**Assignment 1**

#include<bits/stdc++.h>

#include<omp.h>

#include<chrono>

using namespace std;

using namespace std::chrono;

int N = 10, M = 10;

vector<int> graph [100000];

void bfs\_p(int start) {

vector<bool> vis(N);

queue<int> q;

q.push(start);

while(!q.empty()) {

int cur = q.front();

q.pop();

if(!vis[cur]) {

vis[cur] = 1; cout << cur <<" ";

// #pragma omp parallel spawns a group of threads,

// #pragma omp for divides loop iterations between the spawned threads.

// #pragma opm parallel for does both spawning of threads and dividing loop iterations

#pragma omp parallel for

for (int next: graph[cur]) {

if(!vis[next]) q.push(next);

}

}

}

}

void bfs(int start) {

vector<bool> vis(N);

queue<int> q;

q.push(start);

while(!q.empty()) {

int cur = q.front();

q.pop();

if(!vis[cur]) {

vis[cur] = 1; cout << cur <<" ";

for (int next: graph[cur]) {

if(!vis[next]) q.push(next);

}

}

}

}

void dfs\_p(int start) {

vector<bool> vis(N);

stack<int> q;

q.push(start);

while(!q.empty()) {

int cur = q.top();

q.pop();

if(!vis[cur]) {

vis[cur] = 1; cout << cur <<" ";

#pragma omp parallel for

for (int next: graph[cur]) {

if(!vis[next]) q.push(next);

}

}

}

}

void dfs(int start) {

vector<bool> vis(N);

stack<int> q;

q.push(start);

while(!q.empty()) {

int cur = q.top();

q.pop();

if(!vis[cur]) {

vis[cur] = 1; cout << cur <<" ";

for (int next: graph[cur]) {

if(!vis[next]) q.push(next);

}

}

}

}

int main() {

M = 4;

cout << "Enter 4 edges :" << endl;

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

int x, y; cin >> x >> y;

graph[x].push\_back(y);

graph[y].push\_back(x);

}

cout << "Paralel BFS traversal : ";

auto start = high\_resolution\_clock::now();

bfs\_p(0);

cout << endl;

auto end = high\_resolution\_clock::now();

auto dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Normal BFS traversal : ";

start = high\_resolution\_clock::now();

bfs(0);

cout << endl;

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Paralel DFS traversal : ";

start = high\_resolution\_clock::now();

dfs\_p(0);

cout << endl;

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Common DFS traversal : ";

start = high\_resolution\_clock::now();

dfs(0);

cout << endl;

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

}

**ASSIGNMENT 2**

#include <bits/stdc++.h>

#include <omp.h>

#include<chrono>

using namespace std;

using namespace std::chrono;

int N = 1000;

void bubble\_sort\_parallel(int a[], int n) {

#pragma omp parallel shared (a, n)

{

int i,j;

#pragma omp for

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

for(j = 0; j < n-i-1; j++) {

if(a[j] > a[j+1]) swap(a[j], a[j+1]);

}

}

}

}

void bubble\_sort(int a[], int n) {

int i,j;

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

for(j = 0; j < n-i-1; j++) {

if(a[j] > a[j+1])

swap(a[j], a[j+1]);

}

}

}

void merge(int a[], int l, int md, int r) {

vector<int> temp(r - l + 1);

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

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

if(a[i] <= a[j])

temp[k++] = a[i++];

else

temp[k++] = a[j++];

}

while(i <= md)

temp[k++] = a[i++];

while(j <= r)

temp[k++] = a[j++];

for(int i = 0; i < k; i++)

a[l+i] = temp[i];

}

void merge\_sort(int a[], int l, int r) {

if( l < r){

int md = (l + r) / 2;

merge\_sort(a, l, md);

merge\_sort(a, md + 1, r);

merge(a, l, md, r);

}

}

void merge\_sort\_parallel(int a[], int l, int r) {

if( l < r){

int md = (l + r) / 2;

#pragma omp parallel sections

{

#pragma omp section

merge\_sort\_parallel(a, l, md);

#pragma omp section

merge\_sort\_parallel(a, md + 1, r);

merge(a, l, md, r);

}

}

}

int main() {

int a[N], a1[N], a2[N], a3[N];

