#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <omp.h>

#include <chrono>

#include <cstdlib>

using namespace std;

using namespace std::chrono;

class Graph {

public:

int V;

vector<vector<int>> adj;

Graph(int V) {

this->V = V;

adj.resize(V);

}

void addEdge(int u, int v) {

adj[u].push\_back(v);

adj[v].push\_back(u);

}

void generateRandomGraph(int edges);

void sequentialBFS(int start);

void parallelBFS(int start);

void sequentialDFS(int start);

void parallelDFS(int start);

};

void Graph::generateRandomGraph(int edges) {

srand(time(0));

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

int u = rand() % V;

int v = rand() % V;

if (u != v) {

addEdge(u, v);

}

}

}

void Graph::sequentialBFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int node = q.front();

q.pop();

for (int neighbor : adj[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

void Graph::parallelBFS(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int size = q.size();

vector<int> levelNodes;

#pragma omp parallel for shared(visited, q)

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

int node;

#pragma omp critical

{

if (!q.empty()) {

node = q.front();

q.pop();

}

}

for (int neighbor : adj[node]) {

if (!visited[neighbor]) {

visited[neighbor] = true;

#pragma omp critical

levelNodes.push\_back(neighbor);

}

}

}

for (int node : levelNodes) {

q.push(node);

}

}

}

void Graph::sequentialDFS(int start) {

vector<bool> visited(V, false);

stack<int> s;

s.push(start);

while (!s.empty()) {

int node = s.top();

s.pop();

if (!visited[node]) {

visited[node] = true;

for (int neighbor : adj[node]) {

if (!visited[neighbor]) {

s.push(neighbor);

}

}

}

}

}

void Graph::parallelDFS(int start) {

vector<bool> visited(V, false);

stack<int> s;

s.push(start);

#pragma omp parallel

{

while (!s.empty()) {

int node;

#pragma omp critical

{

if (!s.empty()) {

node = s.top();

s.pop();

}

}

if (!visited[node]) {

visited[node] = true;

#pragma omp parallel for

for (int i = 0; i < adj[node].size(); i++) {

int neighbor = adj[node][i];

if (!visited[neighbor]) {

#pragma omp critical

s.push(neighbor);

}

}

}

}

}

}

int main() {

int V, E;

cout << "Enter the number of vertices: ";

cin >> V;

cout << "Enter the number of edges: ";

cin >> E;

if (E > (V \* (V - 1)) / 2) {

cout << "Too many edges for the given number of vertices. Adjusting to maximum possible edges.\n";

E = (V \* (V - 1)) / 2;

}

Graph g(V);

g.generateRandomGraph(E);

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

g.sequentialBFS(0);

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

auto seqBFS\_time = duration\_cast<microseconds>(stop - start);

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

g.parallelBFS(0);

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

auto parBFS\_time = duration\_cast<microseconds>(stop - start);

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

g.sequentialDFS(0);

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

auto seqDFS\_time = duration\_cast<microseconds>(stop - start);

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

g.parallelDFS(0);

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

auto parDFS\_time = duration\_cast<microseconds>(stop - start);

cout << "Sequential BFS Time: " << seqBFS\_time.count() << " microseconds" << endl;

cout << "Parallel BFS Time: " << parBFS\_time.count() << " microseconds" << endl;

cout << "Speedup for BFS: " << (double)seqBFS\_time.count() / parBFS\_time.count() << endl;

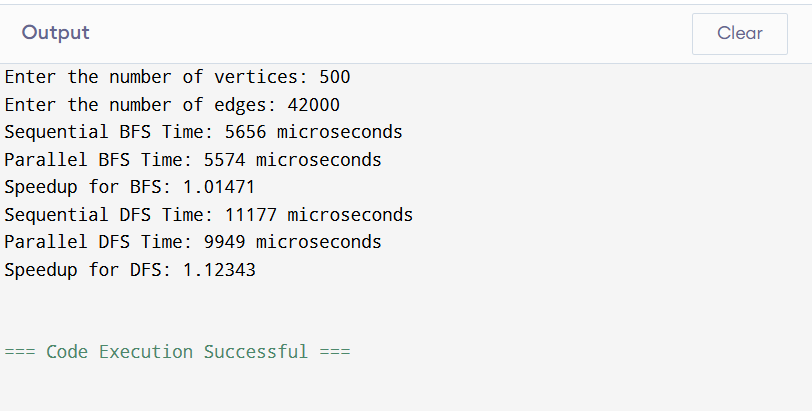
cout << "Sequential DFS Time: " << seqDFS\_time.count() << " microseconds" << endl;

