**High Performance Computing Lab**

**Practical No. 8**

**Name:** Sahil Santosh Otari

**PRN:** 2020BTECS00025

**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.

Code:

%% cu

#include <stdio.h>

\_\_global\_\_ void addVector(int \*v1, int \*v2, int \*result, int N)

{

    int i = threadIdx.x;

    if (i < N)

    {

        result[i] = v1[i] + v2[i];

    }

}

int main()

{

    int N = 100;

    int v1[N], v2[N], result[N];

    for (int i = 0; i < N; i++)

    {

        v1[i] = 1;

        v2[i] = 2;

    }

    //  initializing poiters for device vectors

    int \*d\_v1, \*d\_v2, \*d\_result;

    // allocating memory for the device vectors

    cudaMalloc(&d\_v1, N \* sizeof(int));

    cudaMalloc(&d\_v2, N \* sizeof(int));

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

    // copying from host to device

    cudaMemcpy(d\_v1, v1, N \* sizeof(int), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_v2, v2, N \* sizeof(int), cudaMemcpyHostToDevice);

    addVector<<<1, N>>>(d\_v1, d\_v2, d\_result, N);

    cudaDeviceSynchronize();

    // copying from device to host

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

    for (int i = 0; i < N; i++)

    {

        printf("%d ", result[i]);

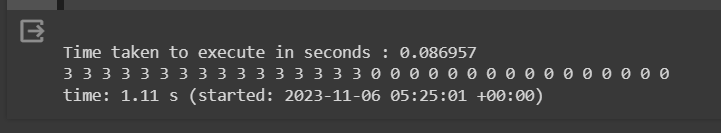
    }

    return 0;

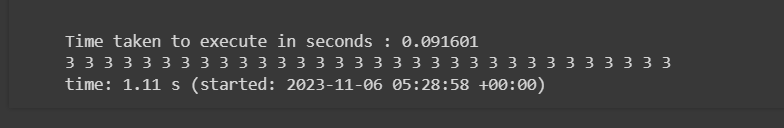
}

**Result:**

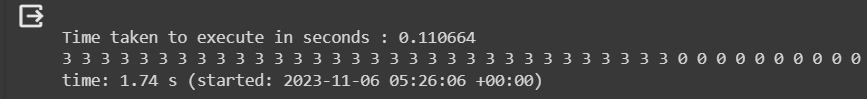
Problem size = 32, number of threads = n/2



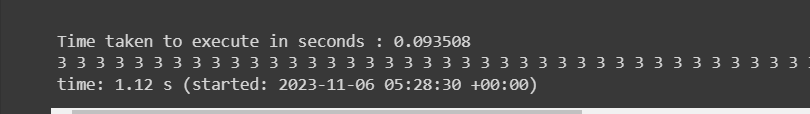
Problem size = 32, number of threads = n



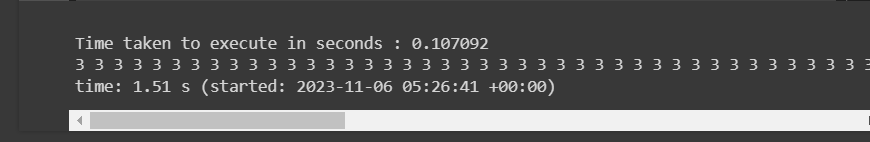
Problem size = 64, number of threads = n/2



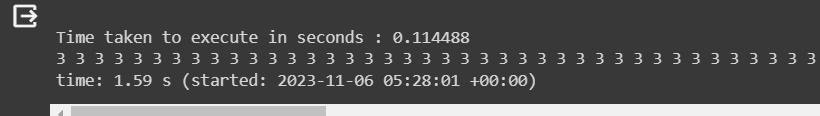
Problem size = 64, number of threads = n



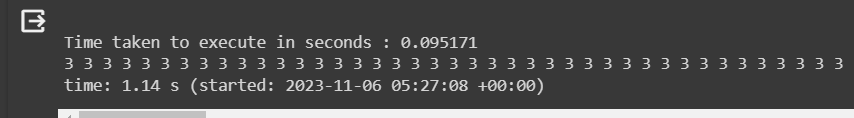
Problem size = 128, number of threads = n/2



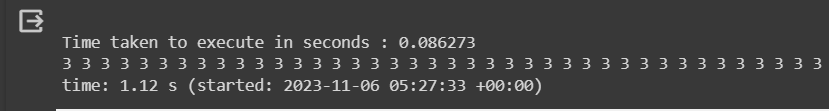
Problem size = 128, number of threads = n



Problem size = 256, number of threads = n/2



Problem size = 256, number of threads = n



**Speedup analysis:**

|  |  |  |
| --- | --- | --- |
| **Number of threads** | **Data Size** | **Execution time** |
| N/2 | 32 | 0.086957 |
| N | 32 | 0.091601 |
| N/2 | 64 | 0.110664 |
| N | 64 | 0.093508 |
| N/2 | 128 | 0.107092 |
| N | 128 | 0.114488 |
| N/2 | 256 | 0.095071 |
| N | 256 | 0.086273 |

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.

**Problem Statement 2:**

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

**Code:**

%% cu

#include <stdio.h>

#include <math.h>

#include <stdlib.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 = 100;

    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;

}

Result:

