STARTING COMMANDS FOR CUDA :

```
# Check CUDA version
!nvcc --version

# Install CUDA package
!pip install git+https://github.com/afnan47/cuda.git

# Load nvcc plugin
%load_ext nvcc_plugin
```

OUTPUT COMMANDS FOR CUDA :

```
!nvcc filename.cu -o filename
!./filename
```

COMMANDS FOR TERMINAL :

1. cat filename.cpp

2. g++ -o filename –fopenmp filename.cpp

3. ./filename

4. g++ filename.cpp –lgomp -o filename

// BFS AND DFS USING OPENMP

CODE 1 (GRAPH) :

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>

using namespace std;

// Graph class representing the adjacency list
class Graph {
    int V;  // Number of vertices
    vector<vector<int>> adj;  // Adjacency list
```

```
public:
    Graph(int V) : V(V), adj(V) {}

    // Add an edge to the graph
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    // Parallel Depth-First Search
    void parallelDFS(int startVertex) {
        vector<bool> visited(V, false);
        parallelDFSUtil(startVertex, visited);
    }

    // Parallel DFS utility function
    void parallelDFSUtil(int v, vector<bool>& visited) {
        visited[v] = true;
        cout << v << " ";

        #pragma omp parallel for
        for (int i = 0; i < adj[v].size(); ++i) {
            int n = adj[v][i];
            if (!visited[n])
                parallelDFSUtil(n, visited);
        }
    }

    // Parallel Breadth-First Search
    void parallelBFS(int startVertex) {
        vector<bool> visited(V, false);
        queue<int> q;

        visited[startVertex] = true;
        q.push(startVertex);

        while (!q.empty()) {
            int v = q.front();
            q.pop();
            cout << v << " ";

            #pragma omp parallel for
            for (int i = 0; i < adj[v].size(); ++i) {
                int n = adj[v][i];
                if (!visited[n]) {
                    visited[n] = true;
                    q.push(n);
                }
            }
        }
```

```
    }
};

int main() {
    // Create a graph
    Graph g(6);
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);
    g.addEdge(4, 5);
    g.addEdge(5, 3);


    cout << "Depth-First Search (DFS): ";
    g.parallelDFS(0);
    cout << endl;

    cout << "Breadth-First Search (BFS): ";
    g.parallelBFS(0);
    cout << endl;

    return 0;
}
```

CODE 2 (USER INPUT) (GRAPH) :

```
#include <iostream>
#include <vector>
#include <queue>
#include <omp.h>

using namespace std;

// Graph class representing the adjacency list
class Graph {
    int V;  // Number of vertices
    vector<vector<int>> adj;  // Adjacency list

public:
    Graph(int V) : V(V), adj(V) {}

    // Add an edge to the graph
    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }
```

```
    // Parallel Depth-First Search
    void parallelDFS(int startVertex) {
        vector<bool> visited(V, false);
        parallelDFSUtil(startVertex, visited);
    }

    // Parallel DFS utility function
    void parallelDFSUtil(int v, vector<bool>& visited) {
        visited[v] = true;
        cout << v << " ";

        #pragma omp parallel for
        for (int i = 0; i < adj[v].size(); ++i) {
            int n = adj[v][i];
            if (!visited[n])
                parallelDFSUtil(n, visited);
        }
    }

    // Parallel Breadth-First Search
    void parallelBFS(int startVertex) {
        vector<bool> visited(V, false);
        queue<int> q;

        visited[startVertex] = true;
        q.push(startVertex);

        while (!q.empty()) {
            int v = q.front();
            q.pop();
            cout << v << " ";

            #pragma omp parallel for
            for (int i = 0; i < adj[v].size(); ++i) {
                int n = adj[v][i];
                if (!visited[n]) {
                    visited[n] = true;
                    q.push(n);
                }
            }
        }
    }
};

int main() {
    int V, E; // Number of vertices and edges
    cout << "Enter the number of vertices: ";
    cin >> V;
    cout << "Enter the number of edges: ";
    cin >> E;
```