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

a[i] = rand() % N;

a1[i] = a[i]; a2[i] = a[i]; a3[i] = a[i];

}

cout << "Array after original Bubble sort: ";

auto start = high\_resolution\_clock::now();

bubble\_sort(a, N);

auto end = high\_resolution\_clock::now();

auto dur = duration\_cast<microseconds>(end - start);

for(int i = 0; i < min(N, 100); i++) {

cout << a[i] <<" ";

} cout << endl;

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Array after parallel Bubble sort: ";

start = high\_resolution\_clock::now();

bubble\_sort\_parallel(a1, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

for(int i = 0; i < min(N, 100); i++) {

cout << a1[i] <<" ";

} cout << endl;

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Array after original Merge sort: ";

start = high\_resolution\_clock::now();

merge\_sort(a2, 0, N-1);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

for(int i = 0; i < min(N, 100); i++) {

cout << a2[i] << " ";

} cout << endl;

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Array after parallel Merge sort: ";

start = high\_resolution\_clock::now();

merge\_sort\_parallel(a3, 0, N-1);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

for(int i = 0; i < min(N, 100); i++) {

cout << a3[i] << " ";

} cout << endl;

cout << "Time taken : " <<dur.count() << " ms" <<endl;

}

**ASSIGNMENT 3**

#include <bits/stdc++.h>

#include <omp.h>

#include<chrono>

using namespace std;

using namespace std::chrono;

void max\_reduction(int a[], int n) {

int mx = INT\_MIN;

//Reduction clause to perform recurrence calculations in parallel

#pragma omp parallel for reduction(max: mx)

for(int i = 0; i < n; i++)

if(a[i] > mx)

mx = a[i];

cout << "Maximum value: " << mx << endl;

}

void max\_original(int a[], int n) {

int mx = INT\_MIN;

for(int i = 0; i < n; i++)

if(a[i] > mx)

mx = a[i];

cout << "Maximum value: " << mx << endl;

}

void min\_reduction(int a[], int n) {

int mn = INT\_MAX;

#pragma omp parallel for reduction(min: mn)

for(int i = 0; i < n; i++)

if(a[i] < mn)

mn = a[i];

cout << "Minimum value: " << mn << endl;

}

void min\_original(int a[], int n) {

int mn = INT\_MAX;

for(int i = 0; i < n; i++)

if(a[i] < mn)

mn = a[i];

cout << "Minimum value: " << mn << endl;

}

void sum\_reduction(int a[], int n) {

int sum = 0;

#pragma omp parallel for reduction(+: sum)

for(int i = 0; i < n; i++)

sum += a[i];

cout << "Sum: " << sum << endl;

}

void sum\_original(int a[], int n) {

int sum = 0;

for(int i = 0; i < n; i++)

sum += a[i];

cout << "Sum: " << sum << endl;

}

void avg\_reduction(int a[], int n) {

double sum = 0, cnt = n;

#pragma omp parallel for reduction(+: sum)

for(int i = 0; i < n; i++)

sum += a[i];

double avg = sum / cnt;

cout << "Average: " << avg << endl;

}

void avg\_original(int a[], int n) {

double sum = 0, cnt = n;

for(int i = 0; i < n; i++)

sum += a[i];

double avg = sum / cnt;

cout << "Average: " << avg << endl;

}

int main() {

int N;

cout << "Enter number of elements: ";

cin >> N;

int a[N];

cout << "Generated array: ";

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

a[i] = rand() % N;

cout << a[i] << " ";

} cout<<"\n";

cout << "\n\nMax Original: \n";

auto start = high\_resolution\_clock::now();

max\_original(a, N);

auto end = high\_resolution\_clock::now();

auto dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Max Operation using Reduction: \n";

start = high\_resolution\_clock::now();

max\_reduction(a, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "\n\nMin Original: \n";

start = high\_resolution\_clock::now();

min\_original(a, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Min Operation using Reduction: \n";

start = high\_resolution\_clock::now();

min\_reduction(a, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "\n\nSum Original: \n";

start = high\_resolution\_clock::now();

sum\_original(a, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Sum Operation using Reduction: \n";

start = high\_resolution\_clock::now();

sum\_reduction(a, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "\n\nAverage Original: \n";

start = high\_resolution\_clock::now();

avg\_original(a, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Average Operation using Reduction: \n";

start = high\_resolution\_clock::now();

avg\_reduction(a, N);

end = high\_resolution\_clock::now();