cout << "Parallel DFS Time: " << parDFS\_time.count() << " microseconds" << endl;

cout << "Speedup for DFS: " << (double)seqDFS\_time.count() / parDFS\_time.count() << endl;

return 0;

}



#include <iostream>

#include <chrono>

#include <omp.h>

#include <vector>

using namespace std;

using namespace std::chrono;

class Sorting {

private:

    vector<int> arr;

    int n;

    void merge(vector<int>& arr, int start, int mid, int end) {

        vector<int> left(arr.begin() + start, arr.begin() + mid + 1);

        vector<int> right(arr.begin() + mid + 1, arr.begin() + end + 1);

        int i = 0, j = 0, k = start;

        while (i < left.size() && j < right.size()) {

            if (left[i] <= right[j]) {

                arr[k++] = left[i++];

            } else {

                arr[k++] = right[j++];

            }

        }

        while (i < left.size()) arr[k++] = left[i++];

        while (j < right.size()) arr[k++] = right[j++];

    }

public:

    Sorting(vector<int> inputArr) : arr(inputArr), n(inputArr.size()) {}

    void bubbleSort() {

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

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

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

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

                }

            }

        }

    }

    void mergeSort(int start, int end) {

        if (start < end) {

            int mid = start + (end - start) / 2;

            mergeSort(start, mid);

            mergeSort(mid + 1, end);

            merge(arr, start, mid, end);

        }

    }

    void parallelBubbleSort() {

        bool sorted = false;

        while (!sorted) {

            sorted = true;

            #pragma omp parallel for shared(arr, sorted)

            for (int i = 0; i < n - 1; i += 2) {

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

                    swap(arr[i], arr[i + 1]);

                    sorted = false;

                }

            }

            #pragma omp parallel for shared(arr, sorted)

            for (int i = 1; i < n - 1; i += 2) {

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

                    swap(arr[i], arr[i + 1]);

                    sorted = false;

                }

            }

        }

    }

    void parallelMergeSort(int start, int end) {

        if (start < end) {

            int mid = start + (end - start) / 2;

            #pragma omp parallel sections

            {

                #pragma omp section

                parallelMergeSort(start, mid);

                #pragma omp section

                parallelMergeSort(mid + 1, end);

            }

            merge(arr, start, mid, end);

        }

    }

    void displayArray() {

        for (int i : arr) {

            cout << i << " ";

        }

        cout << endl;

    }

    vector<int> getArray() {

        return arr;

    }

};

int main() {

    cout << "Enter number of elements: ";

    int n;

    cin >> n;

    vector<int> inputArr(n);

    for (int i = 0; i < n; i++) {

        inputArr[i] = rand() % 100;

        cout << inputArr[i] << " ";

    }

    cout << "\n\n";

    Sorting sorter(inputArr);

    vector<int> originalArr = sorter.getArray();

    cout << "Sequential Execution:\n\n";

    cout << "Bubble Sort: ";

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

    sorter.bubbleSort();

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

    sorter.displayArray();

    double seq\_bubble\_time = duration<double, milli>(end - start).count();

    cout << "TIME TAKEN: " << seq\_bubble\_time << " ms\n";

    sorter = Sorting(originalArr);

    cout << "\nMerge Sort: ";

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

    sorter.mergeSort(0, n - 1);

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

    sorter.displayArray();

    double seq\_merge\_time = duration<double, milli>(end - start).count();

    cout << "TIME TAKEN: " << seq\_merge\_time << " ms\n";

    cout << "\nParallel Execution:\n\n";

    sorter = Sorting(originalArr);

    cout << "Parallel Bubble Sort: ";