```
    // Create a graph
    Graph g(V);
    cout << "Enter edges (vertex1 vertex2):" << endl;
    for (int i = 0; i < E; ++i) {
        int v, w;
        cin >> v >> w;
        g.addEdge(v, w);
    }

    cout << "Depth-First Search (DFS): ";
    g.parallelDFS(0);
    cout << endl;

    cout << "Breadth-First Search (BFS): ";
    g.parallelBFS(0);
    cout << endl;

    return 0;
}
```

CODE 3 (GRAPH AND TREE) :

DFS

```
 #include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

const int MAXN = 1e5;
vector<int> adj[MAXN + 5]; // adjacency list
bool visited[MAXN + 5];     // mark visited nodes

void dfs(int node)
{
    visited[node] = true;
    cout << node << " "; // Print the visited node here
#pragma omp parallel for
    for (int i = 0; i < adj[node].size(); i++)
    {
        int next_node = adj[node][i];
        if (!visited[next_node])
        {
            dfs(next_node);
        }
    }
}
```

```
int main()
{
    cout << "Please enter nodes and edges: ";
    int n, m; // number of nodes and edges
    cin >> n >> m;
    for (int i = 1; i <= m; i++)
    {
        int u, v; // edge between u and v
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int start_node; // start node of DFS
    cout << "Enter the start node for DFS: ";
    cin >> start_node;
    dfs(start_node);
    cout << endl; // Print a newline after DFS traversal
    return 0;
}
```

BFS

```
#include <iostream>
#include <queue>
#include <vector>
#include <omp.h>

using namespace std;

int main() {
    int num_vertices, num_edges, source;
    cout << "Enter number of vertices, edges, and source node: ";
    cin >> num_vertices >> num_edges >> source;

    // Input validation
    if (source < 1 || source > num_vertices) {
        cout << "Invalid source node!" << endl;
        return 1;
    }

    vector<vector<int>> adj_list(num_vertices + 1);
    for (int i = 0; i < num_edges; i++) {
        int u, v;
        cin >> u >> v;
        // Input validation for edges
```

```
        if (u < 1 || u > num_vertices || v < 1 || v > num_vertices) {
            cout << "Invalid edge: " << u << " " << v << endl;
            return 1;
        }
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }

    queue<int> q;
    vector<bool> visited(num_vertices + 1, false);
    q.push(source);
    visited[source] = true;

    while (!q.empty()) {
        int curr_vertex = q.front();
        q.pop();
        cout << curr_vertex << " ";

        // Parallel loop for neighbors
#pragma omp parallel for
        for (int i = 0; i < adj_list[curr_vertex].size(); i++) {
            int neighbour = adj_list[curr_vertex][i];
            if (!visited[neighbour]) {
                visited[neighbour] = true;
                q.push(neighbour);
            }
        }
    }

    cout << endl;
    return 0;
}
```

// BFS AND DFS USING CUDA

CODE 1 BFS :

```
%%writefile breadthfirst.cu
#include <iostream>
#include <queue>
```

```
#include <vector>
#include <omp.h>

using namespace std;

int main() {
    int num_vertices, num_edges, source;
    cout << "Enter number of vertices, edges, and source node: ";
    cin >> num_vertices >> num_edges >> source;

    // Input validation
    if (source < 1 || source > num_vertices) {
        cout << "Invalid source node!" << endl;
        return 1;
    }

    vector<vector<int>> adj_list(num_vertices + 1);
    for (int i = 0; i < num_edges; i++) {
        int u, v;
        cin >> u >> v;
        // Input validation for edges
        if (u < 1 || u > num_vertices || v < 1 || v > num_vertices) {
            cout << "Invalid edge: " << u << " " << v << endl;
            return 1;
        }
        adj_list[u].push_back(v);
        adj_list[v].push_back(u);
    }

    queue<int> q;
    vector<bool> visited(num_vertices + 1, false);
    q.push(source);
    visited[source] = true;

    while (!q.empty()) {
        int curr_vertex = q.front();
        q.pop();
        cout << curr_vertex << " ";

