0.089000
TIME TAKEN(Sequential CPU): 3.940000
##################################################
############ LENGTH OF LIST: 1000000 ############
TIME TAKEN(Parallel GPU): 0.914000
TIME TAKEN(Sequential CPU): 39.063000
##################################################
```

1.4 Heap Sort

1.4.1 CODE

```
#include <stdio.h>
#include <cuda_runtime.h>
#include <iostream>
#include <math.h>

static const int blockSize = 2047; // array size
static const int iterations = 10;  // number of iterations
int countBlocks = 2500;

inline void cudaCheck(const cudaError_t &err, const std::string &mes)
{
    if (err != cudaSuccess)
    {
        std::cout << (mes + " - " + cudaGetErrorString(err)) << std::endl;
        exit(EXIT_FAILURE);
    }
}

__device__ void swap(int *a, int *b)
{
    const int t = *a;
    *a = *b;
    *b = t;
}

__device__ void maxHeapify(int *maxHeap, int heapSize, int idx)
{
    int largest = idx;          // Initialize largest as root
    int left = (idx << 1) + 1;  // left = 2*idx + 1
    int right = (idx + 1) << 1; // right = 2*idx + 2

    // See if left child of root exists and is greater than root
    if (left < heapSize && maxHeap[left] > maxHeap[largest])
    {
        largest = left;
    }

    // See if right child of root exists and is greater than
    // the largest so far
    if (right < heapSize && maxHeap[right] > maxHeap[largest])
    {
        largest = right;
    }

    // Change root, if needed
    if (largest != idx)
    {
        swap(&maxHeap[largest], &maxHeap[idx]);
        maxHeapify(maxHeap, heapSize, largest);
```

```
    }
}

// A utility function to create a max heap of given capacity
__device__ void createAndBuildHeap(int *array, int size)
{
    // Start from bottommost and rightmost internal mode and heapify all
    // internal modes in bottom up way
    for (int i = (size - 2) / 2; i >= 0; --i)
    {
        maxHeapify(array, size, i);
    }
}

__global__ void heapSortKernel(int *iA, int size)
{
    // A = A + blockIdx.x * blockSize;
    iA = iA + blockIdx.x * blockSize;
    __shared__ int A[blockSize];
    for (int i = threadIdx.x; i < blockSize; i += blockDim.x)
    {
        A[i] = iA[i];
    }
    __syncthreads();
    // int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (threadIdx.x == 0)
    {
        // Build a heap from the input data.
        createAndBuildHeap(A, size);

        // Repeat following steps while heap size is greater than 1.
        // The last element in max heap will be the minimum element
        int changedSizeOfHeap = size;
        while (changedSizeOfHeap > 1)
        {
            // The largest item in Heap is stored at the root. Replace
            // it with the last item of the heap followed by reducing the
            // size of heap by 1.
            swap(A, &A[changedSizeOfHeap - 1]);
            --changedSizeOfHeap; // Reduce heap size

            // Finally, heapify the root of tree.
            maxHeapify(A, changedSizeOfHeap, 0);
        }
    }
    for (int i = threadIdx.x; i < blockSize; i += blockDim.x)
    {
        iA[i] = A[i];
    }
}

void heapifySeq(int arr[], int n, int i)
{
```

```
    int largest = i;   // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        std::swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapifySeq(arr, n, largest);
    }
}

// main function to do heap sort
void heapSortSeq(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapifySeq(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--)
    {
        // Move current root to end
        std::swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapifySeq(arr, i, 0);
    }
}

int main(int argc, char *argv[])
{
    cudaError_t err = cudaSuccess;
    // Print the vector length to be used, and compute its size
    int numElements = blockSize * countBlocks;
    size_t size = numElements * sizeof(int);

    // Allocate the host input vector A
    int *h_A = (int *)malloc(size);
    int *h_A_seq = (int *)malloc(size);
    if (h_A == NULL)
    {
        std::cout << "Failed to allocate host vectors!" << std::endl;
```

```
    exit(EXIT_FAILURE);
}
// Allocate the device input vector A
int *d_A = NULL;
err = cudaMalloc((void **)&d_A, size);
cudaCheck(err, "failed to allocated device vector A");

// Time timer;
clock_t start, end;
// timer.begin("common");
// Initialize the host input vectors
for (int i = 0; i < numElements; ++i)
{
    // h_A[i] = rand()/(int)RAND_MAX;
    h_A[i] = rand() % 1000 + 1;
    h_A_seq[i] = h_A[i];
}

err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaCheck(err, "failed to copy vector A to device");

std::cout << "Testing the performance of heap sort on CUDA \n and "
          << "comparing it with CPU running time" << std::endl;

std::cout << "N = " << numElements << std::endl;

std::cout << "Running on CPU\n";
start = clock();
heapSortSeq(h_A_seq, numElements);
end = clock();
double time_taken = double(end - start) / CLOCKS_PER_SEC;
std::cout << "CPU time taken = " << time_taken << " sec" << std::endl;

// Launch the CUDA Kernel
int threadsPerBlock = 32;
int blocksPerGrid = countBlocks;
// timer.begin("sort");
std::cout << "Running on GPU\n";
start = clock();
heapSortKernel<<<blocksPerGrid, threadsPerBlock>>>(d_A, blockSize);
end = clock();
time_taken = double(end - start) / CLOCKS_PER_SEC;
std::cout << "GPU time taken = " << time_taken << " sec" << std::endl;

cudaDeviceSynchronize();
// timer.end("sort");
err = cudaGetLastError();
cudaCheck(err, "failed to launch kernel");

err = cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);
cudaCheck(err, "failed to copy vector A to host");

int running_time = ((int)(end - start)) / CLOCKS_PER_SEC;
```

```
    std::cout << "running time = " << running_time << std::endl;

    // Free device global memory
    err = cudaFree(d_A);
    cudaCheck(err, "failed to free device vector A");
    // Free host memory
    free(h_A);
    err = cudaDeviceReset();
    cudaCheck(err, "failed to deinitialize the device");

    std::cout << "Done." << std::endl;
    return 0;
}
```

1.4.2 OUTPUT

```
Testing the performance of heap sort on CUDA
 and comparing it with CPU running time
N = 5117500
Running on CPU
CPU time taken = 3.204 sec
Running on GPU
GPU time taken = 0 sec
running time = 0
```