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

    sorter.parallelBubbleSort();

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

    sorter.displayArray();

    double par\_bubble\_time = duration<double, milli>(end - start).count();

    cout << "TIME TAKEN: " << par\_bubble\_time << " ms\n";

    cout << "Speedup Factor (Bubble Sort): " << seq\_bubble\_time / par\_bubble\_time << "\n";

    sorter = Sorting(originalArr);

    cout << "\nParallel Merge Sort: ";

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

    sorter.parallelMergeSort(0, n - 1);

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

    sorter.displayArray();

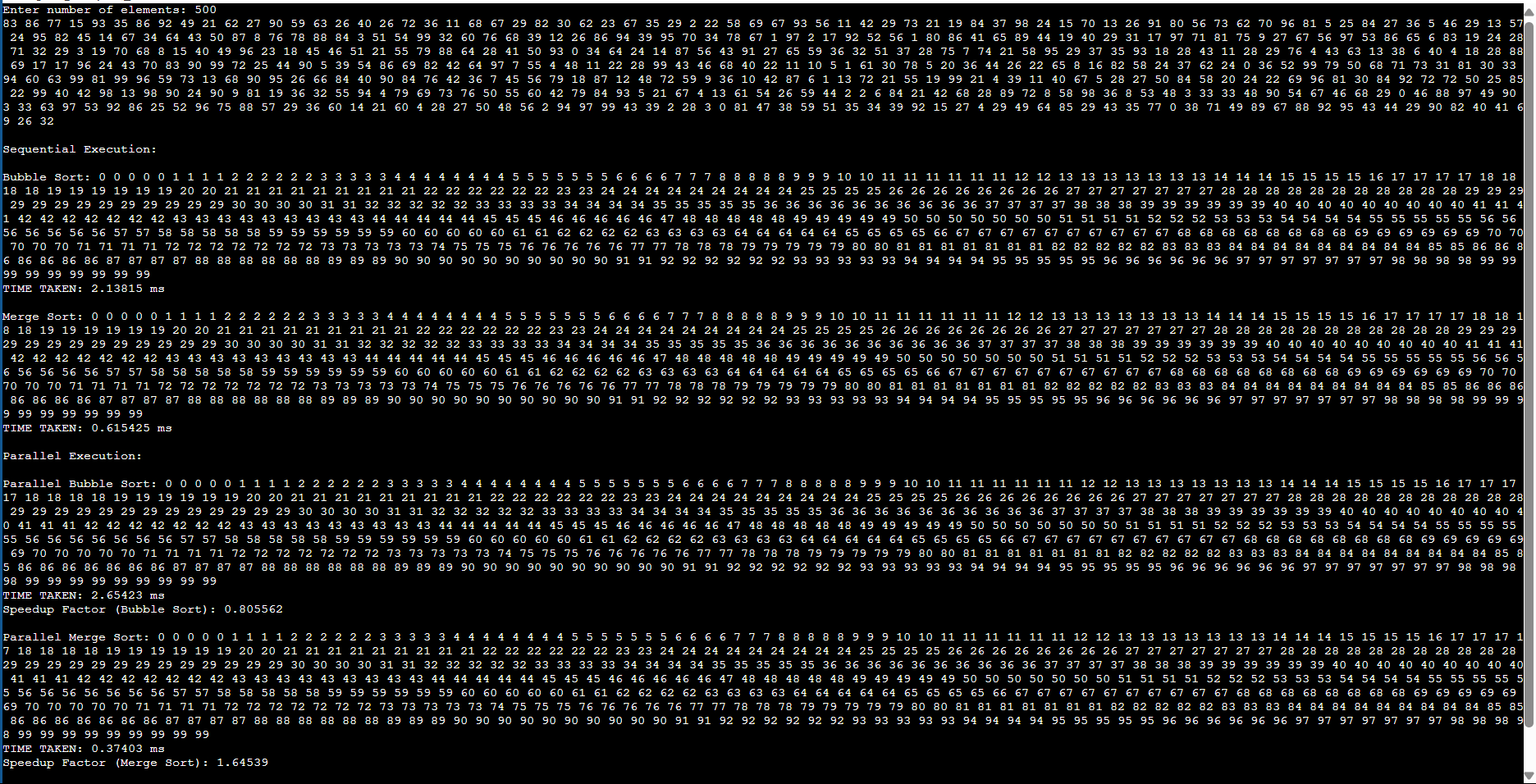
    double par\_merge\_time = duration<double, milli>(end - start).count();