        // Sequential loop for neighbors
        for (int i = 0; i < adj_list[curr_vertex].size(); i++) {
            int neighbour = adj_list[curr_vertex][i];
            if (!visited[neighbour]) {
                visited[neighbour] = true;
                q.push(neighbour);
```

```
            }
        }
    }

    cout << endl;
    return 0;
}
```

```
!nvcc breadthfirst.cu -o breadthfirst
!./breadthfirst
```

OUTPUT :

```
Enter number of vertices, edges, and source node: 6 5 3
3 2
3 5
2 1
5 4
5 6
3 2 5 1 4 6
```

CODE 2 DFS :

```
%%writefile depthfirst.cu
#include <iostream>
#include <vector>
using namespace std;
const int MAXN = 1e5;
vector<int> adj[MAXN+5]; // adjacency list
bool visited[MAXN+5]; // mark visited nodes

void dfs(int node) {
    visited[node] = true;
    cout << node << " "; // Print the visited node here
    for (int i = 0; i < adj[node].size(); i++) {
        int next_node = adj[node][i];
        if (!visited[next_node]) {
            dfs(next_node);
        }
    }
}

int main() {
    cout << "Please enter nodes and edges: ";
    int n, m; // number of nodes and edges
    cin >> n >> m;
    for (int i = 1; i <= m; i++) {
```

```
        int u, v; // edge between u and v
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int start_node; // start node of DFS
    cout << "Enter the start node for DFS: ";
    cin >> start_node;
    dfs(start_node);
    cout << endl; // Print a newline after DFS traversal
    return 0;
}
```

```
!nvcc depthfirst.cu -o depthfirst
!./depthfirst
```

OUTPUT :

```
Please enter nodes and edges: 5 4
1 2
1 3
2 4
3 5
Enter the start node for DFS: 1
1 2 4 3 5
```

// Bubble Sort AND Merge Sort USING OPEN MP

CODE 1 - BUBBLE SORT :

```
#include <iostream>
#include <vector>
#include <chrono>    // For std::chrono
#include <omp.h>    // OpenMP

// Sequential Bubble Sort
void bubbleSortSequential(int* arr, int 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;
            }
        }
    }
}
```

```
// Parallel Bubble Sort using OpenMP
void bubbleSortParallel(int* arr, int size) {
    #pragma omp parallel for
    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;
            }
        }
    }
}

int main() {
    int size;
    std::cout << "Enter the number of integers: ";
    std::cin >> size;

    int arr[size];

    // Taking input from the user
    std::cout << "Enter " << size << " integers:\n";
    for (int i = 0; i < size; i++) {
        std::cin >> arr[i];
    }

    // Measure sequential bubble sort time
    auto startSeqBubble = std::chrono::steady_clock::now();
    bubbleSortSequential(arr, size);
    auto endSeqBubble = std::chrono::steady_clock::now();
    double timeSeqBubble = std::chrono::duration<double>(endSeqBubble -
startSeqBubble).count();

    // Measure parallel bubble sort time
    auto startParBubble = std::chrono::steady_clock::now();
    bubbleSortParallel(arr, size);
    auto endParBubble = std::chrono::steady_clock::now();
    double timeParBubble = std::chrono::duration<double>(endParBubble -
startParBubble).count();

    // Print sorted array
    std::cout << "Sorted array:\n";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    // Print execution times
    std::cout << "Sequential Bubble Sort Time: " << timeSeqBubble << " seconds\n";
```

```
    std::cout << "Parallel Bubble Sort Time: " << timeParBubble << " seconds\n";

    return 0;
}
```

CODE 2 - MERGE SORT :

// Bubble Sort AND Merge Sort USING CUDA

CODE 1 - BUBBLE SORT :

