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

```
Time taken to execute in seconds : 0.086957
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
time: 1.11 s (started: 2023-11-06 05:25:01 +00:00)
```

Problem size = 32, number of threads = n

```
Time taken to execute in seconds : 0.091601
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
time: 1.11 s (started: 2023-11-06 05:28:58 +00:00)
```

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

```
Time taken to execute in seconds : 0.110664
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 0 0 0 0 0 0 0 0 0 0
time: 1.74 s (started: 2023-11-06 05:26:06 +00:00)
```

Problem size = 64, number of threads = n

```
Time taken to execute in seconds : 0.093508
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
time: 1.12 s (started: 2023-11-06 05:28:30 +00:00)
```

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

```
Time taken to execute in seconds : 0.107092
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
time: 1.51 s (started: 2023-11-06 05:26:41 +00:00)
```

Problem size = 128, number of threads = n

```
Time taken to execute in seconds : 0.114488
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
time: 1.59 s (started: 2023-11-06 05:28:01 +00:00)
```

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

```
Time taken to execute in seconds : 0.095171
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
time: 1.14 s (started: 2023-11-06 05:27:08 +00:00)
```

Problem size = 256, number of threads = n

```
Time taken to execute in seconds : 0.086273
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
time: 1.12 s (started: 2023-11-06 05:27:33 +00:00)
```

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:

```
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
Particle 78: x = 0.069906, y = 0.204655
Particle 79: x = 0.461420, y = 0.819677
Particle 80: x = 0.573319, y = 0.755581
Particle 81: x = 0.051939, y = 0.157807
Particle 82: x = 0.999994, y = 0.204329
Particle 83: x = 0.889956, y = 0.125468
Particle 84: x = 0.997799, y = 0.054058
Particle 85: x = 0.870540, y = 0.072329
Particle 86: x = 0.004162, y = 0.923069
Particle 87: x = 0.593892, y = 0.180372
Particle 88: x = 0.163132, y = 0.391690
Particle 89: x = 0.913027, y = 0.819695
Particle 90: x = 0.359095, y = 0.552485
Particle 91: x = 0.579430, y = 0.452576
Particle 92: x = 0.687387, y = 0.099640
Particle 93: x = 0.530808, y = 0.757294
Particle 94: x = 0.304295, y = 0.992228
Particle 95: x = 0.576971, y = 0.877614
Particle 96: x = 0.747809, y = 0.628910
Particle 97: x = 0.035421, y = 0.747803
Particle 98: x = 0.833239, y = 0.925377
Particle 99: x = 0.873271, y = 0.831038

time: 1.44 s (started: 2023-10-28 14:29:41 +00:00)
```