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

**Year:** 2022-23 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 9**

**Exam Seat No:**

2019BTECS00037 – Onkar Gavali

**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.

#include <stdio.h>

void initWith(float num, float \*a, int N)

{

for(int i = 0; i < N; ++i)

{

a[i] = num;

}

}

/\*

\* Device kernel stores into `result` the sum of each

\* same-indexed value of `a` and `b`.

\*/

\_\_global\_\_ void addVectorsInto(float \*result, float \*a, float \*b, int N)

{

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

int stride = blockDim.x \* gridDim.x;

for(int i = index; i < N; i += stride)

{

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

}

}

void checkElementsAre(float target, float \*vector, int N)

{

for(int i = 0; i < N; i++)

{

if(vector[i] != target)

{

printf("FAIL: vector[%d] - %0.0f does not equal %0.0f\n", i, vector[i], target);

exit(1);

}

}

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

}

int main()

{

const int N = 2<<24;

size\_t size = N \* sizeof(float);

float \*a;

float \*b;

float \*c;

cudaMallocManaged(&a, size);

cudaMallocManaged(&b, size);

cudaMallocManaged(&c, size);

addVectorsInto<<<8,32>>>(c,a,b,N);

cudaDeviceSynchronize();

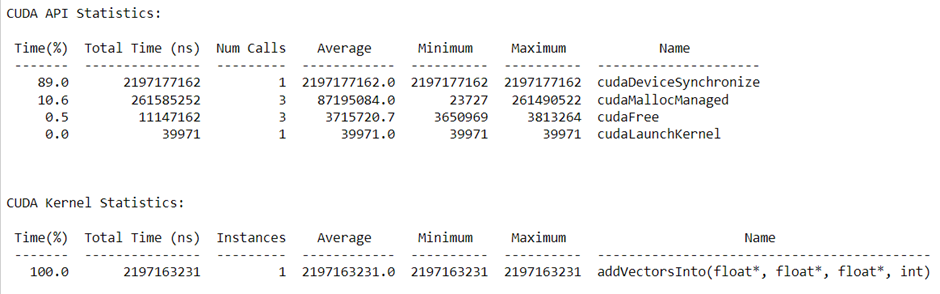
cudaFree(a);

cudaFree(b);

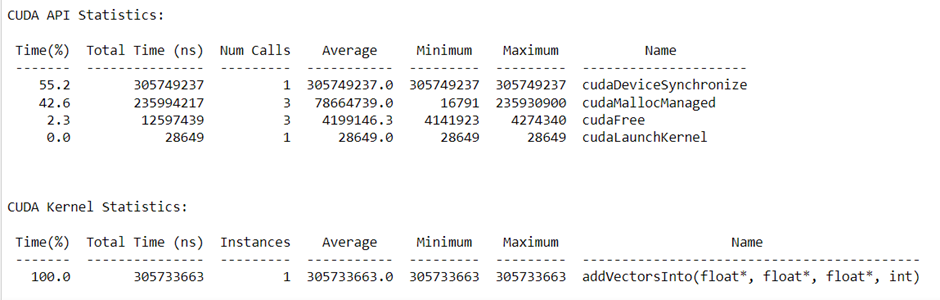
cudaFree(c);

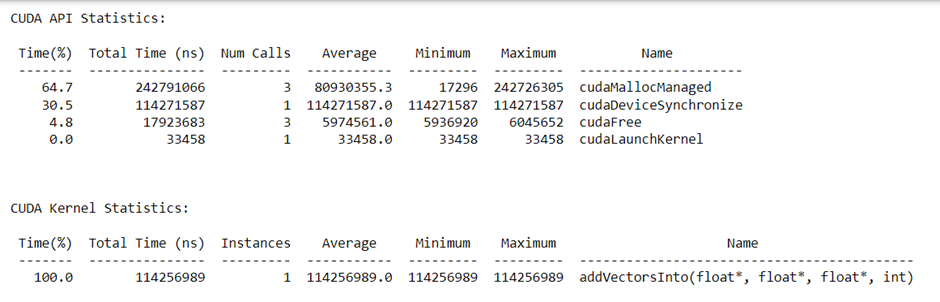
}

Answer: For <<<1,1>>>

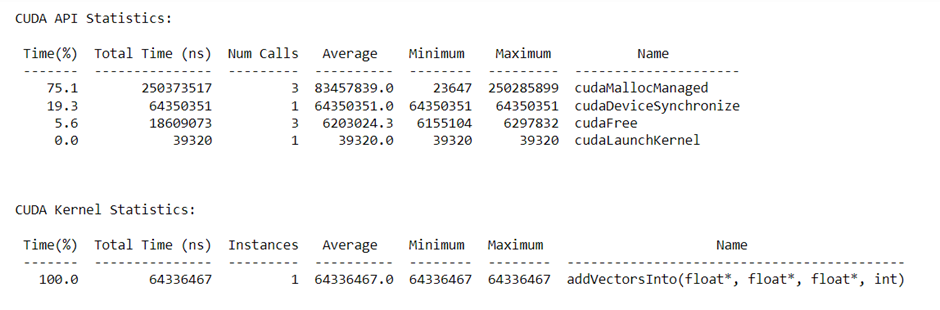


For <<<2,32>>>

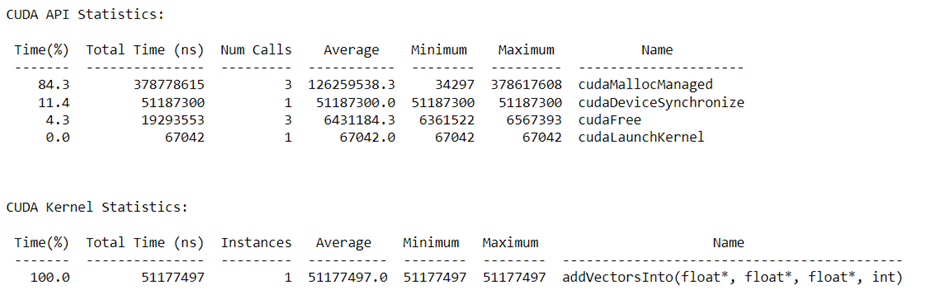


For <<<4,64>>>

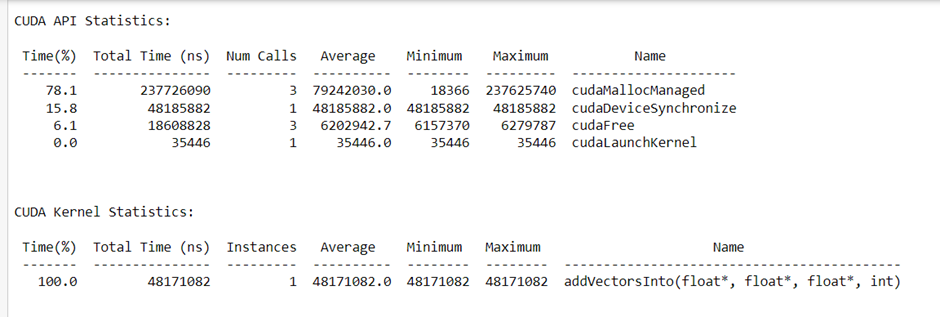
For <<<2,1024>>>



For<<<64,32>>> :



For <<<64,1024>>> :



N=2^24

| No of Blocks | No of Threads | Total Time | Speedup |
| --- | --- | --- | --- |
| 1 | 1 | 2198891617 | - |
| 2 | 32 | 305733663 | 7.1865 |
| 2 | 1024 | 64336467 | 34.1779 |
| 64 | 32 | 51177497 | 42.96 |
| 64 | 1024 | 48171082 | 45.647 |

N=2^16

| No of Blocks | No of Threads | Total Time | Speedup |
| --- | --- | --- | --- |
| 1 | 1 | 19129450 | - |
| 2 | 32 | 1658524 | 11.534 |
| 2 | 1024 | 674578 | 28.357 |
| 64 | 32 | 688595 | 27.780 |
| 64 | 1024 | 610708 | 31.323 |

N=2^10

| No of Blocks | No of Threads | Total Time | Speedup |
| --- | --- | --- | --- |
| 1 | 1 | 756945 | - |
| 2 | 32 | 246332 | 3.07 |
| 2 | 1024 | 491415 | 1.54 |
| 64 | 32 | 265371 | 2.852 |
| 64 | 1024 | 615062 | 1.230 |

Conclusion :

From above graph we can conclude that

- For vector-vector addition problem for larger values of N time for parallel execution decreases exponentially as we increase number of threads

- For vector-vector addition problem for smaller values of N time first decreases and then increases exponentially as we increase number of threads

- Parallel execution is preferred for higher values of N

**Problem Statement 2:**

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

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include "timer.h"

#include "files.h"

#define SOFTENING 1e-9f

/\*

\* Each body contains x, y, and z coordinate positions,

\* as well as velocities in the x, y, and z directions.

\*/

typedef struct { float x, y, z, vx, vy, vz; } Body;

/\*

\* Calculate the gravitational impact of all bodies in the system

\* on all others.