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

return 0;

}

**ASSIGNMENT 5**

#include <bits/stdc++.h>

#include <omp.h>

#include<chrono>

using namespace std;

using namespace std::chrono;

int N;

double linear\_regression\_parallel(const double x[], const double y[]) {

double x\_mean = 0, y\_mean= 0;

#pragma omp parallel for reduction(+: x\_mean, y\_mean)

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

x\_mean += x[i];

y\_mean += y[i];

}

double n = N;

x\_mean /= n;

y\_mean /= n;

double num = 0, den = 0;

#pragma omp parallel for reduction(+: num, den)

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

num += (x[i] - x\_mean) \* (y[i] - y\_mean);

den += (x[i] - x\_mean) \* (x[i] - x\_mean);

}

return num / den;

}

double linear\_regression\_original(const double x[], const double y[]) {

double x\_mean = 0, y\_mean= 0;

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

x\_mean += x[i];

y\_mean += y[i];

}

double n = N;

x\_mean /= n;

y\_mean /= n;

double num = 0, den = 0;

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

num += (x[i] - x\_mean) \* (y[i] - y\_mean);

den += (x[i] - x\_mean) \* (x[i] - x\_mean);

}

return num / den;

}

/\* Input

15

1 2

2 5

3 6

4 9

5 11

6 15

7 20

8 24

9 26

15 34

17 37

91 193

88 189

122 270

111 240

\*/

int main() {

cout<<"Enter total number of elemebnts in dataset: ";

cin>>N;

double x[N], y[N];

cout << "Enter co-ordinates(x, y) of " << N <<" points" <<endl;

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

cin >> x[i] >> y[i];

}

cout<<"\n\nUsing sequential computing---\n ";

cout << "Linear Regression line has slope : ";

auto start = high\_resolution\_clock::now();

double result = linear\_regression\_original(x, y);

auto end = high\_resolution\_clock::now();

cout << result << endl;

auto dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

cout << "Using parallel computing---\n ";

cout << "Linear Regression line has slope : ";

start = high\_resolution\_clock::now();

result = linear\_regression\_parallel(x, y);

end = high\_resolution\_clock::now();

cout << result << endl;

dur = duration\_cast<microseconds>(end - start);

cout << "Time taken : " <<dur.count() << " ms" <<endl;

return 0;

}

//Cuda one

#include <iostream>

#include <cuda\_runtime.h>

#include <bits/stdc++.h>

#include <chrono>

using namespace std;

using namespace std::chrono;

// Serial vector addition

void serialVectorAdd(const float\* a, const float\* b, float\* c, int size)

{

    for (int i = 0; i < size; ++i) {

        c[i] = a[i] + b[i];

    }

}

// Parallel vector addition using CUDA

\_\_global\_\_ void parallelVectorAdd(const float\* a, const float\* b, float\* c, int size)

{

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

    if (idx < size)

        c[idx] = a[idx] + b[idx];

}

int main()

{

    int size = 1000000;

    size\_t bytes = size \* sizeof(float);

    // Allocate memory for host arrays

    float\* h\_a = new float[size];

    float\* h\_b = new float[size];

    float\* h\_c\_serial = new float[size];

    float\* h\_c\_parallel = new float[size];

    // Generate random numbers for input arrays

    for (int i = 0; i < size; ++i) {

        h\_a[i] = rand() % 1000;

        h\_b[i] = rand() % 1000;

    }

    // Serial vector addition

    auto serialStartTime = high\_resolution\_clock::now();

    serialVectorAdd(h\_a, h\_b, h\_c\_serial, size);

    auto serialEndTime = high\_resolution\_clock::now();

    auto serialDuration = duration\_cast<nanoseconds>(serialEndTime - serialStartTime);

    // Allocate memory for device arrays

    float\* d\_a, \* d\_b, \* d\_c;

    cudaMalloc((void\*\*)&d\_a, bytes);

    cudaMalloc((void\*\*)&d\_b, bytes);

    cudaMalloc((void\*\*)&d\_c, bytes);

