HPC

Assign 1- in cuda

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <unordered\_set>

#include <cuda\_runtime.h>

using namespace std;

\_\_global\_\_ void dfs\_kernel(bool\* visited, int\* adj, int\* stack, int\* stack\_top, int start) {

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

if (v == start) {

stack[atomicAdd(stack\_top, 1)] = v;

visited[v] = true;

}

\_\_syncthreads();

while (\*stack\_top >= 0) {

int top = atomicAdd(stack\_top, -1);

v = stack[top];

if (!visited[v]) {

printf("%d ", v); // Output the vertex visited

visited[v] = true;

}

for (int i = adj[v]; i < adj[v + 1]; ++i) {

int neighbor = adj[i];

if (!visited[neighbor]) {

stack[atomicAdd(stack\_top, 1)] = neighbor;

}

}

\_\_syncthreads();

}

}

\_\_global\_\_ void bfs\_kernel(bool\* visited, int\* adj, int\* queue, int\* queue\_front, int\* queue\_rear, int start) {

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

if (v == start) {

queue[atomicAdd(queue\_rear, 1)] = v;

visited[v] = true;

}

\_\_syncthreads();

while (\*queue\_front <= \*queue\_rear) {

int front = atomicAdd(queue\_front, 1);

v = queue[front];

printf("%d ", v); // Output the vertex visited

for (int i = adj[v]; i < adj[v + 1]; ++i) {

int neighbor = adj[i];

if (!visited[neighbor]) {

int rear = atomicAdd(queue\_rear, 1);

queue[rear] = neighbor;

visited[neighbor] = true;

}

}

\_\_syncthreads();

}

}

class Graph {

int V;

vector<int> adj\_list;

bool\* visited;

int\* d\_adj\_list;

bool\* d\_visited;

int\* d\_stack;

int\* d\_stack\_top;

int\* d\_queue;

int\* d\_queue\_front;

int\* d\_queue\_rear;

public:

Graph(int V);

void addEdge(int v, int w);

void DFS(int v);

void BFS(int v);

void cleanup();

};

Graph::Graph(int V) {

this->V = V;

adj\_list.push\_back(0); // Initialize the adjacency list with 0

cudaMalloc(&d\_adj\_list, sizeof(int) \* (V + 1));

cudaMalloc(&d\_visited, sizeof(bool) \* V);

cudaMalloc(&d\_stack, sizeof(int) \* V);

cudaMalloc(&d\_stack\_top, sizeof(int));

cudaMalloc(&d\_queue, sizeof(int) \* V);

cudaMalloc(&d\_queue\_front, sizeof(int));

cudaMalloc(&d\_queue\_rear, sizeof(int));

}

void Graph::addEdge(int v, int w) {

adj\_list.push\_back(w);

}

void Graph::DFS(int v) {

visited = new bool[V];

memset(visited, false, V \* sizeof(bool));

cudaMemcpy(d\_adj\_list, &adj\_list[0], sizeof(int) \* adj\_list.size(), cudaMemcpyHostToDevice);

cudaMemcpy(d\_visited, visited, sizeof(bool) \* V, cudaMemcpyHostToDevice);

int\* stack\_top = new int;

\*stack\_top = -1;

cudaMemcpy(d\_stack\_top, stack\_top, sizeof(int), cudaMemcpyHostToDevice);

dfs\_kernel<<<1, V>>>(d\_visited, d\_adj\_list, d\_stack, d\_stack\_top, v);

cudaDeviceSynchronize();

delete[] visited;

delete stack\_top;

}

void Graph::BFS(int v) {

visited = new bool[V];

memset(visited, false, V \* sizeof(bool));

cudaMemcpy(d\_adj\_list, &adj\_list[0], sizeof(int) \* adj\_list.size(), cudaMemcpyHostToDevice);

cudaMemcpy(d\_visited, visited, sizeof(bool) \* V, cudaMemcpyHostToDevice);

int\* queue\_front = new int;

int\* queue\_rear = new int;

\*queue\_front = 0;

\*queue\_rear = -1;

cudaMemcpy(d\_queue\_front, queue\_front, sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_queue\_rear, queue\_rear, sizeof(int), cudaMemcpyHostToDevice);

bfs\_kernel<<<1, V>>>(d\_visited, d\_adj\_list, d\_queue, d\_queue\_front, d\_queue\_rear, v);

cudaDeviceSynchronize();

delete[] visited;

delete queue\_front;

delete queue\_rear;

