Code

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

using namespace std;

// Function to perform BFS from a given vertex
void bfs(int startVertex, vector<bool> &visited, vector<vector<int>> &graph)
{
    // Create a queue for BFS
    queue<int> q;

    // Mark the start vertex as visited and enqueue it
    visited[startVertex] = true;
    q.push(startVertex);

    // Loop until the queue is empty
    while (!q.empty())
    {
        // Dequeue a vertex from the queue
        int v = q.front();
        q.pop();

// Enqueue all adjacent vertices that are not visited
#pragma omp parallel for
        for (int i = 0; i < graph[v].size(); i++)
        {
            int u = graph[v][i];
#pragma omp critical
            {
                if (!visited[u])
                {
                    visited[u] = true;
                    q.push(u);
                }
            }
        }
    }
}

// Parallel Breadth-First Search
void parallelBFS(vector<vector<int>> &graph, int numCores)
{
    int numVertices = graph.size();
    vector<bool> visited(numVertices, false); // Keep track of visited vertices

    double startTime = omp_get_wtime(); // Start timer

// Perform BFS from all unvisited vertices using specified number of cores
```

```
#pragma omp parallel for num_threads(numCores)
  for (int v = 0; v < numVertices; v++)
  {
    if (!visited[v])
    {
      bfs(v, visited, graph);
    }
  }

  double endTime = omp_get_wtime(); // End timer

  cout << "Number of cores used: " << numCores << endl;
  cout << "Time taken: " << endTime - startTime << " seconds" << endl;
  cout << "------------------------" << endl;
}

int main()
{
  // Generate a random graph with 10,000 vertices and 50,000 edges
  int numVertices = 10000;
  int numEdges = 50000;
  vector<vector<int>> graph(numVertices);
  srand(time(0));
  for (int i = 0; i < numEdges; i++)
  {
    int u = rand() % numVertices;
    int v = rand() % numVertices;
    graph[u].push_back(v);
    graph[v].push_back(u);
  }

  // Array containing number of cores
  int numCoresArr[] = {1, 2, 3, 4, 5, 6, 7, 8};

  // Loop over different number of cores and execute parallel BFS
  for (int i = 0; i < sizeof(numCoresArr) / sizeof(numCoresArr[0]); i++)
  {
    int numCores = numCoresArr[i];
    cout << "Running parallel BFS with " << numCores << " core(s)..." << endl;
    parallelBFS(graph, numCores);
  }

  return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ dfs.cpp -o dfs -fopenmp -pthread
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./dfs
Running parallel DFS with 1 core(s)...
Number of cores used: 1
Time taken: 10.921 seconds
------------------------
Running parallel DFS with 2 core(s)...
Number of cores used: 2
Time taken: 0.0769999 seconds
------------------------
Running parallel DFS with 3 core(s)...
Number of cores used: 3
Time taken: 0.0639999 seconds
------------------------
Running parallel DFS with 4 core(s)...
Number of cores used: 4
Time taken: 0.0469999 seconds
------------------------
Running parallel DFS with 5 core(s)...
Number of cores used: 5
Time taken: 0.043 seconds
------------------------
Running parallel DFS with 6 core(s)...
Number of cores used: 6
Time taken: 0.0400002 seconds
------------------------
Running parallel DFS with 7 core(s)...
Number of cores used: 7
Time taken: 0.0380001 seconds
------------------------
Running parallel DFS with 8 core(s)...
Number of cores used: 8
Time taken: 0.0370002 seconds
------------------------
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```

