Assignment 4

Q1

#include <stdio.h>

#include <cuda.h>

#define N 1024

\_\_global\_\_ void differentTasksKernel(int \*output) {

int tid = threadIdx.x;

if (tid == 0) {

int sum = 0;

for (int i = 1; i <= N; i++) {

sum += i;

}

output[0] = sum;

}

if (tid == 1) {

// Direct formula

output[1] = N \* (N + 1) / 2;

}

}

int main() {

int \*d\_output, h\_output[2];

cudaMalloc((void \*\*)&d\_output, 2 \* sizeof(int));

differentTasksKernel<<<1, 2>>>(d\_output);

cudaMemcpy(h\_output, d\_output, 2 \* sizeof(int), cudaMemcpyDeviceToHost);

printf("Iterative Sum (Thread 0): %d\n", h\_output[0]);

printf("Formula Sum (Thread 1): %d\n", h\_output[1]);

cudaFree(d\_output);

return 0;

}

Q2

A) #include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 1000

void merge(int arr[], int l, int m, int r) {

int i = l, j = m + 1, k = 0;

int \*temp = (int \*)malloc((r - l + 1) \* sizeof(int));

while (i <= m && j <= r) {

if (arr[i] < arr[j]) temp[k++] = arr[i++];

else temp[k++] = arr[j++];

}

while (i <= m) temp[k++] = arr[i++];

while (j <= r) temp[k++] = arr[j++];

for (i = l, k = 0; i <= r; i++, k++) arr[i] = temp[k];

free(temp);

}

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = (l + r) / 2;

#pragma omp parallel sections

{

#pragma omp section

mergeSort(arr, l, m);

#pragma omp section

mergeSort(arr, m + 1, r);

}

merge(arr, l, m, r);

}

}

int main() {

int arr[N];

for (int i = 0; i < N; i++) arr[i] = rand() % 1000;

double start = omp\_get\_wtime();

mergeSort(arr, 0, N - 1);

double end = omp\_get\_wtime();

printf("Pipelined CPU Merge Sort Time: %f seconds\n", end - start);

return 0;

}

B) #include <stdio.h>

#include <cuda.h>

#define N 1000

\_\_global\_\_ void mergeSortKernel(int \*arr) {

int tid = threadIdx.x;

for (int i = 0; i < N - 1; i++) {

for (int j = tid; j < N - i - 1; j += blockDim.x) {

if (arr[j] > arr[j + 1]) {

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

\_\_syncthreads(); }

}

int main() {

int h\_arr[N], \*d\_arr;

for (int i = 0; i < N; i++) h\_arr[i] = rand() % 1000;

cudaMalloc((void \*\*)&d\_arr, N \* sizeof(int));

cudaMemcpy(d\_arr, h\_arr, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaEvent\_t start, stop;

cudaEventCreate(&start); cudaEventCreate(&stop);

cudaEventRecord(start);

mergeSortKernel<<<1, 256>>>(d\_arr);

cudaDeviceSynchronize();

cudaEventRecord(stop); cudaEventSynchronize(stop);

float ms;

cudaEventElapsedTime(&ms, start, stop);

cudaMemcpy(h\_arr, d\_arr, N \* sizeof(int), cudaMemcpyDeviceToHost);

printf("CUDA Merge Sort Time: %f ms\n", ms);

cudaFree(d\_arr);

return 0;

}

c) #include <stdio.h>

#include <stdlib.h>

#include <cuda.h>

#include <time.h>

#define N 1024 // Must be a power of 2

\_\_global\_\_ void bitonicSortKernel(int \*arr, int j, int k) {

unsigned int idx = threadIdx.x + blockDim.x \* blockIdx.x;

unsigned int ixj = idx ^ j;

if (ixj > idx) {

if ((idx & k) == 0) {

if (arr[idx] > arr[ixj]) {

int temp = arr[idx];

arr[idx] = arr[ixj];

arr[ixj] = temp;

}

} else {

if (arr[idx] < arr[ixj]) {

int temp = arr[idx];

arr[idx] = arr[ixj];

arr[ixj] = temp;

}

}

}

}

void printArray(int \*arr) {

for (int i = 0; i < N; i++) printf("%d ", arr[i]);

printf("\n");

}

int main() {

int h\_arr[N], \*d\_arr;

srand(time(NULL));

for (int i = 0; i < N; i++) {

h\_arr[i] = rand() % 1000;

}

printf("Unsorted Array:\n");

printArray(h\_arr);

cudaMalloc((void \*\*)&d\_arr, N \* sizeof(int));

cudaMemcpy(d\_arr, h\_arr, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

dim3 blocks(N / 256);

dim3 threads(256);

for (int k = 2; k <= N; k \*= 2) {

for (int j = k / 2; j > 0; j /= 2) {

bitonicSortKernel<<<blocks, threads>>>(d\_arr, j, k);

cudaDeviceSynchronize();

}

}

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float ms = 0;

cudaEventElapsedTime(&ms, start, stop);

cudaMemcpy(h\_arr, d\_arr, N \* sizeof(int), cudaMemcpyDeviceToHost);

printf("\nSorted Array (CUDA Bitonic Sort):\n");

printArray(h\_arr);

printf("CUDA Bitonic Sort Time: %.5f ms\n", ms);

cudaFree(d\_arr);

return 0;

}

Assignment 5

#include <stdio.h>

#include <cuda\_runtime.h>

#define N 1024

\_\_device\_\_ \_\_constant\_\_ float d\_A[N];

\_\_device\_\_ \_\_constant\_\_ float d\_B[N];

\_\_device\_\_ float d\_C[N];

\_\_global\_\_ void vectorAdd() {

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

if (idx < N) {

d\_C[idx] = d\_A[idx] + d\_B[idx];

}

}

float h\_A[N], h\_B[N], h\_C[N];

int main() {

for (int i = 0; i < N; i++) {

h\_A[i] = i \* 1.0f;

h\_B[i] = i \* 2.0f;

}

cudaMemcpyToSymbol(d\_A, h\_A, N \* sizeof(float));

cudaMemcpyToSymbol(d\_B, h\_B, N \* sizeof(float));

// Timing using CUDA events

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start);

vectorAdd<<<(N + 255) / 256, 256>>>();

cudaEventRecord(stop);

cudaEventSynchronize(stop);

float ms = 0;

cudaEventElapsedTime(&ms, start, stop); // in milliseconds

printf("Kernel execution time: %.5f ms\n", ms);

cudaMemcpyFromSymbol(h\_C, d\_C, N \* sizeof(float));

printf("Sample results:\n");

for (int i = 0; i < 5; i++) {

printf("h\_C[%d] = %.2f\n", i, h\_C[i]);

}

cudaDeviceProp prop;

cudaGetDeviceProperties(&prop, 0);

float memoryClockRate = prop.memoryClockRate; // kHz

int memoryBusWidth = prop.memoryBusWidth; // bits

float theoreticalBW = 2.0f \* memoryClockRate \* memoryBusWidth / 8 / 1e6; // GB/s

printf("\nTheoretical Memory Bandwidth: %.2f GB/s\n", theoreticalBW);

float timeInSeconds = ms / 1000.0f;

size\_t RBytes = 2 \* N \* sizeof(float); // A + B

size\_t WBytes = N \* sizeof(float); // C

float measuredBW = (RBytes + WBytes) / (timeInSeconds \* 1e9); // in GB/s

printf("Measured Memory Bandwidth: %.5f GB/s\n", measuredBW);

cudaEventDestroy(start);

cudaEventDestroy(stop);

return 0;

}