}

void Graph::cleanup() {

cudaFree(d\_adj\_list);

cudaFree(d\_visited);

cudaFree(d\_stack);

cudaFree(d\_stack\_top);

cudaFree(d\_queue);

cudaFree(d\_queue\_front);

cudaFree(d\_queue\_rear);

}

int main() {

Graph g(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

cout << "DFS traversal starting from vertex 0: ";

g.DFS(0);

cout << endl;

cout << "BFS traversal starting from vertex 0: ";

g.BFS(0);

cout << endl;

g.cleanup();

return 0;

}

Assign 1 in openmp

#include<iostream>

#include<omp.h>

#include<vector>

#include<queue>

#include<stack>

#include <cstdlib>

#include <ctime>

#define max\_nodes 100

using namespace std;

class Graph{

    int numnodes;

    vector<vector<int>> adj;

public:

    Graph(int x) : numnodes(x), adj(x) {}

    void addedge(int u,int v){

        adj[u].push\_back(v);

        adj[v].push\_back(u);

        return;

    }

    void BFS(int start){

        cout<<"BFS ";

        vector<bool> visited(numnodes,false);

        queue<int> q;

        visited[start] = true;

        q.push(start);

        #pragma omp parallel

        {

            while(true) {

                int u = -1; // Initialize u to an invalid value

                #pragma omp critical (QueuePop)

                {

                    if (!q.empty()) {

                        u = q.front();

                        q.pop();

                    }

                }

                if (u == -1) break; // Exit if the queue is empty

                for (int v: adj[u]){

                    if (!visited[v]){

                        #pragma omp parallel

                        visited[v] = true;

                        #pragma omp critical (QueuePush)

                        {

                            q.push(v);

                        }

                    }

                }

            }

        }

        cout << "BFS from vertex " << start << ":\n";

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

            if (visited[i]) cout << i << " ";

        }

        cout << endl;

    }

    void DFS(int start){

        cout<<"DFS ";

        vector<bool> visited(numnodes,false);

        stack<int> s;

        visited[start] = true;

        s.push(start);

        #pragma omp parallel

        {

            while(true){

                int u = -1; // Initialize u to an invalid value

                #pragma omp critical (StackPop)

                {

                    if (!s.empty()){

                        u = s.top();

                        s.pop();

                    }

                }

                if (u == -1) break; // Exit if the stack is empty

                for(int v: adj[u]){

                    if(!visited[v]){

                        #pragma omp parallel

                        visited[v] = true;

                        #pragma omp critical (StackPush)

                        {

                            s.push(v);

                        }

                    }

                }

            }

        }

        cout << "DFS from vertex " << start << ":\n";

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

            if (visited[i]) cout << i << " ";

        }

        cout << endl;

    }

};

int main() {

    srand(time(NULL));

    Graph g(10);

    for (int i = 0; i < 10; ++i) {

        int n = rand() % 50 + 1;

        int m = rand() % 50 + 1;

        g.addedge(n,m);

    }

    int start = rand() % 50 + 1;

    cout << "Parallel BFS:\n";

    g.BFS(start);

    cout << "\nParallel DFS:\n";

    g.DFS(start);

    return 0;

}

Assign 2 in openmp

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <omp.h>

// Sequential Bubble Sort

void bubbleSort(int arr[], int n) {

    for (int i = 0; i < n-1; i++) {

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

            if (arr[j] > arr[j+1]) {

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

            }

        }

    }

}

// Parallel Bubble Sort

void parallelBubbleSort(int arr[], int n) {

    #pragma omp parallel

    {

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

            #pragma omp for

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

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

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

                }

            }

        }

    }

}

// Merge function for merge sort

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

    int n1 = m - l + 1;

    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    int i = 0;

    int j = 0;

    int k = l;

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

// Sequential Merge Sort

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

    if (l >= r) return;

    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);

    mergeSort(arr, m + 1, r);

    merge(arr, l, m, r);

}

// Parallel Merge Sort

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

    if (l >= r) return;

    int m = l + (r - l) / 2;

    #pragma omp parallel sections

    {

        #pragma omp section

        parallelMergeSort(arr, l, m);

        #pragma omp section

        parallelMergeSort(arr, m + 1, r);

    }

    merge(arr, l, m, r);

}

// Print array

void printArray(int arr[], int size) {

    for (int i = 0; i < size; i++) {

        std::cout << arr[i] << " ";

