Name: Siddhi Hagawane

PRN No: 2020BTECS00049

**High Performance Computing Lab**

**Practical No. 8**

**Title of practical:** Implementation of Vector-Vector addition & N-Body Simulator using CUDA C

**Problem Statement 1:**

Implement Vector-Vector addition using CUDA C. State and justify the speedup using different size of threads and blocks.

Code:

#include <stdio.h>

#include <stdlib.h>

#include <cuda\_runtime.h>

#define N 1000000  // Size of vectors

// CUDA kernel for vector addition

\_\_global\_\_ void vectorAddition(int \*a, int \*b, int \*result, int n) {

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

    while (tid < n) {

        result[tid] = a[tid] + b[tid];

        tid += blockDim.x \* gridDim.x;

    }

}

int main() {

    int \*h\_a, \*h\_b, \*h\_result;  // Host vectors

    int \*d\_a, \*d\_b, \*d\_result;  // Device vectors

    // Allocate memory for host vectors

    h\_a = (int\*)malloc(N \* sizeof(int));

    h\_b = (int\*)malloc(N \* sizeof(int));

    h\_result = (int\*)malloc(N \* sizeof(int));

    // Initialize host vectors

    for (int i = 0; i < N; i++) {

        h\_a[i] = i;

        h\_b[i] = 2 \* i;

    }

    // Allocate memory for device vectors

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

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

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

    // Copy host vectors to device

    cudaMemcpy(d\_a, h\_a, N \* sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_b, h\_b, N \* sizeof(int), cudaMemcpyHostToDevice);

    // Define thread and block configurations

    int threadsPerBlock = 128, blocksPerGrid = 8;

    dim3 grid, block;

    // Measure execution time for different configurations

        // Configure kernel launch parameters

        block.x = threadsPerBlock;

        grid.x = blocksPerGrid;

        // Record start time

        cudaEvent\_t start, stop;

        cudaEventCreate(&start);

        cudaEventCreate(&stop);

        cudaEventRecord(start);

        // Launch vector addition kernel

        vectorAddition<<<grid, block>>>(d\_a, d\_b, d\_result, N);

        // Record end time

        cudaEventRecord(stop);

        cudaEventSynchronize(stop);

        float milliseconds = 0;

        cudaEventElapsedTime(&milliseconds, start, stop);

        // Copy result back to host

        cudaMemcpy(h\_result, d\_result, N \* sizeof(int), cudaMemcpyDeviceToHost);

        printf("Threads per block: %d, Blocks per grid: %d, Execution time: %f ms\n",

               threadsPerBlock, blocksPerGrid, milliseconds);

    // Free device and host memory

    cudaFree(d\_a);

    cudaFree(d\_b);

    cudaFree(d\_result);

    free(h\_a);

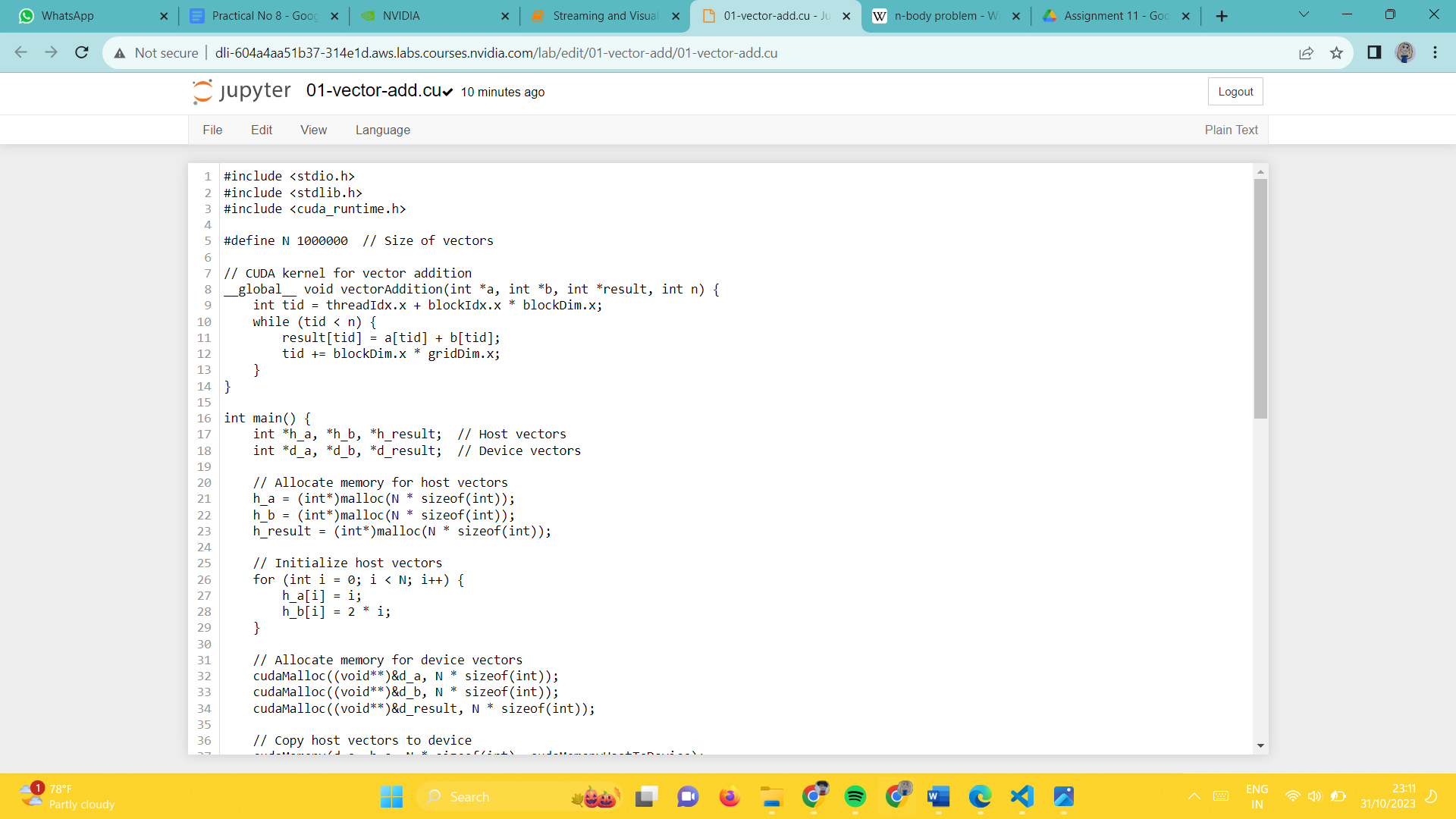
    free(h\_b);

