Name: Mrunal Khade

PRN No: 2020BTECS00057

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.

Screenshots:

```
%%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 pointers 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]);
```

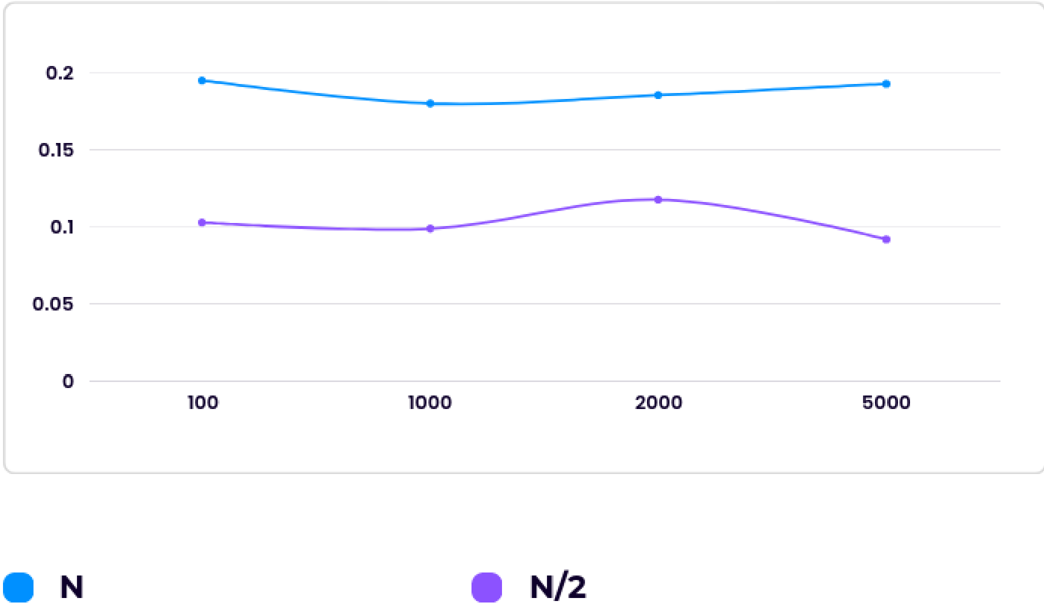
```
    }

    return 0;

}
```

Output:

```
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

Speedup analysis: Tabular and Graphical

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



Analysis:

i) A single block is employed, containing 'n' threads. Each thread is responsible for adding a single element from the vectors.

ii) A single block is utilized, comprising 'n/2' threads. In this scenario, each thread is tasked with adding two elements.

When comparing these two approaches, it becomes evident that the execution time for 'n' threads outperforms 'n/2' threads because it allows each element of the vector to be processed by a dedicated thread.

Problem Statement 2:

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

Screenshots: N-Body Simulator

```
%%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;
}
```

Output:

```
Particle 43: x = 0.228968, y = 0.893372
Particle 44: x = 0.350360, y = 0.686670
Particle 45: x = 0.956468, y = 0.588640
Particle 46: x = 0.657304, y = 0.858676
Particle 47: x = 0.439560, y = 0.923970
Particle 48: x = 0.398437, y = 0.814767
Particle 49: x = 0.684219, y = 0.910972
Particle 50: x = 0.482491, y = 0.215825
Particle 51: x = 0.950252, y = 0.920128
Particle 52: x = 0.147660, y = 0.881062
Particle 53: x = 0.641081, y = 0.431953
Particle 54: x = 0.619596, y = 0.281059
Particle 55: x = 0.786002, y = 0.307458
Particle 56: x = 0.447034, y = 0.226107
Particle 57: x = 0.187533, y = 0.276235
Particle 58: x = 0.556444, y = 0.416501
Particle 59: x = 0.169607, y = 0.906804
Particle 60: x = 0.103171, y = 0.126075
Particle 61: x = 0.495444, y = 0.760475
Particle 62: x = 0.984752, y = 0.935004
Particle 63: x = 0.684445, y = 0.383188
Particle 64: x = 0.749771, y = 0.368664
Particle 65: x = 0.294160, y = 0.232262
Particle 66: x = 0.584489, y = 0.244413
Particle 67: x = 0.152390, y = 0.732149
Particle 68: x = 0.125475, y = 0.793470
Particle 69: x = 0.164102, y = 0.745071
Particle 70: x = 0.074530, y = 0.950104
Particle 71: x = 0.052529, y = 0.521563
Particle 72: x = 0.176211, y = 0.240062
Particle 73: x = 0.797798, y = 0.732654
Particle 74: x = 0.656564, y = 0.967405
Particle 75: x = 0.639458, y = 0.759735
Particle 76: x = 0.093480, y = 0.134902
Particle 77: x = 0.520210, y = 0.078232
```