Code

```
#include <iostream>
#include <vector>
#include <stack>
#include <ctime>
#include <omp.h>

using namespace std;

// Function to perform DFS from a given vertex
void dfs(int startVertex, vector<bool> &visited, vector<vector<int>> &graph)
{
   // Create a stack for DFS
   stack<int> s;

   // Mark the start vertex as visited and push it onto the stack
   visited[startVertex] = true;
   s.push(startVertex);

   // Loop until the stack is empty
   while (!s.empty())
   {
      // Pop a vertex from the stack
      int v = s.top();
      s.pop();

// Push all adjacent vertices that are not visited onto the stack
#pragma omp parallel for
      for (int i = 0; i < graph[v].size(); i++)
      {
         int u = graph[v][i];
#pragma omp critical
         {
            if (!visited[u])
            {
               visited[u] = true;
               s.push(u);
            }
         }
      }
   }
}

// Parallel Depth-First Search
void parallelDFS(vector<vector<int>> &graph, int numCores)
{
   int numVertices = graph.size();
   vector<bool> visited(numVertices, false); // Keep track of visited vertices

   double startTime = omp_get_wtime(); // Start timer

// Perform DFS from all unvisited vertices using specified number of cores
```

```
#pragma omp parallel for num_threads(numCores)
    for (int v = 0; v < numVertices; v++)
    {
        if (!visited[v])
        {
            dfs(v, visited, graph);
        }
    }

    double endTime = omp_get_wtime(); // End timer

    cout << "Number of cores used: " << numCores << endl;
    cout << "Time taken: " << endTime - startTime << " seconds" << endl;
    cout << "------------------------" << endl;
}

int main()
{
    // Generate a random graph with 10,000 vertices and 50,000 edges
    int numVertices = 10000;
    int numEdges = 50000;
    vector<vector<int>> graph(numVertices);
    srand(time(0));
    for (int i = 0; i < numEdges; i++)
    {
        int u = rand() % numVertices;
        int v = rand() % numVertices;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // Array containing number of cores
    int numCoresArr[] = {1, 2, 3, 4, 5, 6, 7, 8};

    // Loop over different number of cores and execute parallel DFS
    for (int i = 0; i < sizeof(numCoresArr) / sizeof(numCoresArr[0]); i++)
    {
        int numCores = numCoresArr[i];
        cout << "Running parallel DFS with " << numCores << " core(s)..." << endl;
        parallelDFS(graph, numCores);
    }

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ bfs.cpp -o dfs -fopenmp -pthread
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./bfs
Running parallel BFS with 1 core(s)...
Number of cores used: 1
Time taken: 11.01 seconds
------------------------
Running parallel BFS with 2 core(s)...
Number of cores used: 2
Time taken: 0.076 seconds
------------------------
Running parallel BFS with 3 core(s)...
Number of cores used: 3
Time taken: 0.0679998 seconds
------------------------
Running parallel BFS with 4 core(s)...
Number of cores used: 4
Time taken: 0.0539999 seconds
------------------------
Running parallel BFS with 5 core(s)...
Number of cores used: 5
Time taken: 0.0469999 seconds
------------------------
Running parallel BFS with 6 core(s)...
Number of cores used: 6
Time taken: 0.043 seconds
------------------------
Running parallel BFS with 7 core(s)...
Number of cores used: 7
Time taken: 0.0419998 seconds
------------------------
Running parallel BFS with 8 core(s)...
Number of cores used: 8
Time taken: 0.0409999 seconds
------------------------
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```

Code

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

#include <array>
#include <chrono>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;
using std::chrono::milliseconds;
using namespace std;

void s_bubble(int *, int);
void p_bubble(int *, int);
void swap(int &, int &);

void s_bubble(int *a, int n)
{
   for (int i = 0; i < n; i++)
   {
      int first = i % 2;
      for (int j = first; j < n - 1; j += 2)
      {
         if (a[j] > a[j + 1])
         {
            swap(a[j], a[j + 1]);
         }
      }
   }
}

void p_bubble(int *a, int n)
{
   for (int i = 0; i < n; i++)
   {
      int first = i % 2;
#pragma omp parallel for shared(a, first) num_threads(16)
      for (int j = first; j < n - 1; j += 2)
      {
         if (a[j] > a[j + 1])
         {
            swap(a[j], a[j + 1]);
         }
      }
   }
}

void swap(int &a, int &b)
```

```
{
   int test;
   test = a;
   a = b;
   b = test;
}

int bench_traverse(std::function<void()> traverse_fn)
{
   auto start = high_resolution_clock::now();
   traverse_fn();
   auto stop = high_resolution_clock::now();

   // Subtract stop and start timepoints and cast it to required unit.
   // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
   // minutes, hours. Use duration_cast() function.
   auto duration = duration_cast<milliseconds>(stop - start);

   // To get the value of duration use the count() member function on the
   // duration object
   return duration.count();
}

int main(int argc, const char **argv)
{
   if (argc < 2)
   {
      cout << "Specify array length.\n";
      return 1;
   }
   int *a, n;
   n = stoi(argv[1]);
   a = new int[n];

   for (int i = 0; i < n; i++)
   {
      a[i] = rand() % n;
   }

   int *b = new int[n];
   copy(a, a + n, b);
   cout << "Generated random array of length " << n << "\n\n";

   int sequentialTime = bench_traverse([&]
                              { s_bubble(a, n); });

   omp_set_num_threads(16);
   int parallelTime = bench_traverse([&]
                              { s_bubble(a, n); });

   float speedUp = (float)sequentialTime / parallelTime;
   float efficiency = speedUp / 16;
```

```cpp
    cout
        << "Sequential Bubble sort: " << sequentialTime << "ms\n";

    cout << "Parallel (16) Bubble sort: " << parallelTime << "ms\n";

    cout << "Speed Up: " << speedUp << "\n";

    cout << "Efficiency: " << efficiency << "\n";

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ bubble.cpp -o bubble -fopenmp
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./bubble 100000
Generated random array of length 100000

Sequential Bubble sort: 36615ms
Parallel (16) Bubble sort: 15881ms
Speed Up: 2.30559
Efficiency: 0.144099
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```

Code

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>

using namespace std;

#define ARRAY_SIZE 5000

void merge(int arr[], int left[], int left_size, int right[], int right_size)
{
   int i = 0, j = 0, k = 0;
   while (i < left_size && j < right_size)
   {
      if (left[i] <= right[j])
      {
         arr[k] = left[i];
         i++;
      }
      else
      {
         arr[k] = right[j];
         j++;
      }
      k++;
   }
   while (i < left_size)
   {
      arr[k] = left[i];
      i++;
      k++;
   }
   while (j < right_size)
   {
      arr[k] = right[j];
      j++;
      k++;
   }
}

void merge_sort(int arr[], int size)
{
   if (size < 2)
   {
      return;
   }
   int mid = size / 2;
   int left[mid], right[size - mid];
   for (int i = 0; i < mid; i++)
   {
```

```
        left[i] = arr[i];
    }
    for (int i = mid; i < size; i++)
    {
        right[i - mid] = arr[i];
    }
#pragma omp parallel sections
    {
#pragma omp section
        {
            merge_sort(left, mid);
        }
#pragma omp section
        {
            merge_sort(right, size - mid);
        }
    }
    merge(arr, left, mid, right, size - mid);
}

int main()
{
    int arr[ARRAY_SIZE];
    int num_threads_array[] = {16};
    int num_threads_array_size = sizeof(num_threads_array) / sizeof(int);

    // Initialize the array with random values
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        arr[i] = rand() % ARRAY_SIZE;
    }

    // Sort the array using normal merge sort
    clock_t start_time = clock();
    merge_sort(arr, ARRAY_SIZE);
    clock_t end_time = clock();
    double normal_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    // Sort the array in parallel using OpenMP
    for (int i = 0; i < num_threads_array_size; i++)
    {
        int num_threads = num_threads_array[i];
        printf("Number of threads: %d\n", num_threads);
        start_time = clock();
        omp_set_num_threads(num_threads);
#pragma omp parallel
        {
#pragma omp single
            {
                merge_sort(arr, ARRAY_SIZE);
            }
        }
```

```
        end_time = clock();
        double parallel_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

        // Print the time taken by both merge sorts
        printf("Time taken (normal merge sort): %f seconds\n", normal_time);
        printf("Time taken (parallel merge sort): %f seconds\n", parallel_time);

        float speedUp = normal_time / parallel_time;
        float efficiency = speedUp / num_threads;

        cout << "Speed Up: " << speedUp << "\n";
        cout << "Efficiency: " << efficiency << "\n";
        printf("\n");
    }

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ merge.cpp -o merge -fopenmp
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./merge
Number of threads: 16
Time taken (normal merge sort): 0.083000 seconds
Time taken (parallel merge sort): 0.070000 seconds
Speed Up: 1.18571
Efficiency: 0.0741071

PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```

Code