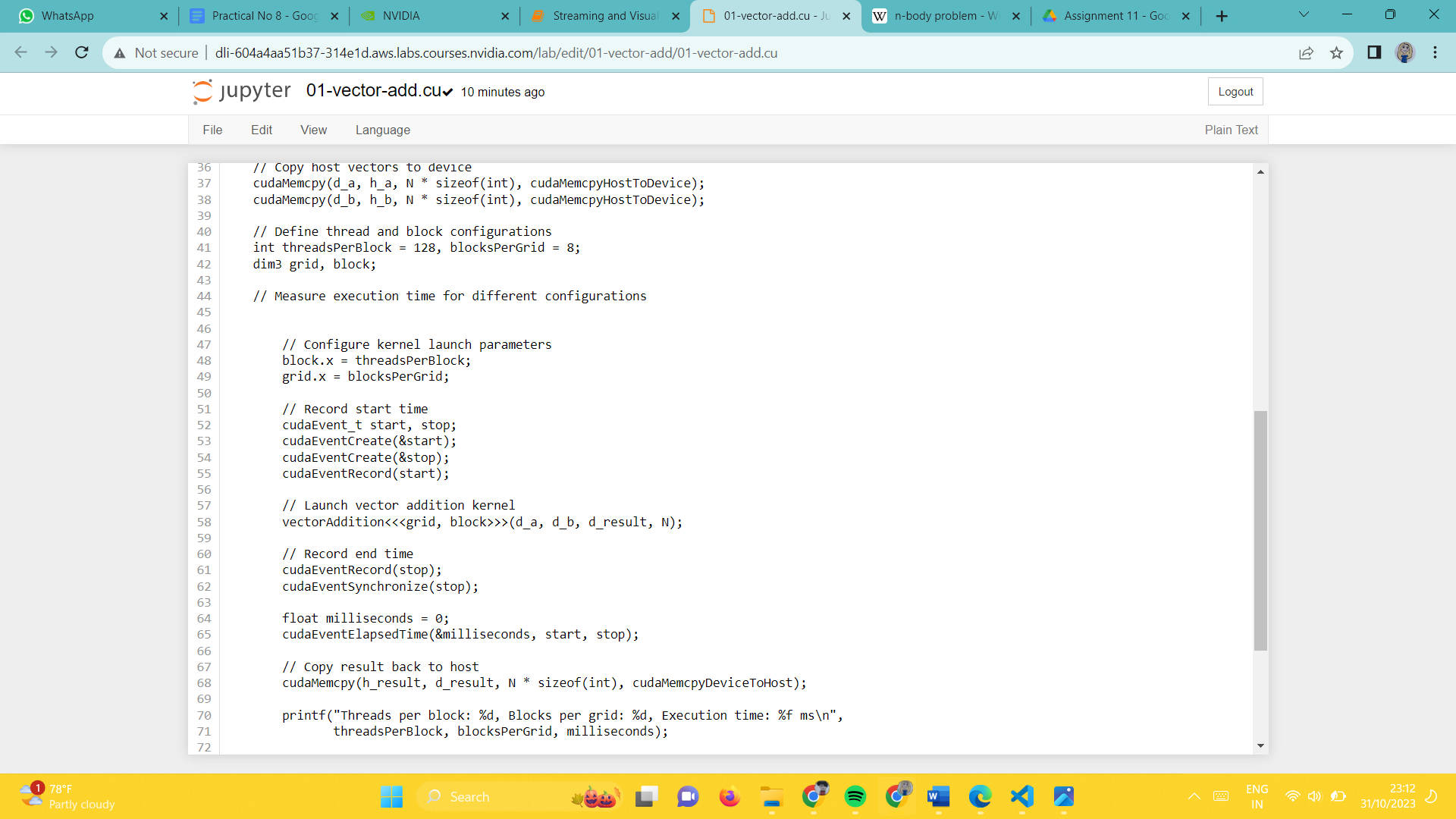
    free(h\_result);