    }

    std::cout << std::endl;

}

int main() {

    const int SIZE = 10;

    int arr1[SIZE], arr2[SIZE];

    // Initialize arrays with random values

    srand(time(NULL));

    for (int i = 0; i < SIZE; i++) {

        arr1[i] = rand() % 100;

        arr2[i] = arr1[i];

    }

    std::cout << "Original Array: ";

    printArray(arr1, SIZE);

    // Sequential Bubble Sort

    bubbleSort(arr1, SIZE);

    std::cout << "Sequential Bubble Sort: ";

    printArray(arr1, SIZE);

    // Parallel Bubble Sort

    parallelBubbleSort(arr2, SIZE);

    std::cout << "Parallel Bubble Sort: ";

    printArray(arr2, SIZE);

    // Reset arrays

    for (int i = 0; i < SIZE; i++) {

        arr1[i] = arr2[i] = rand() % 100;

    }

    std::cout << "Original Array: ";

    printArray(arr1, SIZE);

    // Sequential Merge Sort

    mergeSort(arr1, 0, SIZE - 1);

    std::cout << "Sequential Merge Sort: ";

    printArray(arr1, SIZE);

    // Parallel Merge Sort

    parallelMergeSort(arr2, 0, SIZE - 1);

    std::cout << "Parallel Merge Sort: ";

    printArray(arr2, SIZE);

    return 0;

}

Assign 2 in cuda

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cuda\_runtime.h>

#define ARRAY\_SIZE 10000

#define BLOCK\_SIZE 256

// CUDA kernel for parallel Bubble Sort

\_\_global\_\_ void bubbleSort(int \*arr, int size) {

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

    if (idx < size) {

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

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

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

                    int temp = arr[j];

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

                    arr[j + 1] = temp;

                }

            }

        }

    }

}

// CUDA kernel for parallel Merge Sort (split and merge phases)

\_\_global\_\_ void mergeSort(int \*arr, int \*temp, int left, int right) {

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

    if (idx < right - left) {

        int mid = left + (right - left) / 2;

        int i = left, j = mid + 1, k = left;

        // Merge two sorted subarrays into temp array

        while (i <= mid && j <= right) {

            if (arr[i] <= arr[j])

                temp[k++] = arr[i++];

            else

                temp[k++] = arr[j++];

        }

        while (i <= mid)

            temp[k++] = arr[i++];

        while (j <= right)

            temp[k++] = arr[j++];

        // Copy temp array back to original array

        for (int i = left; i <= right; ++i)

            arr[i] = temp[i];

    }

}

// Function to initialize array with random values

void initializeArray(int \*arr, int size) {

    srand(time(NULL));

    for (int i = 0; i < size; ++i)

        arr[i] = rand() % 1000; // Generate random values between 0 and 999

}

int main() {

    // Initialize array with random values

    int \*arr = new int[ARRAY\_SIZE];

    initializeArray(arr, ARRAY\_SIZE);

    // Sequential Bubble Sort

    clock\_t start = clock();

    // Call sequential Bubble Sort function here

    clock\_t end = clock();

    double seqTime = double(end - start) / CLOCKS\_PER\_SEC;

    // Allocate memory on device

    int \*d\_arr, \*d\_temp;

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

    cudaMalloc((void\*\*)&d\_temp, ARRAY\_SIZE \* sizeof(int));

    // Copy data from host to device

    cudaMemcpy(d\_arr, arr, ARRAY\_SIZE \* sizeof(int), cudaMemcpyHostToDevice);

    // Parallel Bubble Sort

    start = clock();

    bubbleSort<<<(ARRAY\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE, BLOCK\_SIZE>>>(d\_arr, ARRAY\_SIZE);

    cudaDeviceSynchronize();

    end = clock();

    double parallelBubbleTime = double(end - start) / CLOCKS\_PER\_SEC;

    // Parallel Merge Sort

    start = clock();

    mergeSort<<<(ARRAY\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE, BLOCK\_SIZE>>>(d\_arr, d\_temp, 0, ARRAY\_SIZE - 1);

    cudaDeviceSynchronize();

    end = clock();

    double parallelMergeTime = double(end - start) / CLOCKS\_PER\_SEC;

    // Copy sorted array from device to host

    cudaMemcpy(arr, d\_arr, ARRAY\_SIZE \* sizeof(int), cudaMemcpyDeviceToHost);