```
#include <limits.h>
#include <omp.h>
#include <stdlib.h>

#include <array>
#include <chrono>
#include <functional>
#include <iostream>
#include <string>
#include <vector>

using std::chrono::duration_cast;
using std::chrono::high_resolution_clock;
using std::chrono::milliseconds;
using namespace std;

void s_avg(int arr[], int n)
{
  long sum = 0L;
  int i;
  for (i = 0; i < n; i++)
  {
    sum = sum + arr[i];
  }
  // cout << "\nAverage = " << sum / long(n) << "\n";
}

void p_avg(int arr[], int n)
{
  long sum = 0L;
  int i;
#pragma omp parallel for reduction(+ : sum) num_threads(16)
  for (i = 0; i < n; i++)
  {
    sum = sum + arr[i];
  }
  // cout << "\nAverage = " << sum / long(n) << "\n";
}

void s_sum(int arr[], int n)
{
  long sum = 0L;
  int i;
  for (i = 0; i < n; i++)
  {
    sum = sum + arr[i];
  }
  // cout << "\nSum = " << sum << "\n";
}

void p_sum(int arr[], int n)
```

```
{
    long sum = 0L;
    int i;
#pragma omp parallel for reduction(+ : sum) num_threads(16)
    for (i = 0; i < n; i++)
    {
        sum = sum + arr[i];
    }
    // cout << "\nSum = " << sum << "\n";
}

void s_max(int arr[], int n)
{
    int max_val = INT_MIN;
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max_val)
        {
            max_val = arr[i];
        }
    }
    // cout << "\nMax value = " << max_val << "\n";
}

void p_max(int arr[], int n)
{
    int max_val = INT_MIN;
    int i;
#pragma omp parallel for reduction(max : max_val) num_threads(16)
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max_val)
        {
            max_val = arr[i];
        }
    }
    // cout << "\nMax value = " << max_val << "\n";
}

void s_min(int arr[], int n)
{
    int min_val = INT_MAX;
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] < min_val)
        {
            min_val = arr[i];
        }
    }
    // cout << "\nMin value = " << min_val << "\n";
```

```
}

void p_min(int arr[], int n)
{
   int min_val = INT_MAX;
   int i;
#pragma omp parallel for reduction(min : min_val) num_threads(16)
   for (i = 0; i < n; i++)
   {
      if (arr[i] < min_val)
      {
         min_val = arr[i];
      }
   }
   // cout << "\nMin value = " << min_val << "\n";
}

int bench_traverse(std::function<void()> traverse_fn)
{
   auto start = high_resolution_clock::now();
   traverse_fn();
   auto stop = high_resolution_clock::now();

   // Subtract stop and start timepoints and cast it to required unit.
   // Predefined units are nanoseconds, microseconds, milliseconds, seconds,
   // minutes, hours. Use duration_cast() function.
   auto duration = duration_cast<milliseconds>(stop - start);

   // To get the value of duration use the count() member function on the
   // duration object
   duration.count();
}

int main(int argc, const char **argv)
{
   if (argc < 2)
   {
      cout << "Specify array length.\n";
      return 1;
   }
   int *a, n, i;

   n = stoi(argv[1]);
   a = new int[n];

   for (int i = 0; i < n; i++)
   {
      a[i] = rand() % n;
   }

   cout << "Generated random array of length " << n << "\n\n";
   omp_set_num_threads(16);
```

```
    int sequentialMin = bench_traverse([&]
                                   { s_min(a, n); });

    int parallelMin = bench_traverse([&]
                                 { p_min(a, n); });

    int sequentialMax = bench_traverse([&]
                                   { s_max(a, n); });

    int parallelMax = bench_traverse([&]
                                 { p_max(a, n); });

    int sequentialSum = bench_traverse([&]
                                   { s_sum(a, n); });

    int parallelSum = bench_traverse([&]
                                 { p_sum(a, n); });

    int sequentialAverage = bench_traverse([&]
                                       { s_avg(a, n); });

    int parallelAverage = bench_traverse([&]
                                     { p_avg(a, n); });

    cout << "Sequential Min: " << sequentialMin << "ms\n";
    cout << "Parallel (16) Min: " << parallelMin << "ms\n";
    cout << "Speed Up for Min: " << (float)sequentialMin / parallelMin << "\n";
    cout << "Efficiency for Min: " << ((float)sequentialMin / parallelMin) / 16 << "\n";