    cout << "TIME TAKEN: " << par\_merge\_time << " ms\n";

    cout << "Speedup Factor (Merge Sort): " << seq\_merge\_time / par\_merge\_time << "\n";

    return 0;

}



#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <chrono>

#include <omp.h>

using namespace std;

class ParallelMinMax {

private:

    vector<int> arr;

    int size;

    int min\_seq, max\_seq, min\_par, max\_par;

    long long sum\_seq, sum\_par;

    double avg\_seq, avg\_par;

    double time\_min\_seq, time\_max\_seq, time\_sum\_seq, time\_avg\_seq;

    double time\_min\_par, time\_max\_par, time\_sum\_par, time\_avg\_par;

public:

    ParallelMinMax(int size) : size(size), min\_seq(0), max\_seq(0), min\_par(0), max\_par(0), sum\_seq(0), sum\_par(0), avg\_seq(0), avg\_par(0),

        time\_min\_seq(0), time\_max\_seq(0), time\_sum\_seq(0), time\_avg\_seq(0), time\_min\_par(0), time\_max\_par(0), time\_sum\_par(0), time\_avg\_par(0) {

        arr.resize(size);

    }

    void generateRandomArray() {

        srand(time(0));

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

            arr[i] = rand() % 100 + 1;

        }

    }

    void computeMinSequential() {

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

        min\_seq = arr[0];

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

            if (arr[i] < min\_seq) min\_seq = arr[i];

        }

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

        time\_min\_seq = chrono::duration<double>(end - start).count();

    }

    void computeMaxSequential() {

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

        max\_seq = arr[0];

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

            if (arr[i] > max\_seq) max\_seq = arr[i];

        }

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

        time\_max\_seq = chrono::duration<double>(end - start).count();

    }

    void computeSumSequential() {

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

        sum\_seq = 0;

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

            sum\_seq += arr[i];

        }

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

        time\_sum\_seq = chrono::duration<double>(end - start).count();

    }

    void computeAvgSequential() {

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

        avg\_seq = static\_cast<double>(sum\_seq) / size;

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

        time\_avg\_seq = chrono::duration<double>(end - start).count();

    }

    void computeMinParallel() {

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

        min\_par = arr[0];

        #pragma omp parallel for reduction(min:min\_par)

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

            if (arr[i] < min\_par) min\_par = arr[i];

        }

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

        time\_min\_par = chrono::duration<double>(end - start).count();

    }

    void computeMaxParallel() {

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

        max\_par = arr[0];

        #pragma omp parallel for reduction(max:max\_par)

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

            if (arr[i] > max\_par) max\_par = arr[i];

        }

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

        time\_max\_par = chrono::duration<double>(end - start).count();

    }

    void computeSumParallel() {

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

        sum\_par = 0;

        #pragma omp parallel for reduction(+:sum\_par)

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

            sum\_par += arr[i];

        }

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

        time\_sum\_par = chrono::duration<double>(end - start).count();

    }

    void computeAvgParallel() {

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

        avg\_par = static\_cast<double>(sum\_par) / size;

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

        time\_avg\_par = chrono::duration<double>(end - start).count();

    }

    void sequentialComputation() {

        computeMinSequential();

        computeMaxSequential();

        computeSumSequential();

        computeAvgSequential();

    }

    void parallelComputation() {

        computeMinParallel();

        computeMaxParallel();

        computeSumParallel();

        computeAvgParallel();

