**Class:** Final Year (Computer Science and Engineering)

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 8**

**Name:** Sumit Narake

**PRN:**2020BTECS00023

**Batch :** B2

**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.

%%cu

#include <stdio.h>

const int N = 100; // Size of the vectors

void initWith(int \*vector, int N) {

    for (int i = 0; i < N; ++i) {

        vector[i] = i; // Initialize vector elements with their index

    }

}

\_\_global\_\_ void addVectors(int \*result, int \*a, int \*b, int N) {

    int i = threadIdx.x;

    if (i < N) {

        result[i] = a[i] + b[i];

    }

}

void checkElementsAre(int \*vector, int N) {

    for (int i = 0; i < N; i++) {

        if (vector[i] != 2 \* i) {

            printf("FAIL: vector[%d] - %d does not equal %d\n", i, vector[i], 2 \* i);

            return;

        }

    }

    printf("Success! All values calculated correctly.\n");

}

int main() {

    int size = N \* sizeof(int);

    int \*a;

    int \*b;

    int \*c;

    // Allocate memory on the host

    a = (int \*)malloc(size);

    b = (int \*)malloc(size);

    c = (int \*)malloc(size);

    // Initialize vectors on the host

    initWith(a, N);

    initWith(b, N);

    // Allocate memory on the device (GPU)

    int \*d\_a, \*d\_b, \*d\_c;

    cudaMalloc(&d\_a, size);

    cudaMalloc(&d\_b, size);

    cudaMalloc(&d\_c, size);

    // Copy vectors from host to device

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

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

    // Perform vector addition on the GPU

    addVectors<<<1, N>>>(d\_c, d\_a, d\_b, N);

    cudaDeviceSynchronize();

    // Copy result back from device to host

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

    // Print the resulting vector

    printf("Resulting Vector c: ");

    for (int i = 0; i < N; i++) {

        printf("%d ", c[i]);

    }

    printf("\n");

    // Check if the calculation is correct

    checkElementsAre(c, N);

    // Free allocated memory

    free(a);

    free(b);

    free(c);

    cudaFree(d\_a);

    cudaFree(d\_b);

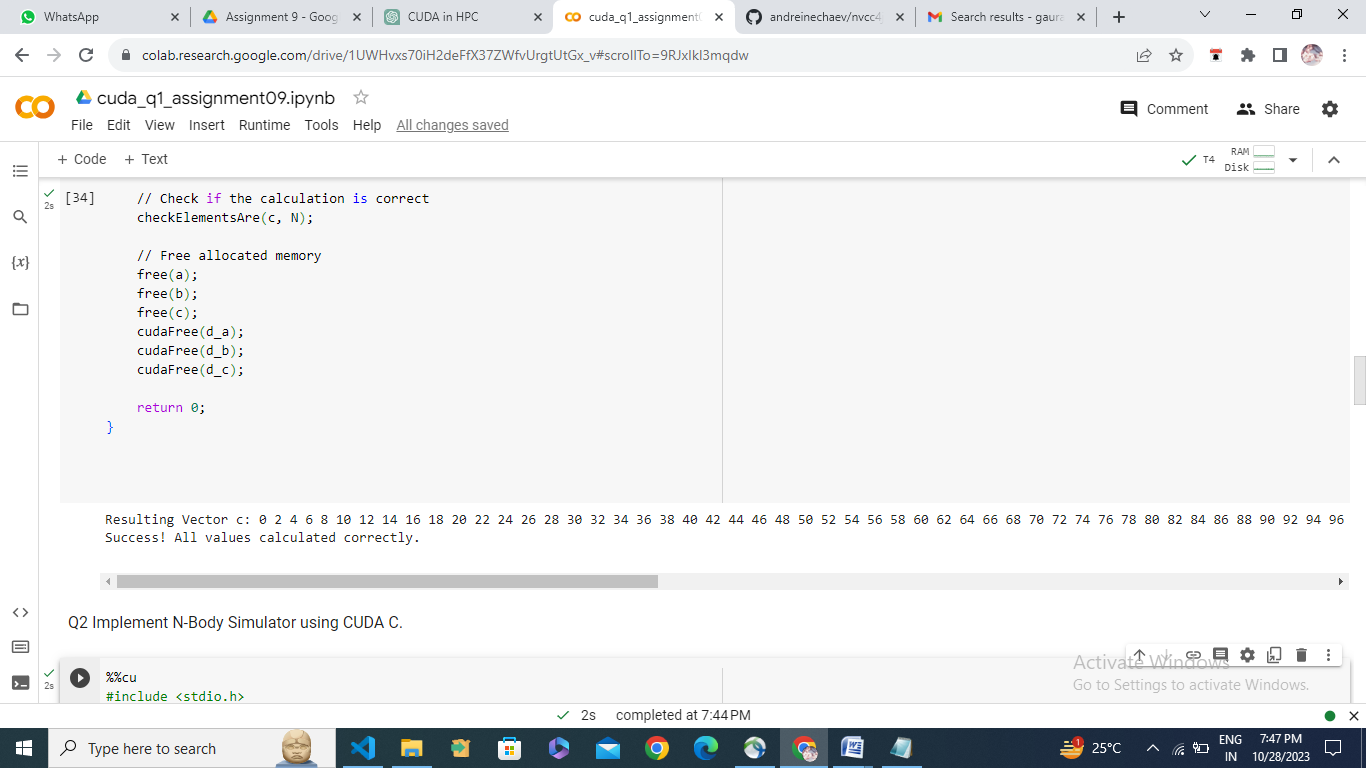
    cudaFree(d\_c);