    // Free device memory

    cudaFree(d\_arr);

    cudaFree(d\_temp);

    // Output results

    std::cout << "Sequential Bubble Sort Time: " << seqTime << " seconds" << std::endl;

    std::cout << "Parallel Bubble Sort Time: " << parallelBubbleTime << " seconds" << std::endl;

    std::cout << "Parallel Merge Sort Time: " << parallelMergeTime << " seconds" << std::endl;

    delete[] arr;

    return 0;

}

Assign 3 in cuda

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cuda\_runtime.h>

#define ARRAY\_SIZE 10000

#define BLOCK\_SIZE 256

// CUDA kernel for parallel reduction to find minimum value

\_\_global\_\_ void minReduction(int \*arr, int \*result, int size) {

    \_\_shared\_\_ int sharedArr[BLOCK\_SIZE];

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

    int tid = threadIdx.x;

    if (idx < size)

        sharedArr[tid] = arr[idx];

    else

        sharedArr[tid] = INT\_MAX;

    \_\_syncthreads();

    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {

        if (tid < stride && idx + stride < size) {

            if (sharedArr[tid] > sharedArr[tid + stride])

                sharedArr[tid] = sharedArr[tid + stride];

        }

        \_\_syncthreads();

    }

    if (tid == 0)

        result[blockIdx.x] = sharedArr[0];

}

// CUDA kernel for parallel reduction to find maximum value

\_\_global\_\_ void maxReduction(int \*arr, int \*result, int size) {

    \_\_shared\_\_ int sharedArr[BLOCK\_SIZE];

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

    int tid = threadIdx.x;

    if (idx < size)

        sharedArr[tid] = arr[idx];

    else

        sharedArr[tid] = INT\_MIN;

    \_\_syncthreads();

    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {

        if (tid < stride && idx + stride < size) {

            if (sharedArr[tid] < sharedArr[tid + stride])

                sharedArr[tid] = sharedArr[tid + stride];

        }

        \_\_syncthreads();

    }

    if (tid == 0)

        result[blockIdx.x] = sharedArr[0];

}

// CUDA kernel for parallel reduction to find sum

\_\_global\_\_ void sumReduction(int \*arr, int \*result, int size) {

    \_\_shared\_\_ int sharedArr[BLOCK\_SIZE];

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

    int tid = threadIdx.x;

    if (idx < size)

        sharedArr[tid] = arr[idx];

    else

        sharedArr[tid] = 0;

    \_\_syncthreads();

    for (unsigned int stride = blockDim.x / 2; stride > 0; stride >>= 1) {

        if (tid < stride && idx + stride < size)

            sharedArr[tid] += sharedArr[tid + stride];

        \_\_syncthreads();

    }

    if (tid == 0)

        result[blockIdx.x] = sharedArr[0];

}

int main() {

    int \*arr = new int[ARRAY\_SIZE];

    srand(time(NULL));

    for (int i = 0; i < ARRAY\_SIZE; ++i)

        arr[i] = rand() % 1000; // Generate random values between 0 and 999

    int \*d\_arr, \*d\_result;

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

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

    cudaMemcpy(d\_arr, arr, ARRAY\_SIZE \* sizeof(int), cudaMemcpyHostToDevice);

    // Finding minimum value

    minReduction<<<(ARRAY\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE, BLOCK\_SIZE>>>(d\_arr, d\_result, ARRAY\_SIZE);

    int minVal;

    cudaMemcpy(&minVal, d\_result, sizeof(int), cudaMemcpyDeviceToHost);

    std::cout << "Minimum value: " << minVal << std::endl;

    // Finding maximum value

    maxReduction<<<(ARRAY\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE, BLOCK\_SIZE>>>(d\_arr, d\_result, ARRAY\_SIZE);

    int maxVal;

    cudaMemcpy(&maxVal, d\_result, sizeof(int), cudaMemcpyDeviceToHost);

    std::cout << "Maximum value: " << maxVal << std::endl;

    // Finding sum

    sumReduction<<<(ARRAY\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE, BLOCK\_SIZE>>>(d\_arr, d\_result, ARRAY\_SIZE);

    int sumVal;

    cudaMemcpy(&sumVal, d\_result, sizeof(int), cudaMemcpyDeviceToHost);

    std::cout << "Sum: " << sumVal << std::endl;