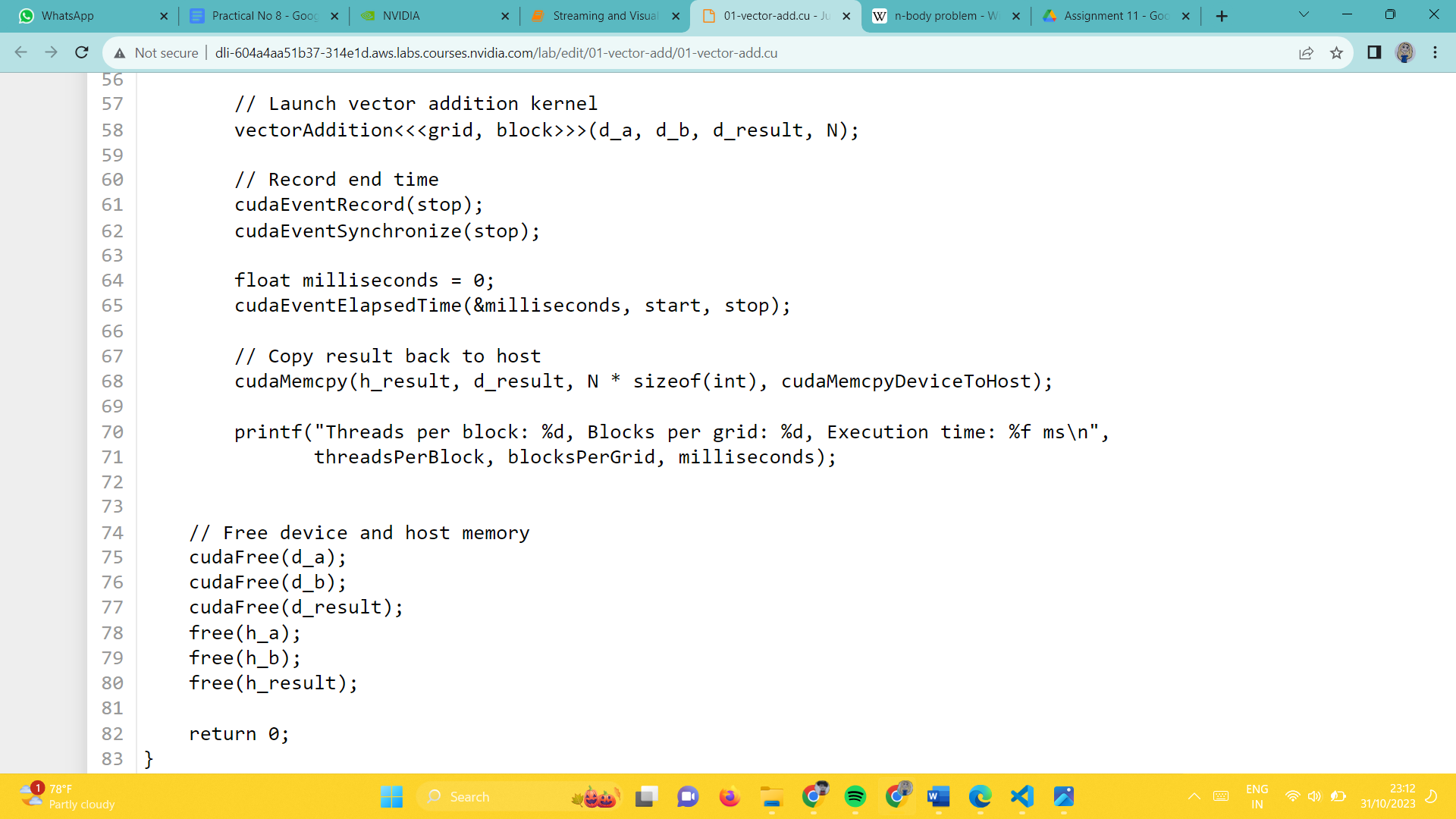
    return 0;