    // Copy input arrays from host to device

    cudaMemcpy(d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

    cudaMemcpy(d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;

    int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

    // Parallel vector addition

    auto parallelStartTime = high\_resolution\_clock::now();

    parallelVectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_a, d\_b, d\_c, size);

    auto parallelEndTime = high\_resolution\_clock::now();

    auto parallelDuration = duration\_cast<nanoseconds>(parallelEndTime - parallelStartTime);

    // Copy the result array from device to host

    cudaMemcpy(h\_c\_parallel, d\_c, bytes, cudaMemcpyDeviceToHost);

    // Print the vectors

    cout << "Input Vectors:" << endl;

    for (int i = 0; i < 10; ++i) {

        cout << "a[" << i << "] = " << h\_a[i] << ", b[" << i << "] = " << h\_b[i] << endl;

    }

    cout << endl;

    cout << "Serial Vector Addition Result:" << endl;

    for (int i = 0; i < 10; ++i) {

        cout << "c\_serial[" << i << "] = " << h\_c\_serial[i] << endl;

    }

    cout << endl;

    cout << "Parallel Vector Addition Result:" << endl;

    for (int i = 0; i < 10; ++i) {

        cout << "c\_parallel[" << i << "] = " <<h\_c\_parallel[i] << endl;

    }

    cout << endl;

    // Print execution time

    cout << "Serial Execution Time: " << serialDuration.count() << " milliseconds" << endl;

    cout << "Parallel Execution Time: " << parallelDuration.count() << " milliseconds" << endl;

    // Clean up allocated memory

    delete[] h\_a;

    delete[] h\_b;

    delete[] h\_c\_serial;

    delete[] h\_c\_parallel;

    cudaFree(d\_a);

    cudaFree(d\_b);

    cudaFree(d\_c);

    return 0;

}

//Cuda 2

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <ctime>

#include <chrono>

#include <thread>

#define TILE\_WIDTH 32

// Kernel function for matrix multiplication on the GPU

\_\_global\_\_ void matrixMult(int \*a, int \*b, int \*c, int n)

{

    int row = blockIdx.y \* blockDim.y + threadIdx.y;

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

    if (row < n && col < n) {

        int sum = 0;

        for (int i = 0; i < n; ++i) {

            sum += a[row \* n + i] \* b[i \* n + col];

        }

        c[row \* n + col] = sum;

    }

}

int main()

{

    int n = 16;

    // Allocate memory for matrices A, B, and C

    int \*a = new int[n \* n];

    int \*b = new int[n \* n];

    int \*c = new int[n \* n];

    // Generate random values for matrices A and B

    std::srand(std::time(0));

    for (int i = 0; i < n \* n; ++i) {

        a[i] = std::rand() % 10;

        b[i] = std::rand() % 10;

    }

    // Variables for GPU memory

    int \*dev\_a, \*dev\_b, \*dev\_c;

    cudaMalloc(&dev\_a, n \* n \* sizeof(int));

    cudaMalloc(&dev\_b, n \* n \* sizeof(int));

    cudaMalloc(&dev\_c, n \* n \* sizeof(int));

    // Copy matrices A and B from host to device

    cudaMemcpy(dev\_a, a, n \* n \* sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(dev\_b, b, n \* n \* sizeof(int), cudaMemcpyHostToDevice);

    // Define grid and block dimensions for kernel launch

    dim3 dimGrid((n - 1) / TILE\_WIDTH + 1, (n - 1) / TILE\_WIDTH + 1, 1);

    dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH, 1);

    // Start timer for parallel execution

    clock\_t parallel\_start = clock();

    // Launch the matrix multiplication kernel on the GPU

    matrixMult<<<dimGrid, dimBlock>>>(dev\_a, dev\_b, dev\_c, n);

    // Wait for GPU to finish execution

    cudaDeviceSynchronize();

    // End timer for parallel execution

    clock\_t parallel\_end = clock();

    double parallel\_time = double(parallel\_end - parallel\_start) / CLOCKS\_PER\_SEC;

    // Copy the result matrix C from device to host

    cudaMemcpy(c, dev\_c, n \* n \* sizeof(int), cudaMemcpyDeviceToHost);