    return 0;

}

**Screenshots:**

**Output:**

****

Speedup analysis: Tabular

Speedup analysis:

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Data Size** | **Execution time** |
| N/2 | 100 | 0.195875 |
| N | 100 | 0.103090 |
| N/2 | 1000 | 0.180973 |
| N | 1000 | 0.099082 |
| N/2 | 2000 | 0.185451 |
| N | 2000 | 0.117914 |
| N/2 | 5000 | 0.192826 |
| N | 5000 | 0.092011 |

For performing addition of two vectors, two different ways are implemented.

i) One block is used consisting of n threads. In this case, each thread will perform the addition of one element each.

ii) One block is used consisting of n/2 threads. So, in this case, each thread needs to execute the addition of two elements.

On performing the comparison, we can see that the execution time for N threads is better than N/2 threads as here we are using dedicated thread for each element of the vector.

Link of source file with output:- [https://colab.research.google.com/drive/1UWHvxs70iH2deFfX37ZWfvUrgtUtGx\_v#scrollTo=9RJxIkI3mqdw](https://colab.research.google.com/drive/1UWHvxs70iH2deFfX37ZWfvUrgtUtGx_v%23scrollTo=9RJxIkI3mqdw)

**Problem Statement 2:**

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

%%cu

#include <stdio.h>

#include <math.h>

const float G = 6.67430e-11; // Gravitational constant

const float SOFTENING = 1e-9; // Softening factor to avoid singularities

\_\_global\_\_ void computeForces(float\* positions, float\* forces, int numParticles, float\* mass) {

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

    if (idx < numParticles) {

        float myPositionX = positions[2\*idx];

        float myPositionY = positions[2\*idx+1];

        forces[2\*idx] = 0.0f;

        forces[2\*idx+1] = 0.0f;

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

            if (j != idx) {

                float deltaX = positions[2\*j] - myPositionX;

                float deltaY = positions[2\*j+1] - myPositionY;

                float dist = sqrt(deltaX\*deltaX + deltaY\*deltaY);

                float force = G \* mass[idx] \* mass[j] / (dist \* dist + SOFTENING\*SOFTENING);

                forces[2\*idx] += force \* deltaX / dist;

                forces[2\*idx+1] += force \* deltaY / dist;

            }

        }

    }

}

int main() {

    const int numParticles = 1000;

    const int numIterations = 1000;

    float\* h\_positions;

    float\* h\_forces;

    float\* d\_positions;

    float\* d\_forces;

    float\* d\_mass;

    size\_t size = 2 \* numParticles \* sizeof(float);

    h\_positions = (float\*)malloc(size);

    h\_forces = (float\*)malloc(size);

    // Initialize positions and masses (for simplicity, all masses are set to 1)

    for (int i = 0; i < 2 \* numParticles; i++) {

        h\_positions[i] = rand() / (float)RAND\_MAX;

    }

    float\* h\_mass = (float\*)malloc(numParticles \* sizeof(float));

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

        h\_mass[i] = 1.0f;

    }

    cudaMalloc(&d\_positions, size);

    cudaMalloc(&d\_forces, size);

    cudaMalloc(&d\_mass, numParticles \* sizeof(float));

    cudaMemcpy(d\_positions, h\_positions, size, cudaMemcpyHostToDevice);

    cudaMemcpy(d\_mass, h\_mass, numParticles \* sizeof(float), cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;

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

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

        computeForces<<<blocksPerGrid, threadsPerBlock>>>(d\_positions, d\_forces, numParticles, d\_mass);

        // Update positions based on forces and velocities

        // You should implement this based on your specific scenario.