}

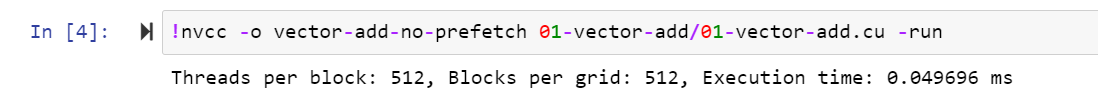
**Screenshots:**

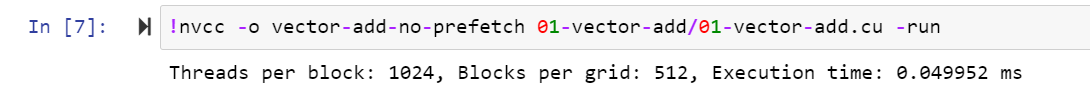


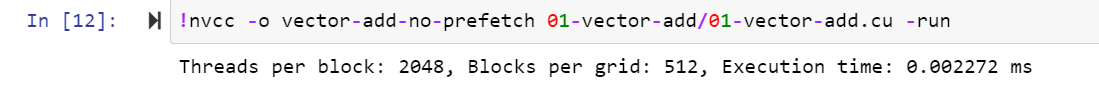




**Output:**

****

****

****

**Speedup analysis: Tabular and Graphical**

**By keeping Tread per block constant and as number of blocks per grid decrease execution time increase.**

**By increasing thread per blocks and decreasing blocks per grid execution time decreases.**

|  |  |  |
| --- | --- | --- |
| **Thread per blocks** | **Blocks per grid** | **Execution Time** |
| **512** | **512** | **0.049696** |
| **512** | **1024** | **0.052064** |
| **512** | **2048** | **0.049664** |
| **1024** | **512** | **0.049952** |
| **1024** | **1024** | **0.052640** |
| **1024** | **2048** | **0.065504** |
| **2048** | **512** | **0.002272** |
| **2048** | **1024** | **0.002272** |
| **2048** | **2048** | **0.002272** |

**Problem Statement 2:**

Implement N-Body Simulator using CUDA C. State and justify the speedup using different size of threads and blocks.

**Code**:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <cuda\_runtime.h>

#define N 1024  // Number of particles

#define G 6.674e-11  // Gravitational constant

// Particle structure

typedef struct {

    float x, y, z;  // Position

    float vx, vy, vz;  // Velocity

} Particle;

// CUDA kernel for updating particle positions and velocities

\_\_global\_\_ void updateParticles(Particle \*particles, int numParticles, float dt) {

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

    while (tid < numParticles) {

        // Compute gravitational forces and update velocities

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

            if (i != tid) {

                float dx = particles[i].x - particles[tid].x;

                float dy = particles[i].y - particles[tid].y;

                float dz = particles[i].z - particles[tid].z;

                float distSquared = dx \* dx + dy \* dy + dz \* dz;

                float invDist = 1.0f / sqrtf(distSquared);

                float invDistCube = invDist \* invDist \* invDist;

                float force = G \* invDistCube;

                particles[tid].vx += force \* dx \* dt;

                particles[tid].vy += force \* dy \* dt;

                particles[tid].vz += force \* dz \* dt;

            }

        }

        // Update positions

        particles[tid].x += particles[tid].vx \* dt;

        particles[tid].y += particles[tid].vy \* dt;

        particles[tid].z += particles[tid].vz \* dt;

        tid += blockDim.x \* gridDim.x;

    }

}

int main() {

    Particle \*h\_particles, \*d\_particles;  // Host and device particles

    // Allocate memory for host particles

    h\_particles = (Particle\*)malloc(N \* sizeof(Particle));

    // Initialize host particles

    for (int i = 0; i < N; i++) {

        h\_particles[i].x = rand() / (float)RAND\_MAX;

        h\_particles[i].y = rand() / (float)RAND\_MAX;

        h\_particles[i].z = rand() / (float)RAND\_MAX;

        h\_particles[i].vx = rand() / (float)RAND\_MAX;

        h\_particles[i].vy = rand() / (float)RAND\_MAX;

        h\_particles[i].vz = rand() / (float)RAND\_MAX;

    }

    // Allocate memory for device particles

    cudaMalloc((void\*\*)&d\_particles, N \* sizeof(Particle));

    // Copy host particles to device

    cudaMemcpy(d\_particles, h\_particles, N \* sizeof(Particle), cudaMemcpyHostToDevice);

    // Define thread and block configurations

    int threadsPerBlock = 64;  // Set an appropriate value

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

    // Measure execution time for different configurations

    float sequentialTime = 0; // Initialize sequential time to zero

    // Record start time for sequential version

    cudaEvent\_t seq\_start, seq\_stop;

    cudaEventCreate(&seq\_start);

    cudaEventCreate(&seq\_stop);

    cudaEventRecord(seq\_start);

    // Perform sequential simulation (for timing comparison only, not for correctness)

    for (int iter = 0; iter < 1000; iter++) {

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

            // ... (sequential computation logic)

        }

    }

    // Record end time for sequential version

    cudaEventRecord(seq\_stop);

    cudaEventSynchronize(seq\_stop);

    float seq\_milliseconds = 0;

    cudaEventElapsedTime(&seq\_milliseconds, seq\_start, seq\_stop);

    sequentialTime = seq\_milliseconds; // Save the sequential time

    // Allocate events for timing

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    // Record start time for parallel version

    cudaEventRecord(start);

    // Launch particle update kernel

    updateParticles<<<blocksPerGrid, threadsPerBlock>>>(d\_particles, N, 0.001f);

    // Record end time for parallel version

    cudaEventRecord(stop);

    cudaEventSynchronize(stop);

    // Calculate and print execution time

    float milliseconds = 0;

    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("Threads per block: %d, Blocks per grid: %d, Execution time: %f ms\n",

           threadsPerBlock, blocksPerGrid, milliseconds);

    // Calculate and print speedup

    float speedup = sequentialTime / milliseconds;

    printf("Speedup: %f\n", speedup);