    cout << "\nSequential Max: " << sequentialMax << "ms\n";
    cout << "Parallel (16) Max: " << parallelMax << "ms\n";
    cout << "Speed Up for Max: " << (float)sequentialMax / parallelMax << "\n";
    cout << "Efficiency for Max: " << ((float)sequentialMax / parallelMax) / 16 << "\n";

    cout << "\nSequential Sum: " << sequentialSum << "ms\n";
    cout << "Parallel (16) Sum: " << parallelSum << "ms\n";
    cout << "Speed Up for Sum: " << (float)sequentialSum / parallelSum << "\n";
    cout << "Efficiency for Sum: " << ((float)sequentialSum / parallelSum) / 16 << "\n";

    cout << "\nSequential Average: " << sequentialAverage << "ms\n";
    cout << "Parallel (16) Average: " << parallelAverage << "ms\n";
    cout << "Speed Up for Average: " << (float)sequentialAverage / parallelAverage << "\n";
    cout << "Efficiency for Average: " << ((float)sequentialAverage / parallelAverage) / 16 << "\n";

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> g++ reduction.cpp -o reduction -fopenmp
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1> ./reduction 10000000
Generated random array of length 10000000

Sequential Min: 36ms
Parallel (16) Min: 7ms
Speed Up for Min: 5.14286
Efficiency for Min: 0.321429

Sequential Max: 36ms
Parallel (16) Max: 4ms
Speed Up for Max: 9
Efficiency for Max: 0.5625

Sequential Sum: 36ms
Parallel (16) Sum: 4ms
Speed Up for Sum: 9
Efficiency for Sum: 0.5625

Sequential Average: 37ms
Parallel (16) Average: 5ms
Speed Up for Average: 7.4
Efficiency for Average: 0.4625
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\1>
```

Addition of two Large Vectors
Code

```
#include <stdio.h>

__global__ void vectorAdd(float *a, float *b, float *c, int n)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < n)
   {
      c[i] = a[i] + b[i];
   }
}

int main()
{
   int n = 1000000;
   size_t bytes = n * sizeof(float);

   // Allocate memory on the host
   float *h_a = (float *)malloc(bytes);
   float *h_b = (float *)malloc(bytes);
   float *h_c = (float *)malloc(bytes);

   // Initialize the vectors
   for (int i = 0; i < n; i++)
   {
      h_a[i] = i;
      h_b[i] = i + 1;
   }

   // Allocate memory on the device
   float *d_a, *d_b, *d_c;
   cudaMalloc(&d_a, bytes);
   cudaMalloc(&d_b, bytes);
   cudaMalloc(&d_c, bytes);

   // Copy data from host to device
   cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
   cudaMemcpy(d_b, h_b, bytes, cudaMemcpyHostToDevice);

   // Launch kernel on the device
   int threadsPerBlock = 256;
   int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
   vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);

   // Copy result from device to host
   cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost);

   // Print first 10 elements of both vectors
   printf("First 10 elements of vector a:\n");
   for (int i = 0; i < 10; i++)
   {
```

```
        printf("%.2f ", h_a[i]);
    }
    printf("\n");
    printf("Size of vector a: %d\n", n);
    printf("\n");

    printf("First 10 elements of vector b:\n");
    for (int i = 0; i < 10; i++)
    {
        printf("%.2f ", h_b[i]);
    }
    printf("\n");
    printf("Size of vector b: %d\n", n);
    printf("\n");

    // Print first 10 elements of resultant vector
    printf("First 10 elements of resultant vector:\n");
    for (int i = 0; i < 10; i++)
    {
        printf("%.2f ", h_c[i]);
    }
    printf("\n");

    // Print size of resultant vector
    printf("Size of resultant vector: %d\n", n);

    // Free memory
    free(h_a);
    free(h_b);
    free(h_c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\4> nvcc addition.cu -o addition
addition.cu
tmpxft_000035f4_00000000-10_addition.cudafe1.cpp
   Creating library addition.lib and object addition.exp
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\4> ./addition
First 10 elements of vector a:
0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00
Size of vector a: 1000000

First 10 elements of vector b:
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00 10.00
Size of vector b: 1000000