    // Print matrices A, B, and the result matrix C

    std::cout << "A matrix:\n";

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            std::cout << a[i \* n + j] << " ";

        }

        std::cout << "\n";

    }

    std::cout << "B matrix:\n";

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            std::cout << b[i \* n + j] << " ";

        }

        std::cout << "\n";

    }

    std::cout <<"Result matrix:\n";

    for (int i = 0; i < n; ++i) {

    for (int j = 0; j < n; ++j) {

    std::cout << c[i \* n + j] << " ";

    }

    std::cout << "\n";

    }

    // Start timer for serial execution

    clock\_t serial\_start = clock();

    std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

    // Perform matrix multiplication on the CPU (serial execution)

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            int sum = 0;

            for (int k = 0; k < n; ++k) {

                sum += a[i \* n + k] \* b[k \* n + j];

            }

            c[i \* n + j] = sum;

        }

    }

    // End timer for serial execution

    clock\_t serial\_end = clock();

    double serial\_time = double(serial\_end - serial\_start) / CLOCKS\_PER\_SEC;

    // Print execution times

    printf("Parallel execution time: : %3.7f ms\n", parallel\_time);

    printf("Serial execution time: : %3.7f ms\n", serial\_time);

    // Free GPU memory

    cudaFree(dev\_a);

    cudaFree(dev\_b);

    cudaFree(dev\_c);

    // Free CPU memory

    delete[] a;

    delete[] b;

    delete[] c;

    return 0;

}

//cuda one

#include <iostream>

#include <cuda\_runtime.h>

#include <bits/stdc++.h>

#include <chrono>

using namespace std;

using namespace std::chrono;

// Serial vector addition

void serialVectorAdd(const float\* a, const float\* b, float\* c, int size)

{

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

c[i] = a[i] + b[i];

}

}

// Parallel vector addition using CUDA

\_\_global\_\_ void parallelVectorAdd(const float\* a, const float\* b, float\* c, int size)

{

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

if (idx < size)

c[idx] = a[idx] + b[idx];

}

int main()

{

int size = 1000000;

size\_t bytes = size \* sizeof(float);

// Allocate memory for host arrays

float\* h\_a = new float[size];

float\* h\_b = new float[size];

float\* h\_c\_serial = new float[size];

float\* h\_c\_parallel = new float[size];

// Generate random numbers for input arrays

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

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

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

}

// Serial vector addition

auto serialStartTime = high\_resolution\_clock::now();

serialVectorAdd(h\_a, h\_b, h\_c\_serial, size);

auto serialEndTime = high\_resolution\_clock::now();

auto serialDuration = duration\_cast<nanoseconds>(serialEndTime - serialStartTime);

// Allocate memory for device arrays

float\* d\_a, \* d\_b, \* d\_c;

cudaMalloc(&d\_a, bytes);

cudaMalloc(&d\_b, bytes);

cudaMalloc(&d\_c, bytes);

// Copy input arrays from host to device

cudaMemcpy(d\_a, h\_a, bytes, cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, bytes, cudaMemcpyHostToDevice);

int threadsPerBlock = 256;

int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

// Parallel vector addition

auto parallelStartTime = high\_resolution\_clock::now();

parallelVectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_a, d\_b, d\_c, size);

cudaDeviceSynchronize();

auto parallelEndTime = high\_resolution\_clock::now();

auto parallelDuration = duration\_cast<nanoseconds>(parallelEndTime - parallelStartTime);

// Copy the result array from device to host

cudaMemcpy(h\_c\_parallel, d\_c, bytes, cudaMemcpyDeviceToHost);

// Print the vectors

cout << "Input Vectors:" << endl;

for (int i = 0; i < 10; ++i) {

cout << "a[" << i << "] = " << h\_a[i] << ", b[" << i << "] = " << h\_b[i] << endl;

}

cout << endl;

cout << "Serial Vector Addition Result:" << endl;

for (int i = 0; i < 10; ++i) {

cout << "c\_serial[" << i << "] = " << h\_c\_serial[i] << endl;