    }

    void displayResults() {

        cout << "---- Sequential Computation ----\n";

        cout << "Min: " << min\_seq << " | Time: " << time\_min\_seq << " sec\n";

        cout << "Max: " << max\_seq << " | Time: " << time\_max\_seq << " sec\n";

        cout << "Sum: " << sum\_seq << " | Time: " << time\_sum\_seq << " sec\n";

        cout << "Average: " << avg\_seq << " | Time: " << time\_avg\_seq << " sec\n\n";

        cout << "---- Parallel Computation ----\n";

        cout << "Min: " << min\_par << " | Time: " << time\_min\_par << " sec\n";

        cout << "Max: " << max\_par << " | Time: " << time\_max\_par << " sec\n";

        cout << "Sum: " << sum\_par << " | Time: " << time\_sum\_par << " sec\n";

        cout << "Average: " << avg\_par << " | Time: " << time\_avg\_par << " sec\n\n";

        cout << "---- Speedup Factors ----\n";

        cout << "Speedup (Min): " << (time\_min\_seq / time\_min\_par) << "x\n";

        cout << "Speedup (Max): " << (time\_max\_seq / time\_max\_par) << "x\n";

        cout << "Speedup (Sum): " << (time\_sum\_seq / time\_sum\_par) << "x\n";

        cout << "Speedup (Average): " << (time\_avg\_seq / time\_avg\_par) << "x\n";

    }

};

int main() {

    int size;

    cout << "Enter array size: ";

    cin >> size;

    if (size <= 0) {

        cout << "Invalid size!" << endl;

        return 1;

    }

    ParallelMinMax pm(size);

    pm.generateRandomArray();

    pm.sequentialComputation();

    pm.parallelComputation();

    pm.displayResults();

    return 0;

}



#include <cuda.h>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <iostream>

#include "cuda\_runtime.h"

using namespace std;

#define N 1000

\_\_global\_\_ void vectorAdd(int \*d\_a, int \*d\_b, int \*d\_c, int n) {

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

if (i < n) {

     d\_c[i] = d\_a[i] + d\_b[i];

}

}

\_\_global\_\_ void matrixMultiplyKernel(float \*a, float \*b, float \*c) {

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

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

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

     float sum = 0;

     for (int i = 0; i < N; i++) {

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

     }

     c[row \* N + col] = sum;

}

}

class VectorAddition {

public:

void performVectorAddition() {

     int \*a, \*b, \*c, \*d;

     int \*d\_a, \*d\_b, \*d\_c;

     size\_t size = N \* sizeof(int);

     a = (int \*)malloc(size);

     b = (int \*)malloc(size);

     c = (int \*)malloc(size);

     d = (int \*)malloc(size);

     srand(time(NULL));

     for (int i = 0; i < N; i++) {

         a[i] = rand() % 100;

         b[i] = rand() % 100;

     }

     clock\_t start\_cpu = clock();

     for (int i = 0; i < N; i++) {

         c[i] = a[i] + b[i];

     }

     clock\_t end\_cpu = clock();

     double cpu\_time = (double)(end\_cpu - start\_cpu) / CLOCKS\_PER\_SEC;

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

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

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

     cudaMemcpy(d\_a, a, size, cudaMemcpyHostToDevice);

     cudaMemcpy(d\_b, b, size, cudaMemcpyHostToDevice);

     int threadsPerBlock = 256;

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

     cudaEvent\_t start, stop;

     cudaEventCreate(&start);

     cudaEventCreate(&stop);

     cudaEventRecord(start);

     vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_a, d\_b, d\_c, N);

     cudaEventRecord(stop);

     cudaMemcpy(d, d\_c, size, cudaMemcpyDeviceToHost);

     cudaEventSynchronize(stop);

     float gpu\_time = 0;

     cudaEventElapsedTime(&gpu\_time, start, stop);

     bool match = true;

     for (int i = 0; i < N; i++) {

         if (c[i] != d[i]) {

             match = false;

             break;