```
%%writefile bubble.cu
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

__device__ void device_swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

__global__ void kernel_bubble_sort_odd_even(int* arr, int size) {
    bool isSorted = false;
    while (!isSorted) {
        isSorted = true;
        int tid = blockIdx.x * blockDim.x + threadIdx.x; //calculating
gloable thread id.
        if (tid % 2 == 0 && tid < size - 1) {
            if (arr[tid] > arr[tid + 1]) {
                device_swap(arr[tid], arr[tid + 1]);
                isSorted = false;
            }
        }
        __syncthreads(); // Synchronize threads within block

        if (tid % 2 != 0 && tid < size - 1) {
            if (arr[tid] > arr[tid + 1]) {
                device_swap(arr[tid], arr[tid + 1]);
                isSorted = false;
            }
        }
```

```
        __syncthreads(); // Synchronize threads within block
    }
}

void bubble_sort_odd_even(vector<int>& arr) {
    int size = arr.size();
    int* d_arr;
    cudaMalloc(&d_arr, size * sizeof(int));
    cudaMemcpy(d_arr, arr.data(), size * sizeof(int),
cudaMemcpyHostToDevice);

    // Calculate grid and block dimensions
    int blockSize = 256;
    int gridSize = (size + blockSize - 1) / blockSize;

    // Perform bubble sort on GPU
    kernel_bubble_sort_odd_even<<<gridSize, blockSize>>>(d_arr, size);

    // Copy sorted array back to host
    cudaMemcpy(arr.data(), d_arr, size * sizeof(int),
cudaMemcpyDeviceToHost);
    cout<<"sorted array"<<endl;
    for(int i=0;i<size;i++){
      cout<<arr[i]<<" ";
    }
    cout<<endl;
    cudaFree(d_arr);
}

int main() {
    vector<int> arr = {5,4 , 3,2 ,1 ,0,6,9,7 };
    double start, end;

    // Measure performance of parallel bubble sort using odd-even
transposition
    start =
chrono::duration_cast<chrono::milliseconds>(chrono::system_clock::now().time
_since_epoch()).count();
    bubble_sort_odd_even(arr);
    end =
chrono::duration_cast<chrono::milliseconds>(chrono::system_clock::now().time
_since_epoch()).count();

    cout << "Parallel bubble sort using odd-even transposition time: " <<
end - start << " milliseconds" << endl;
```

```
    return 0;
}
```

```
!nvcc bubble.cu -o bubble
!./bubble
```

OUTPUT :

```
sorted array
0 1 2 3 4 5 6 7 9
Parallel bubble sort using odd-even transposition time: 101 milliseconds
```

CODE 2 - MERGE SORT :

```
%%writefile merge_sort.cu
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm> // for min function
using namespace std;

// Kernel to merge two sorted halves
__global__ void kernel_merge(int* arr, int* temp, int* subarray_sizes, int
array_size) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;//calculating global
thread id
    int left_start = idx * 2 * (*subarray_sizes);

    if (left_start < array_size) {
        int mid = min(left_start + (*subarray_sizes) - 1, array_size - 1);
        int right_end = min(left_start + 2 * (*subarray_sizes) - 1,
array_size - 1);

        int i = left_start;
        int j = mid + 1;
        int k = left_start;

        // Merge process
        while (i <= mid && j <= right_end) {
            if (arr[i] <= arr[j]) {
                temp[k] = arr[i];
                i++;
```

```
            } else {
                temp[k] = arr[j];
                j++;
            }
            k++;
        }

        while (i <= mid) {
            temp[k] = arr[i];
            i++;
            k++;
        }

        while (j <= right_end) {
            temp[k] = arr[j];
            j++;
            k++;
        }

        // Copy the sorted subarray back to the original array
        for (int t = left_start; t <= right_end; t++) {
            arr[t] = temp[t];
        }
    }
}

void merge_sort(vector<int>& arr) {
    int array_size = arr.size();
    int* d_arr;
    int* d_temp;
    int* d_subarray_size;