        // Reset forces for the next iteration

        cudaMemset(d\_forces, 0, size);

    }

    cudaMemcpy(h\_positions, d\_positions, size, cudaMemcpyDeviceToHost);

    // Print out the positions after the simulation

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

        printf("Particle %d: x = %f, y = %f\n", i, h\_positions[2\*i], h\_positions[2\*i+1]);

    }

    // Clean up

    free(h\_positions);

    free(h\_forces);

    free(h\_mass);

    cudaFree(d\_positions);

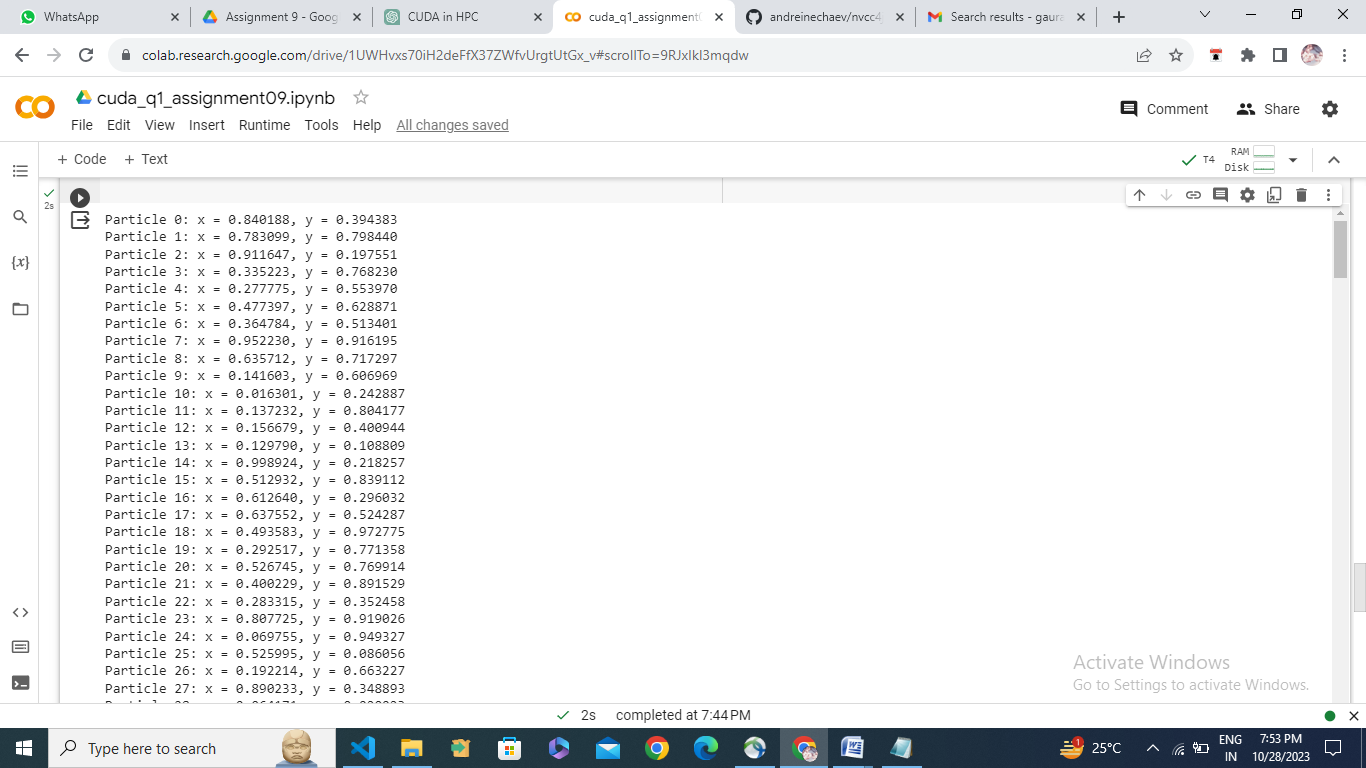
    cudaFree(d\_forces);

    cudaFree(d\_mass);

    return 0;

}

**Output:**



Link of source file with output:- [https://colab.research.google.com/drive/1UWHvxs70iH2deFfX37ZWfvUrgtUtGx\_v#scrollTo=9RJxIkI3mqdw](https://colab.research.google.com/drive/1UWHvxs70iH2deFfX37ZWfvUrgtUtGx_v%23scrollTo=9RJxIkI3mqdw)