    // Free device and host memory

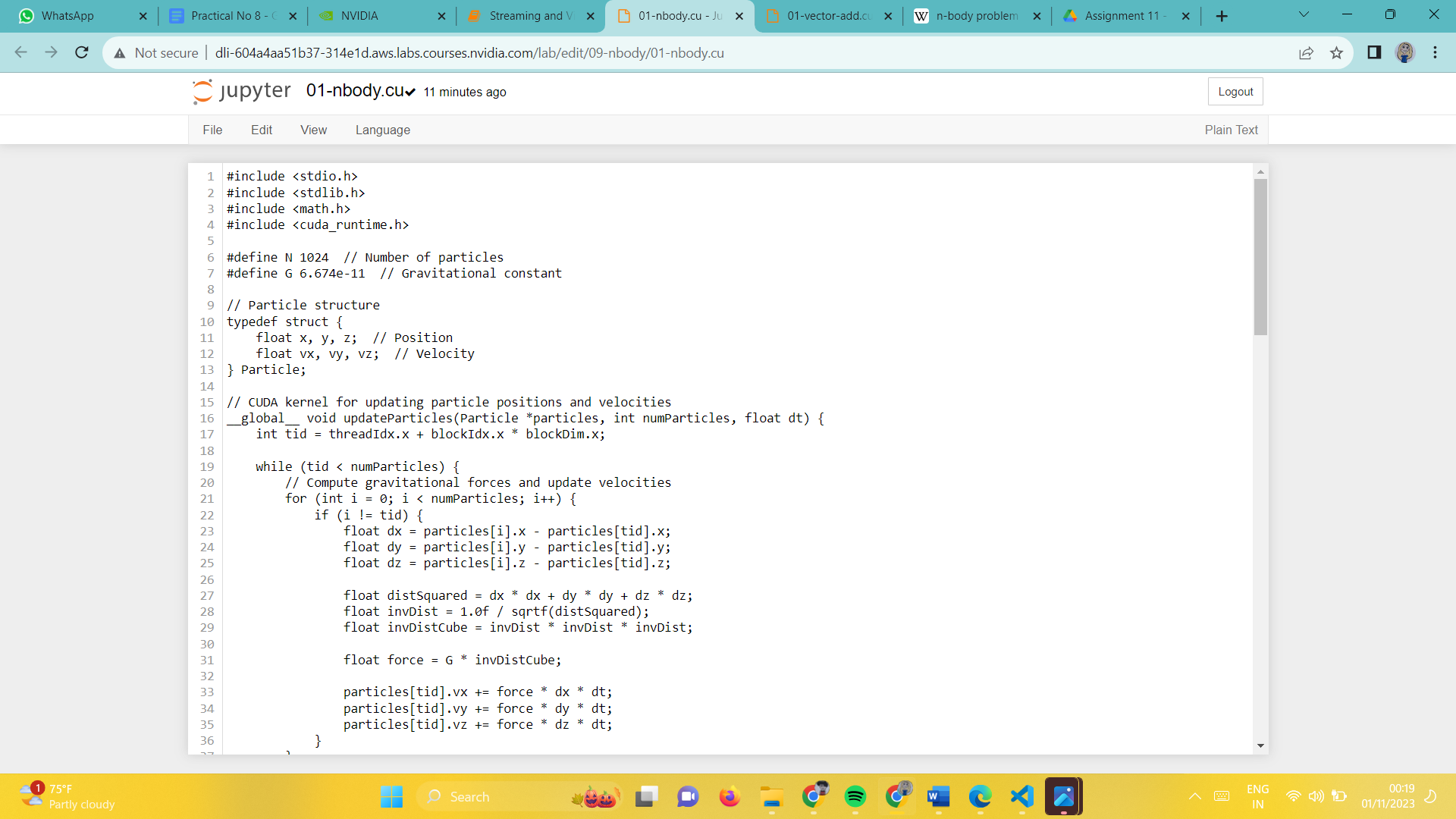
    cudaFree(d\_particles);