         }

     }

     printf("CPU Time: %.6f s\n", cpu\_time);

     printf("GPU Time: %.6f ms\n", gpu\_time);

     printf("Speedup Factor: %.2f\n", (cpu\_time) \*10000/ gpu\_time);

     printf("Arrays Match: %s\n", match ? "Yes" : "No");

     free(a); free(b); free(c); free(d);

     cudaFree(d\_a); cudaFree(d\_b); cudaFree(d\_c);

     cudaEventDestroy(start);

     cudaEventDestroy(stop);

}

};

class MatrixMultiplier {

private:

float \*hostA, \*hostB, \*hostC, \*hostD;

float \*devA, \*devB, \*devC;

int size;

float cpuTime, gpuTime;

public:

MatrixMultiplier() {

     size = N \* N \* sizeof(float);

     hostA = (float \*)malloc(size);

     hostB = (float \*)malloc(size);

     hostC = (float \*)malloc(size);

     hostD = (float \*)malloc(size);

     cudaMalloc((void \*\*)&devA, size);

     cudaMalloc((void \*\*)&devB, size);

     cudaMalloc((void \*\*)&devC, size);

     cpuTime = gpuTime = 0.0;

}

     ~MatrixMultiplier() {

     free(hostA);

     free(hostB);

     free(hostC);

     free(hostD);

     cudaFree(devA);

     cudaFree(devB);

     cudaFree(devC);

}

     void initializeMatrices() {

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

         hostA[i] = rand() % 100;

         hostB[i] = rand() % 100;

     }

}

void gpuMatrixMultiplication() {

     cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);

     cudaMemcpy(devB, hostB, size, cudaMemcpyHostToDevice);

     dim3 dimBlock(16, 16);

     dim3 dimGrid((N + 15) / 16, (N + 15) / 16);

     clock\_t tic = clock();

     matrixMultiplyKernel<<<dimGrid, dimBlock>>>(devA, devB, devC);

     cudaDeviceSynchronize();

     clock\_t toc = clock();

     gpuTime = ((float)(toc - tic)) / CLOCKS\_PER\_SEC;

     cudaMemcpy(hostC, devC, size, cudaMemcpyDeviceToHost);

}

     void cpuMatrixMultiplication() {

     clock\_t tic = clock();

     for (int i = 0; i < N; i++) {

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

             float sum = 0;

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

                 sum += hostA[i \* N + k] \* hostB[k \* N + j];

             }

             hostD[i \* N + j] = sum;

         }

     }

             clock\_t toc = clock();

     cpuTime = ((float)(toc - tic)) / CLOCKS\_PER\_SEC;

}

bool verifyEquality() {

     float tolerance = 1e-5;

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

         if (fabs(hostC[i] - hostD[i]) > tolerance) {

             printf("Mismatch at index %d: GPU = %f, CPU = %f\n", i, hostC[i], hostD[i]);

             return false;

             }

     }

     return true;

}

void printResults() {

     printf("CPU Time: %f seconds\n", cpuTime);

     printf("GPU Time: %f seconds\n", gpuTime);

     if (gpuTime > 0) {

         printf("Speed-Up Factor: %.2f x\n", (cpuTime) / gpuTime);

     } else {

         printf("Speed-Up Factor: N/A (GPU time too small)\n");

     }

}

};

int main() {

int choice;

VectorAddition vectorAdder;

MatrixMultiplier matrixMultiplier;

matrixMultiplier.initializeMatrices();

matrixMultiplier.cpuMatrixMultiplication();

matrixMultiplier.gpuMatrixMultiplication();

bool success = matrixMultiplier.verifyEquality();

do {

     cout << "\nMenu:" << endl;

     cout << "1. Vector Addition" << endl;

     cout << "2. Matirx Multiplication" << endl;

     cout << "3. Exit" << endl;

     cout << "Enter your choice: ";

     cin >> choice;

     switch (choice) {

         case 1:

             vectorAdder.performVectorAddition();

             break;

         case 2:

             matrixMultiplier.printResults();

             printf("Verification: %s\n", success ? "true" : "false");

             break;

         case 3:

             cout << "Exiting..." << endl;

             break;

         default:

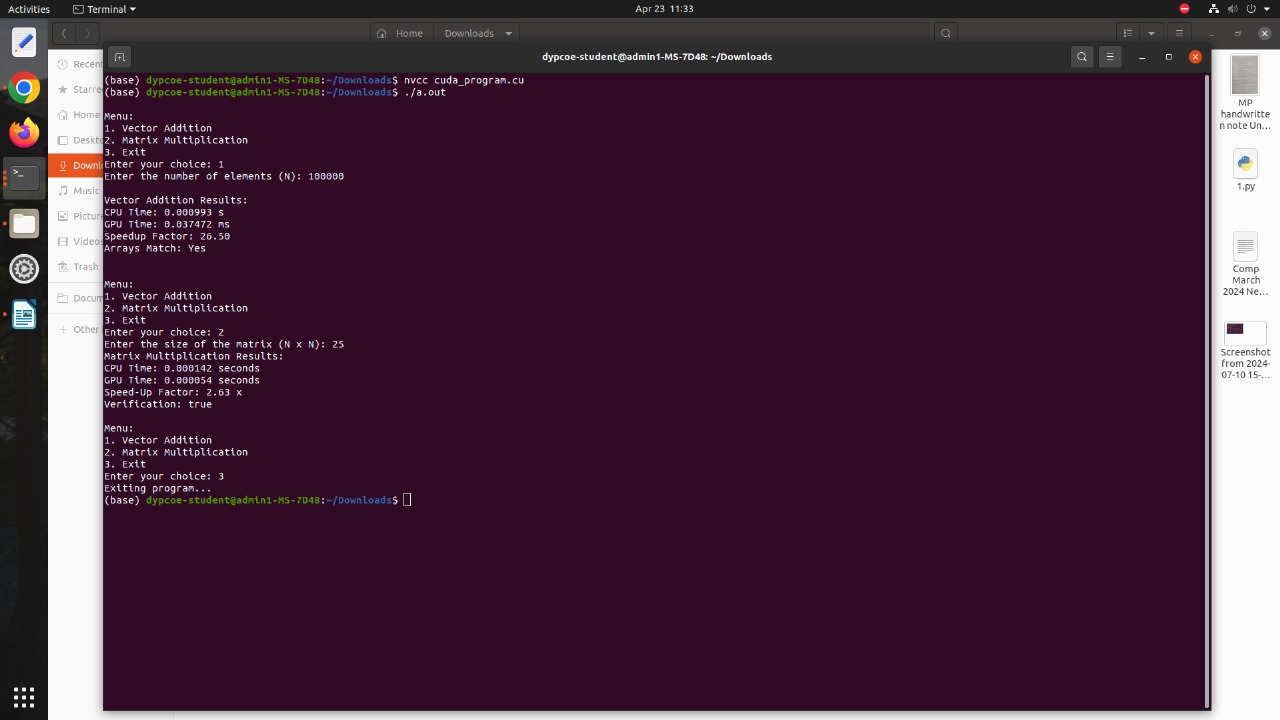
             cout << "Invalid choice!" << endl;

     }

} while (choice != 3);

return 0;

}



#include <iostream>

#include <vector>

#include <cmath>

#include <chrono>

#include <cuda.h>

#define N 1000  // number of data points

// CUDA Kernel for parallel sum calculation

\_\_global\_\_ void computeSums(const float\* x, const float\* y, float\* sumX, float\* sumY, float\* sumXY, float\* sumX2, int n) {

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

if (i < n) {

     atomicAdd(sumX, x[i]);

     atomicAdd(sumY, y[i]);

     atomicAdd(sumXY, x[i] \* y[i]);

     atomicAdd(sumX2, x[i] \* x[i]);

}

}

// CPU implementation of linear regression

void linearRegressionCPU(const float\* x, const float\* y, int n, float& b0, float& b1) {

float sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;

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

     sumX += x[i];

     sumY += y[i];

     sumXY += x[i] \* y[i];

     sumX2 += x[i] \* x[i];

}

b1 = (n \* sumXY - sumX \* sumY) / (n \* sumX2 - sumX \* sumX);

b0 = (sumY - b1 \* sumX) / n;