    // Finding average

    float avgVal = static\_cast<float>(sumVal) / ARRAY\_SIZE;

    std::cout << "Average: " << avgVal << std::endl;

    cudaFree(d\_arr);

    cudaFree(d\_result);

    delete[] arr;

    return 0;

}

Assign 4 in cuda

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cuda\_runtime.h>

#define ARRAY\_SIZE 1000000

#define BLOCK\_SIZE 256

// CUDA kernel for vector addition

\_\_global\_\_ void vectorAdd(int \*a, int \*b, int \*c, int size) {

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

    if (idx < size)

        c[idx] = a[idx] + b[idx];

}

int main() {

    int \*a = new int[ARRAY\_SIZE];

    int \*b = new int[ARRAY\_SIZE];

    int \*c = new int[ARRAY\_SIZE];

    srand(time(NULL));

    for (int i = 0; i < ARRAY\_SIZE; ++i) {

        a[i] = rand() % 1000;

        b[i] = rand() % 1000;

    }

    int \*d\_a, \*d\_b, \*d\_c;

    cudaMalloc((void\*\*)&d\_a, ARRAY\_SIZE \* sizeof(int));

    cudaMalloc((void\*\*)&d\_b, ARRAY\_SIZE \* sizeof(int));

    cudaMalloc((void\*\*)&d\_c, ARRAY\_SIZE \* sizeof(int));

    cudaMemcpy(d\_a, a, ARRAY\_SIZE \* sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_b, b, ARRAY\_SIZE \* sizeof(int), cudaMemcpyHostToDevice);

    vectorAdd<<<(ARRAY\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE, BLOCK\_SIZE>>>(d\_a, d\_b, d\_c, ARRAY\_SIZE);

    cudaDeviceSynchronize();

    cudaMemcpy(c, d\_c, ARRAY\_SIZE \* sizeof(int), cudaMemcpyDeviceToHost);

    for (int i = 0; i < ARRAY\_SIZE; ++i)

         std::cout << c[i] << " ";

    std::cout << std::endl;

    cudaFree(d\_a);

    cudaFree(d\_b);

    cudaFree(d\_c);

    delete[] a;

    delete[] b;

    delete[] c;

    return 0;

}

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cuda\_runtime.h>

#define MATRIX\_SIZE 64

#define BLOCK\_SIZE 16

// CUDA kernel for matrix multiplication

\_\_global\_\_ void matrixMul(int \*a, int \*b, int \*c, int size) {

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

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

    if (row < size && col < size) {

        int sum = 0;

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

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

        c[row \* size + col] = sum;

    }

}

int main() {

    int \*a = new int[MATRIX\_SIZE \* MATRIX\_SIZE];

    int \*b = new int[MATRIX\_SIZE \* MATRIX\_SIZE];

    int \*c = new int[MATRIX\_SIZE \* MATRIX\_SIZE];

    srand(time(NULL));

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

        a[i] = rand() % 100;

        b[i] = rand() % 100;

    }

    int \*d\_a, \*d\_b, \*d\_c;

    cudaMalloc((void\*\*)&d\_a, MATRIX\_SIZE \* MATRIX\_SIZE \* sizeof(int));

    cudaMalloc((void\*\*)&d\_b, MATRIX\_SIZE \* MATRIX\_SIZE \* sizeof(int));

    cudaMalloc((void\*\*)&d\_c, MATRIX\_SIZE \* MATRIX\_SIZE \* sizeof(int));

    cudaMemcpy(d\_a, a, MATRIX\_SIZE \* MATRIX\_SIZE \* sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_b, b, MATRIX\_SIZE \* MATRIX\_SIZE \* sizeof(int), cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(BLOCK\_SIZE, BLOCK\_SIZE);

    dim3 numBlocks((MATRIX\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE, (MATRIX\_SIZE + BLOCK\_SIZE - 1) / BLOCK\_SIZE);

    matrixMul<<<numBlocks, threadsPerBlock>>>(d\_a, d\_b, d\_c, MATRIX\_SIZE);

    cudaDeviceSynchronize();

    cudaMemcpy(c, d\_c, MATRIX\_SIZE \* MATRIX\_SIZE \* sizeof(int), cudaMemcpyDeviceToHost);

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

        std::cout << c[i] << " ";

        if ((i + 1) % MATRIX\_SIZE == 0)

            std::cout << std::endl;

    }

    cudaFree(d\_a);

    cudaFree(d\_b);

    cudaFree(d\_c);

    delete[] a;

    delete[] b;

    delete[] c;

    return 0;

}