    // Allocate memory on the GPU
    cudaMalloc(&d_arr, array_size * sizeof(int));
    cudaMalloc(&d_temp, array_size * sizeof(int));
    cudaMalloc(&d_subarray_size, sizeof(int)); // Holds the subarray size
for each step

    cudaMemcpy(d_arr, arr.data(), array_size * sizeof(int),
cudaMemcpyHostToDevice);

    int blockSize = 256; // Threads per block
    int gridSize;        // Number of blocks in the grid, depending on the
subarray size

```

```
    // Start with width of 1, then double each iteration
    int width = 1;
    while (width < array_size) {
        cudaMemcpy(d_subarray_size, &width, sizeof(int),
cudaMemcpyHostToDevice);

        gridSize = (array_size / (2 * width)) + 1;

        kernel_merge<<<gridSize, blockSize>>>(d_arr, d_temp,
d_subarray_size, array_size);
        cudaDeviceSynchronize(); // Ensure all threads finish before the
next step

        // Double the subarray width for the next iteration
        width *= 2;
    }

    // Copy the sorted array back to the host
    cudaMemcpy(arr.data(), d_arr, array_size * sizeof(int),
cudaMemcpyDeviceToHost);

    // Free GPU memory
    cudaFree(d_arr);
    cudaFree(d_temp);
    cudaFree(d_subarray_size);
}

int main() {
    vector<int> arr = {6, 5, 4, 1, 7, 9, 8, 3, 2};
    double start, end;

    start =
chrono::duration_cast<chrono::milliseconds>(chrono::system_clock::now().time
_since_epoch()).count();
    merge_sort(arr);
    end =
chrono::duration_cast<chrono::milliseconds>(chrono::system_clock::now().time
_since_epoch()).count();

    cout << "Parallel merge sort time: " << end - start << " milliseconds"
<< endl;
    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
```

```
    cout << endl;

    return 0;
}
```

```
!nvcc merge_sort.cu -o merge
!./merge
```

OUTPUT :

```
Parallel merge sort time: 199 milliseconds
Sorted array: 1 2 3 4 5 6 7 8 9
```

// Parallel Reduction USING CUDA

```
%%writefile sum.cu
#include <iostream>
#include <vector>
#include <climits>

__global__ void min_reduction_kernel(int* arr, int size, int* result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        atomicMin(result, arr[tid]);
    }
}

__global__ void max_reduction_kernel(int* arr, int size, int* result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        atomicMax(result, arr[tid]);
    }
}

__global__ void sum_reduction_kernel(int* arr, int size, int* result) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        atomicAdd(result, arr[tid]);
    }
}
```

```
__global__ void average_reduction_kernel(int* arr, int size, int* sum) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        atomicAdd(sum, arr[tid]);
    }
}

int main() {
    int size;
    std::cout << "Enter the size of the array: ";
    std::cin >> size;

    std::vector<int> arr(size);
    for (int i = 0; i < size; ++i) {
        std::cout << "Enter element " << i << ": ";
        std::cin >> arr[i];
    }

    int* d_arr;
    int* d_result;
    int result_min = INT_MAX;
    int result_max = INT_MIN;
    int result_sum = 0;

    // Allocate memory on the device
    cudaMalloc(&d_arr, size * sizeof(int));
    cudaMalloc(&d_result, sizeof(int));

    // Copy data from host to device
    cudaMemcpy(d_arr, arr.data(), size * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_result, &result_min, sizeof(int), cudaMemcpyHostToDevice);

    // Perform min reduction
    min_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size,
d_result);
    cudaMemcpy(&result_min, d_result, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Minimum value: " << result_min << std::endl;

    // Perform max reduction
    cudaMemcpy(d_result, &result_max, sizeof(int), cudaMemcpyHostToDevice);
    max_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size,
d_result);
    cudaMemcpy(&result_max, d_result, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Maximum value: " << result_max << std::endl;
```