}

cout << endl;

cout << "Parallel Vector Addition Result:" << endl;

for (int i = 0; i < 10; ++i) {

cout << "c\_parallel[" << i << "] = " <<h\_c\_parallel[i] << endl;

}

cout << endl;

// Print execution time

cout << "Serial Execution Time: " << serialDuration.count() << " milliseconds" << endl;

cout << "Parallel Execution Time: " << parallelDuration.count() << " milliseconds" << endl;

// Clean up allocated memory

delete[] h\_a;

delete[] h\_b;

delete[] h\_c\_serial;

delete[] h\_c\_parallel;

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

return 0;

}

//cuda 2

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <ctime>

#include <chrono>

#include <thread>

#define TILE\_WIDTH 32

// Kernel function for matrix multiplication on the GPU

\_\_global\_\_ void matrixMult(int \*a, int \*b, int \*c, int n)

{

int row = blockIdx.y \* blockDim.y + threadIdx.y;

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

if (row < n && col < n) {

int sum = 0;

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

sum += a[row \* n + i] \* b[i \* n + col];

}

c[row \* n + col] = sum;

}

}

int main()

{

int n = 16;

// Allocate memory for matrices A, B, and C

int \*a = new int[n \* n];

int \*b = new int[n \* n];

int \*c = new int[n \* n];

// Generate random values for matrices A and B

std::srand(std::time(0));

for (int i = 0; i < n \* n; ++i) {

a[i] = std::rand() % 10;

b[i] = std::rand() % 10;

}

// Variables for GPU memory

int \*dev\_a, \*dev\_b, \*dev\_c;

cudaMalloc(&dev\_a, n \* n \* sizeof(int));

cudaMalloc(&dev\_b, n \* n \* sizeof(int));

cudaMalloc(&dev\_c, n \* n \* sizeof(int));

// Copy matrices A and B from host to device

cudaMemcpy(dev\_a, a, n \* n \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(dev\_b, b, n \* n \* sizeof(int), cudaMemcpyHostToDevice);

// Define grid and block dimensions for kernel launch

dim3 dimGrid((n - 1) / TILE\_WIDTH + 1, (n - 1) / TILE\_WIDTH + 1, 1);

dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH, 1);

// Start timer for parallel execution

clock\_t parallel\_start = clock();

// Launch the matrix multiplication kernel on the GPU

matrixMult<<<dimGrid, dimBlock>>>(dev\_a, dev\_b, dev\_c, n);

// Wait for GPU to finish execution

cudaDeviceSynchronize();

// End timer for parallel execution

clock\_t parallel\_end = clock();

double parallel\_time = double(parallel\_end - parallel\_start) / CLOCKS\_PER\_SEC;

// Copy the result matrix C from device to host

cudaMemcpy(c, dev\_c, n \* n \* sizeof(int), cudaMemcpyDeviceToHost);

// Print matrices A, B, and the result matrix C

std::cout << "A matrix:\n";

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

for (int j = 0; j < n; ++j) {

std::cout << a[i \* n + j] << " ";

}

std::cout << "\n";

}

std::cout << "B matrix:\n";

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

for (int j = 0; j < n; ++j) {

std::cout << b[i \* n + j] << " ";

}

std::cout << "\n";

}

std::cout <<"Result matrix:\n";

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

for (int j = 0; j < n; ++j) {

std::cout << c[i \* n + j] << " ";

}

std::cout << "\n";

}

// Start timer for serial execution

clock\_t serial\_start = clock();

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

// Perform matrix multiplication on the CPU (serial execution)

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

for (int j = 0; j < n; ++j) {

int sum = 0;

for (int k = 0; k < n; ++k) {

sum += a[i \* n + k] \* b[k \* n + j];

}

c[i \* n + j] = sum;

}

}

// End timer for serial execution

clock\_t serial\_end = clock();

double serial\_time = double(serial\_end - serial\_start) / CLOCKS\_PER\_SEC;

// Print execution times

printf("Parallel execution time: : %3.7f ms\n", parallel\_time);

printf("Serial execution time: : %3.7f ms\n", serial\_time);

// Free GPU memory

cudaFree(dev\_a);

cudaFree(dev\_b);

cudaFree(dev\_c);

// Free CPU memory

delete[] a;

delete[] b;

delete[] c;

return 0;

}