First 10 elements of resultant vector:
1.00 3.00 5.00 7.00 9.00 11.00 13.00 15.00 17.00 19.00
Size of resultant vector: 1000000
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\4>
```

Matrix Multiplication using CUDA C
Code

```
#include <stdio.h>

#define TILE_WIDTH 32

__global__ void matrixMul(float *a, float *b, float *c, int m, int n, int p)
{
  __shared__ float As[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;
  int by = blockIdx.y;
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int row = by * TILE_WIDTH + ty;
  int col = bx * TILE_WIDTH + tx;

  float Cvalue = 0.0;

  for (int k = 0; k < n / TILE_WIDTH; k++)
  {
    As[ty][tx] = a[row * n + k * TILE_WIDTH + tx];
    Bs[ty][tx] = b[(k * TILE_WIDTH + ty) * p + col];

    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; i++)
    {
      Cvalue += As[ty][i] * Bs[i][tx];
    }

    __syncthreads();
  }

  c[row * p + col] = Cvalue;
}

int main()
{
  int m = 1024;
  int n = 1024;
  int p = 1024;
  size_t bytesA = m * n * sizeof(float);
  size_t bytesB = n * p * sizeof(float);
  size_t bytesC = m * p * sizeof(float);

  // Allocate memory on the host
  float *h_a = (float *)malloc(bytesA);
  float *h_b = (float *)malloc(bytesB);
  float *h_c = (float *)malloc(bytesC);
```

```
// Initialize matrices
for (int i = 0; i < m * n; i++)
{
   h_a[i] = 1.0;
}
for (int i = 0; i < n * p; i++)
{
   h_b[i] = 2.0;
}

// Allocate memory on the device
float *d_a, *d_b, *d_c;
cudaMalloc(&d_a, bytesA);
cudaMalloc(&d_b, bytesB);
cudaMalloc(&d_c, bytesC);

// Copy data from host to device
cudaMemcpy(d_a, h_a, bytesA, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, bytesB, cudaMemcpyHostToDevice);

// Launch kernel on the device
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid((p + dimBlock.x - 1) / dimBlock.x, (m + dimBlock.y - 1) / dimBlock.y);
matrixMul<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, m, n, p);

// Copy result from device to host
cudaMemcpy(h_c, d_c, bytesC, cudaMemcpyDeviceToHost);

// Print 3x3 parts of both matrices
printf("Matrix A (3x3 part):\n");
for (int i = 0; i < 3; i++)
{
   for (int j = 0; j < 3; j++)
   {
      printf("%.2f ", h_a[i * n + j]);
   }
   printf("\n");
}
printf("Size of Matrix A: %dx%d\n", m, n);
printf("\n");

printf("Matrix B (3x3 part):\n");
for (int i = 0; i < 3; i++)
{
   for (int j = 0; j < 3; j++)
   {
      printf("%.2f ", h_b[i * p + j]);
   }
   printf("\n");
}
printf("Size of Matrix B: %dx%d\n", n, p);
```

```
    printf("\n");

    // Print 3x3 part of resultant matrix
    printf("Resultant Matrix (3x3 part):\n");
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            printf("%.2f ", h_c[i * p + j]);
        }
        printf("\n");
    }

    // Print size of resultant matrix
    printf("Size of Resultant Matrix: %dx%d\n", m, p);

    // Free memory on the host and device
    free(h_a);
    free(h_b);
    free(h_c);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Output

```
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\4> nvcc multiplication.cu -o multiplication
multiplication.cu
tmpxft_00006360_00000000-10_multiplication.cudafe1.cpp
   Creating library multiplication.lib and object multiplication.exp
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\4> ./multiplication
Matrix A (3x3 part):
1.00 1.00 1.00
1.00 1.00 1.00
1.00 1.00 1.00
Size of Matrix A: 1024x1024

Matrix B (3x3 part):
2.00 2.00 2.00
2.00 2.00 2.00
2.00 2.00 2.00
Size of Matrix B: 1024x1024

Resultant Matrix (3x3 part):
2048.00 2048.00 2048.00
2048.00 2048.00 2048.00
2048.00 2048.00 2048.00
Size of Resultant Matrix: 1024x1024
PS D:\Gaurav\College\4 BE\Sem 8\High Performance Computing\Lab Assignments\4>
```