```
    // Perform sum reduction
    cudaMemcpy(d_result, &result_sum, sizeof(int), cudaMemcpyHostToDevice);
    sum_reduction_kernel<<<(size + 255) / 256, 256>>>(d_arr, size,
d_result);
    cudaMemcpy(&result_sum, d_result, sizeof(int), cudaMemcpyDeviceToHost);
    std::cout << "Sum: " << result_sum << std::endl;

    // Perform average reduction on CPU side
    double average = static_cast<double>(result_sum) / size;
    std::cout << "Average: " << average << std::endl;

    // Free device memory
    cudaFree(d_arr);
    cudaFree(d_result);

    return 0;
}
```

```
!nvcc sum.cu -o sum
!./sum
```

OUTPUT :

```
Enter the size of the array: 5
Enter element 0: 5
Enter element 1: 4
Enter element 2: 8
Enter element 3: 6
Enter element 4: 3
Minimum value: 3
Maximum value: 8
Sum: 26
Average: 5.2
```

// Parallel Reduction USING OPENMP

#include <iostream>
#include <vector>
#include <omp.h>

```cpp
#include <climits>

using namespace std;

void min_reduction(vector<int>& arr) {
    int min_value = INT_MAX;
#pragma omp parallel for reduction(min: min_value)
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] < min_value) {
            min_value = arr[i];
        }
    }
    cout << "Minimum value: " << min_value << endl;
}

void max_reduction(vector<int>& arr) {
    int max_value = INT_MIN;
#pragma omp parallel for reduction(max: max_value)
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }
    cout << "Maximum value: " << max_value << endl;
}

void sum_reduction(vector<int>& arr) {
    int sum = 0;
#pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];
    }
    cout << "Sum: " << sum << endl;
}

void average_reduction(vector<int>& arr) {
    int sum = 0;
#pragma omp parallel for reduction(+: sum)
    for (int i = 0; i < arr.size(); i++) {
        sum += arr[i];
    }
    cout << "Average: " << (double)sum / arr.size() << endl;
}

int main() {
    int n;
```

```
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; ++i) {
        cin >> arr[i];
    }

    min_reduction(arr);
    max_reduction(arr);
    sum_reduction(arr);
    average_reduction(arr);

    return 0;
}
```

OUTPUT :

```
Enter the number of elements: 5
Enter 5 elements: 5
4
8
6
3
Minimum value: 3
Maximum value: 8
Sum: 26
Average: 5.2
```

// Addition of Two Large Vectors USING CUDA

```
%%writefile add.cu
#include <iostream>
#include <cstdlib> // Include <cstdlib> for rand()
using namespace std;

__global__
void add(int* A, int* B, int* C, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        C[tid] = A[tid] + B[tid];
    }
}

void print(int* vector, int size) {
    for (int i = 0; i < size; i++) {
        cout << vector[i] << " ";
    }
    cout << endl;
}

int main() {
    int N;
    cout << "Enter the size of the vectors: ";
    cin >> N;

    int* A, * B, * C;
    int vectorSize = N;
    size_t vectorBytes = vectorSize * sizeof(int);

    // Allocate host memory
    A = new int[vectorSize];
    B = new int[vectorSize];
    C = new int[vectorSize];

    // Initialize host arrays
    cout << "Enter elements of vector A:" << endl;
    for (int i = 0; i < N; i++) {
        cin >> A[i];
    }
    cout << "Enter elements of vector B:" << endl;
    for (int i = 0; i < N; i++) {
        cin >> B[i];
    }
    cout << "Vector A: ";
```

```
    print(A, N);
    cout << "Vector B: ";
    print(B, N);

    int* X, * Y, * Z;
    // Allocate device memory
    cudaMalloc(&X, vectorBytes);
    cudaMalloc(&Y, vectorBytes);
    cudaMalloc(&Z, vectorBytes);

    // Check for CUDA memory allocation errors
    if (X == nullptr || Y == nullptr || Z == nullptr) {
        cerr << "CUDA memory allocation failed" << endl;
        return 1;
    }