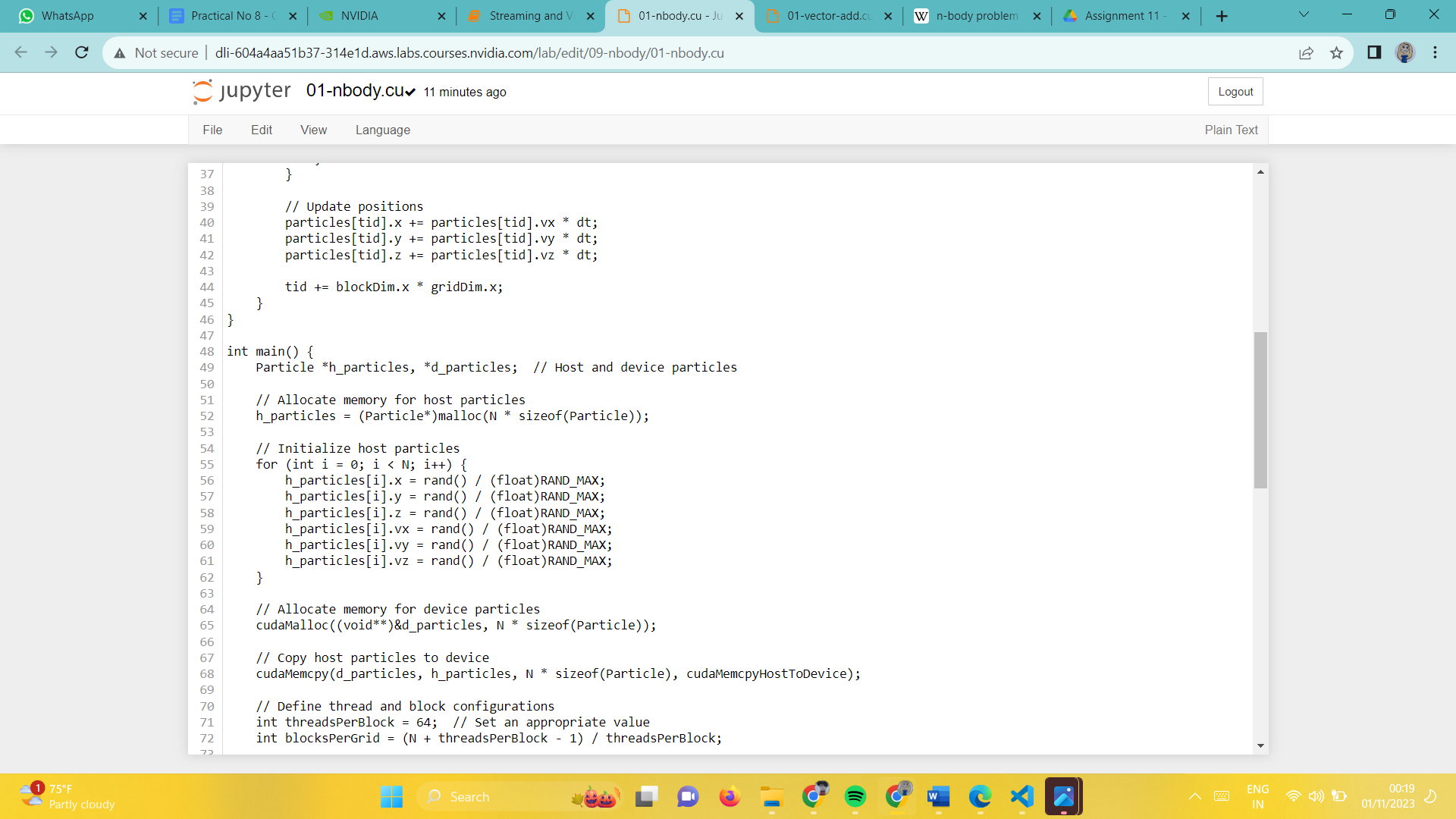
    free(h\_particles);

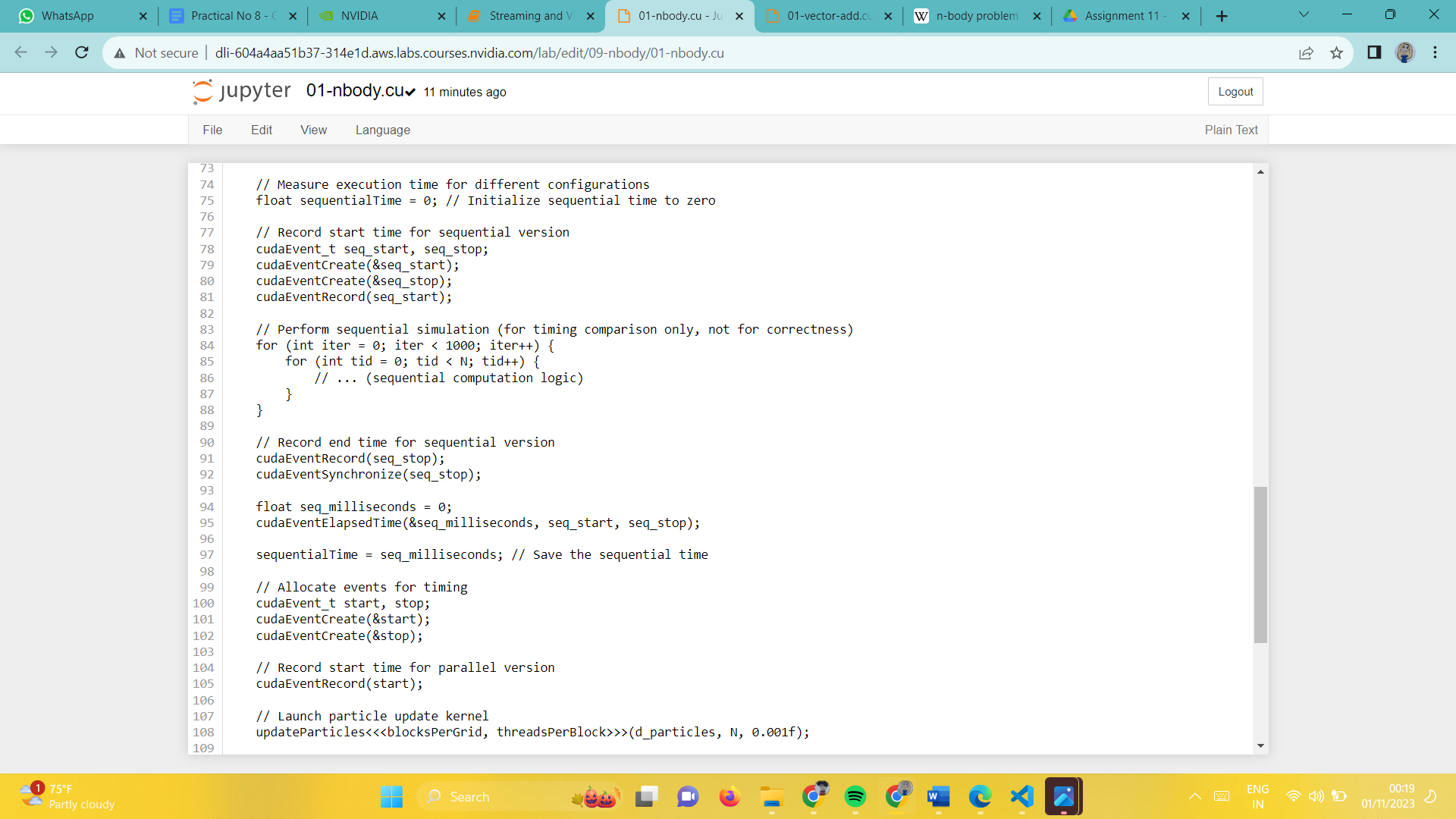
    return 0;