}

int main() {

// Host data

std::vector<float> h\_x(N), h\_y(N);

float b0\_cpu, b1\_cpu;

float b0\_gpu, b1\_gpu;

// Generate synthetic data: y = 2.5 + 1.2x

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

     h\_x[i] = i;

     h\_y[i] = 2.5f + 1.2f \* i;

}

// === CPU Linear Regression ===

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

linearRegressionCPU(h\_x.data(), h\_y.data(), N, b0\_cpu, b1\_cpu);

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

float cpu\_time = std::chrono::duration<float, std::milli>(end\_cpu - start\_cpu).count();

// === GPU Linear Regression ===

float \*d\_x, \*d\_y, \*d\_sumX, \*d\_sumY, \*d\_sumXY, \*d\_sumX2;

cudaMalloc(&d\_x, N \* sizeof(float));

cudaMalloc(&d\_y, N \* sizeof(float));

cudaMalloc(&d\_sumX, sizeof(float));

cudaMalloc(&d\_sumY, sizeof(float));

cudaMalloc(&d\_sumXY, sizeof(float));

cudaMalloc(&d\_sumX2, sizeof(float));

// Initialize device memory for sums

cudaMemset(d\_sumX, 0, sizeof(float));

cudaMemset(d\_sumY, 0, sizeof(float));

cudaMemset(d\_sumXY, 0, sizeof(float));

cudaMemset(d\_sumX2, 0, sizeof(float));

cudaMemcpy(d\_x, h\_x.data(), N \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_y, h\_y.data(), N \* sizeof(float), cudaMemcpyHostToDevice);

int threadsPerBlock = 256;

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

cudaEvent\_t start\_gpu, stop\_gpu;

cudaEventCreate(&start\_gpu);

cudaEventCreate(&stop\_gpu);

cudaEventRecord(start\_gpu);

computeSums<<<blocksPerGrid, threadsPerBlock>>>(d\_x, d\_y, d\_sumX, d\_sumY, d\_sumXY, d\_sumX2, N);

cudaEventRecord(stop\_gpu);

cudaEventSynchronize(stop\_gpu);

float gpu\_time = 0;

cudaEventElapsedTime(&gpu\_time, start\_gpu, stop\_gpu);

float sumX, sumY, sumXY, sumX2;

cudaMemcpy(&sumX, d\_sumX, sizeof(float), cudaMemcpyDeviceToHost);

cudaMemcpy(&sumY, d\_sumY, sizeof(float), cudaMemcpyDeviceToHost);

cudaMemcpy(&sumXY, d\_sumXY, sizeof(float), cudaMemcpyDeviceToHost);

cudaMemcpy(&sumX2, d\_sumX2, sizeof(float), cudaMemcpyDeviceToHost);

b1\_gpu = (N \* sumXY - sumX \* sumY) / (N \* sumX2 - sumX \* sumX);

b0\_gpu = (sumY - b1\_gpu \* sumX) / N;

bool verified = fabs(b0\_cpu - b0\_gpu) < 1e-2 && fabs(b1\_cpu - b1\_gpu) < 1e-2;

std::cout << "=== Linear Regression using CUDA ===\n";

std::cout << "CPU Time: " << cpu\_time << " ms\n";

std::cout << "GPU Time: " << gpu\_time << " ms\n";

std::cout << "Speedup Factor: " << cpu\_time / gpu\_time << "x\n";

std::cout << "Verification: " << (verified ? "PASSED ✅" : "FAILED ❌") << "\n";

std::cout << "Equation (CPU): y = " << b0\_cpu << " + " << b1\_cpu << " \* x\n";

std::cout << "Equation (GPU): y = " << b0\_gpu << " + " << b1\_gpu << " \* x\n";

cudaFree(d\_x);

cudaFree(d\_y);

cudaFree(d\_sumX);

cudaFree(d\_sumY);

cudaFree(d\_sumXY);

cudaFree(d\_sumX2);

cudaEventDestroy(start\_gpu);

cudaEventDestroy(stop\_gpu);

return 0;

}