    // Copy data from host to device
    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(Y, B, vectorBytes, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    // Launch kernel
    add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);

    // Check for kernel launch errors
    cudaError_t kernelLaunchError = cudaGetLastError();
    if (kernelLaunchError != cudaSuccess) {
        cerr << "CUDA kernel launch failed: " <<
cudaGetErrorString(kernelLaunchError) << endl;
        return 1;
    }

    // Copy result from device to host
    cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);

    // Check for CUDA memcpy errors
    cudaError_t memcpyError = cudaGetLastError();
    if (memcpyError != cudaSuccess) {
        cerr << "CUDA memcpy failed: " << cudaGetErrorString(memcpyError) <<
endl;
        return 1;
    }
```

```
    cout << "Addition: ";
    print(C, N);

    // Free device memory
    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    // Free host memory
    delete[] A;
    delete[] B;
    delete[] C;

    return 0;
}
```

```
!nvcc add.cu -o add
!./add
```

OUTPUT :

```
Enter the size of the vectors: 3
Enter elements of vector A:
1 2 3
Enter elements of vector B:
4 5 6
Vector A: 1 2 3
Vector B: 4 5 6
Addition: 5 7 9
```

// Matrix Multiplication USING CUDA C

```
%%writefile matrix_mult.cu
#include <iostream>
#include <cuda.h>
using namespace std;

#define BLOCK_SIZE 1

__global__ void gpuMM(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.f;
    for (int n = 0; n < N; ++n)
        sum += A[row * N + n] * B[n * N + col];
    C[row * N + col] = sum;
}

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

    // Get matrix size from user
    cout << "Enter size of matrix (N): ";
    cin >> N;
    if (N % BLOCK_SIZE != 0) {
        cerr << "Matrix size must be a multiple of BLOCK_SIZE." << endl;
        return 1;
    }

    cout << "\nExecuting Matrix Multiplication" << endl;
    cout << "Matrix size: " << N << "x" << N << endl;

    // Allocate memory for matrices on the host
    float *hA, *hB, *hC;
    hA = new float[N * N];
    hB = new float[N * N];
    hC = new float[N * N];

    // Read matrices from user
    cout << "Enter elements of matrix A (" << N << "x" << N << "):" << endl;
    for (int i = 0; i < N * N; ++i)
        cin >> hA[i];

    cout << "Enter elements of matrix B (" << N << "x" << N << "):" << endl;
    for (int i = 0; i < N * N; ++i)
        cin >> hB[i];
```

```
    // Allocate memory for matrices on the device
    int size = N * N * sizeof(float);
    float *dA, *dB, *dC;
    cudaMalloc(&dA, size);
    cudaMalloc(&dB, size);
    cudaMalloc(&dC, size);

    // Copy matrices from the host to the device
    cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);

    dim3 threadBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(N / BLOCK_SIZE, N / BLOCK_SIZE);

    // Execute the matrix multiplication kernel
    gpuMM<<<grid, threadBlock>>>(dA, dB, dC, N);

    // Copy the result matrix from the device to the host
    cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);

    // Display the result matrix
    cout << "\nResultant matrix:\n";
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            cout << hC[row * N + col] << " ";
        }
        cout << endl;
    }

    // Free device memory
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);

    // Free host memory
    delete[] hA;
    delete[] hB;
    delete[] hC;

    cout << "Finished." << endl;
    return 0;
}
```

```
!nvcc matrix_mult.cu -o matrix_mult
```

```
!./matrix_mult
```

OUTPUT :

```
Enter size of matrix (N): 3

Executing Matrix Multiplication
Matrix size: 3x3
Enter elements of matrix A (3x3):
1 2 3
4 5 6
7 8 9
Enter elements of matrix B (3x3):
9 8 7
6 5 4
3 2 1

Resultant matrix:
30 24 18
84 69 54
138 114 90
Finished.
```