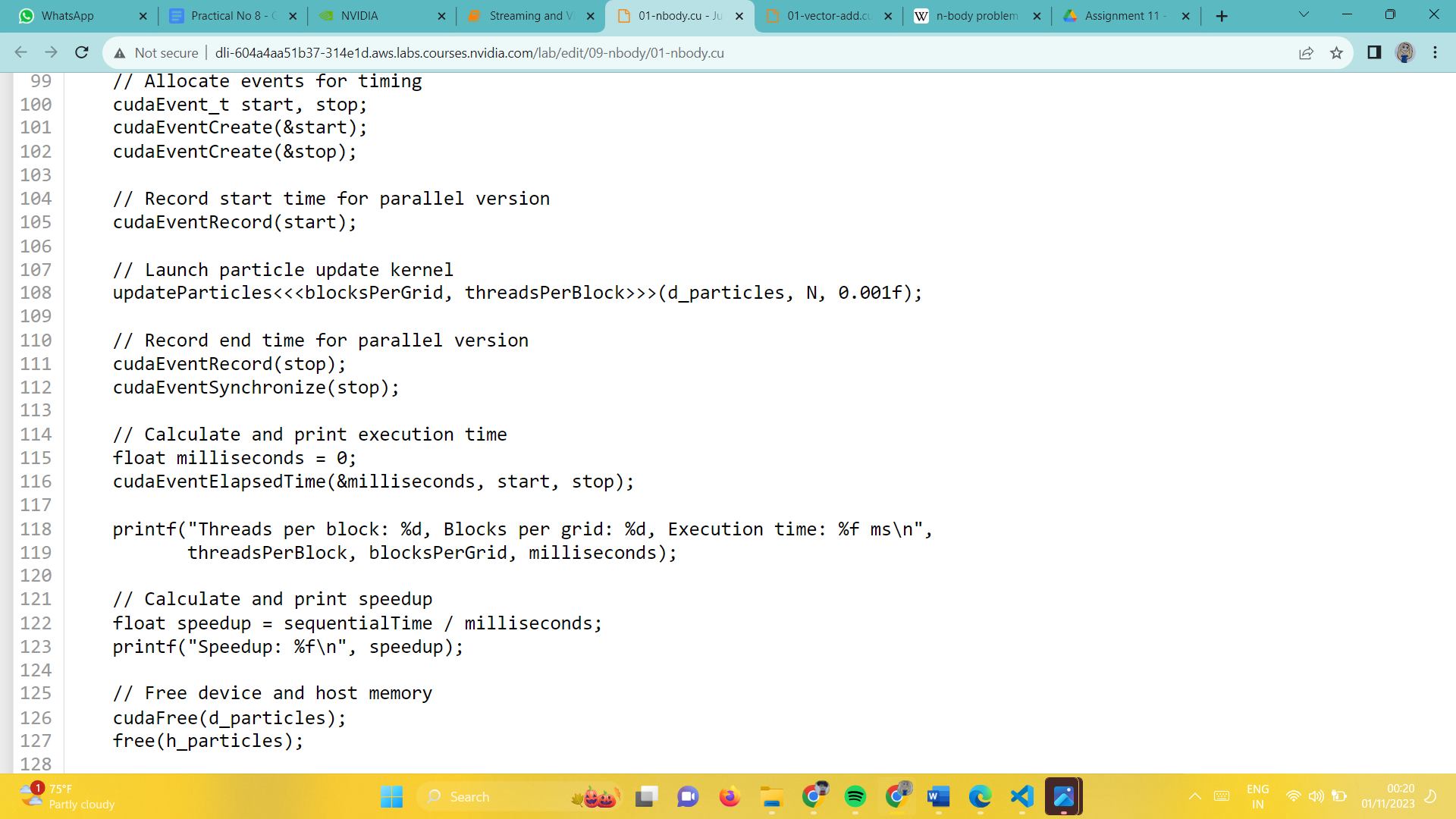
}

**Screenshots:**

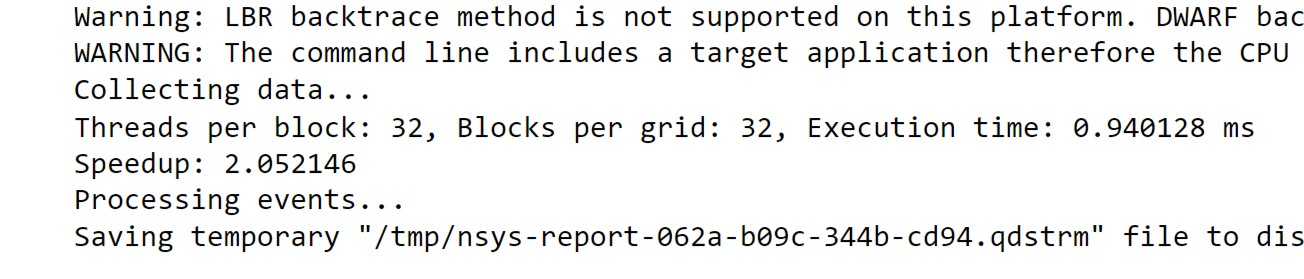


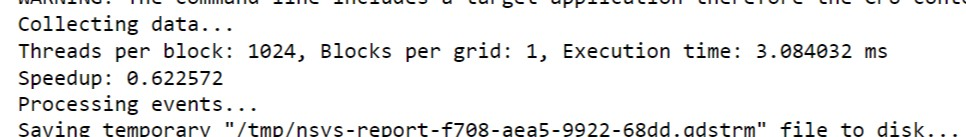


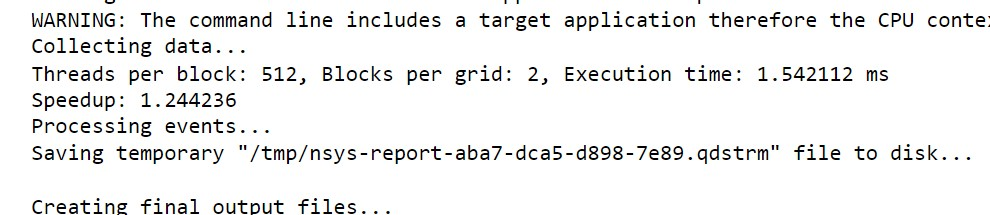


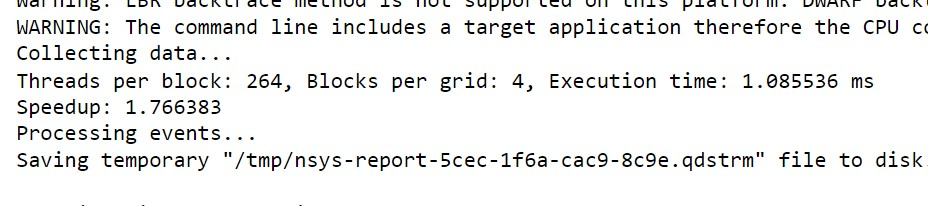


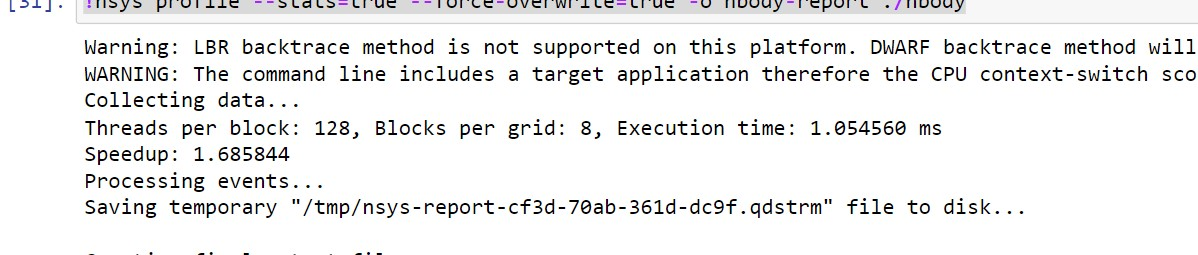
**Output:**

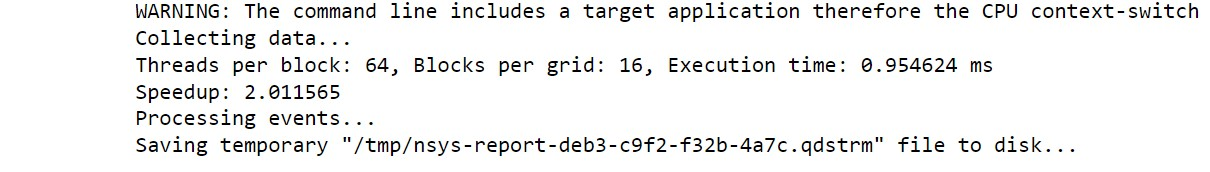












**Speedup analysis: Tabular and Graphical**

|  |  |  |  |
| --- | --- | --- | --- |
| **Threads per block** | **Blocks per grid** | **Execution time** | **Speedup** |
| **32** | **32** | **0.940128** | **2.052146** |
| **64** | **16** | **0.954624** | **2.011565** |
| **128** | **8** | **1.054560** | **1.685844** |
| **264** | **4** | **1.085536** | **1.766383** |
| **512** | **2** | **1.542112** | **1.244236** |
| **1024** | **1** | **3.084032** | **0.622572** |

**GitHub Link:**

<https://github.com/Siddhish16/HPC-Assignments>