\*/

\_\_global\_\_ void bodyForce(Body \*p, float dt, int n) {

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

//int stride = gridDim.x \* blockDim.x;

if( i < n){

float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;

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

float dx = p[j].x - p[i].x;

float dy = p[j].y - p[i].y;

float dz = p[j].z - p[i].z;

float distSqr = dx\*dx + dy\*dy + dz\*dz + SOFTENING;

float invDist = rsqrtf(distSqr);

float invDist3 = invDist \* invDist \* invDist;

Fx += dx \* invDist3; Fy += dy \* invDist3; Fz += dz \* invDist3;

}

p[i].vx += dt\*Fx; p[i].vy += dt\*Fy; p[i].vz += dt\*Fz;

}

}

int main(const int argc, const char\*\* argv) {

// The assessment will test against both 2<11 and 2<15.

// Feel free to pass the command line argument 15 when you generate ./nbody report files

int nBodies = 2<<11;

if (argc > 1) nBodies = 2<<atoi(argv[1]);

// The assessment will pass hidden initialized values to check for correctness.

// You should not make changes to these files, or else the assessment will not work.

const char \* initialized\_values;

const char \* solution\_values;

if (nBodies == 2<<11) {

initialized\_values = "09-nbody/files/initialized\_4096";

solution\_values = "09-nbody/files/solution\_4096";

} else { // nBodies == 2<<15

initialized\_values = "09-nbody/files/initialized\_65536";

solution\_values = "09-nbody/files/solution\_65536";

}

if (argc > 2) initialized\_values = argv[2];

if (argc > 3) solution\_values = argv[3];

const float dt = 0.01f; // Time step

const int nIters = 10; // Simulation iterations

int bytes = nBodies \* sizeof(Body);

float \*buf;

cudaMallocManaged(&buf, bytes);

Body \*p = (Body\*)buf;

read\_values\_from\_file(initialized\_values, buf, bytes);

double totalTime = 0.0;

/\*

\* This simulation will run for 10 cycles of time, calculating gravitational

\* interaction amongst bodies, and adjusting their positions to reflect.

\*/

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

StartTimer();

/\*

\* You will likely wish to refactor the work being done in `bodyForce`,

\* and potentially the work to integrate the positions.

\*/

size\_t threads\_per\_block = 256;

size\_t number\_of\_blocks = (nBodies + threads\_per\_block - 1) / threads\_per\_block;

bodyForce<<<number\_of\_blocks, threads\_per\_block>>>(p, dt, nBodies); // compute interbody forces

cudaDeviceSynchronize();

/\*

\* This position integration cannot occur until this round of `bodyForce` has completed.

\* Also, the next round of `bodyForce` cannot begin until the integration is complete.

\*/

for (int i = 0 ; i < nBodies; i++) { // integrate position

p[i].x += p[i].vx\*dt;

p[i].y += p[i].vy\*dt;

p[i].z += p[i].vz\*dt;

}

const double tElapsed = GetTimer() / 1000.0;

totalTime += tElapsed;

}

double avgTime = totalTime / (double)(nIters);

float billionsOfOpsPerSecond = 1e-9 \* nBodies \* nBodies / avgTime;

write\_values\_to\_file(solution\_values, buf, bytes);

// You will likely enjoy watching this value grow as you accelerate the application,

// but beware that a failure to correctly synchronize the device might result in

// unrealistically high values.

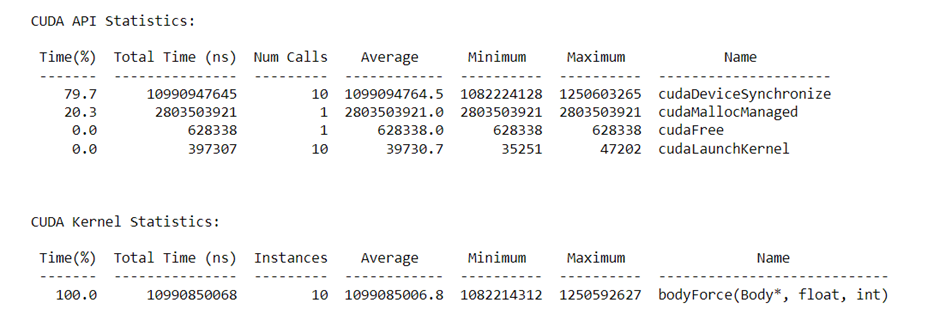
printf("%0.3f Billion Interactions / second\n", billionsOfOpsPerSecond);

cudaFree(buf);

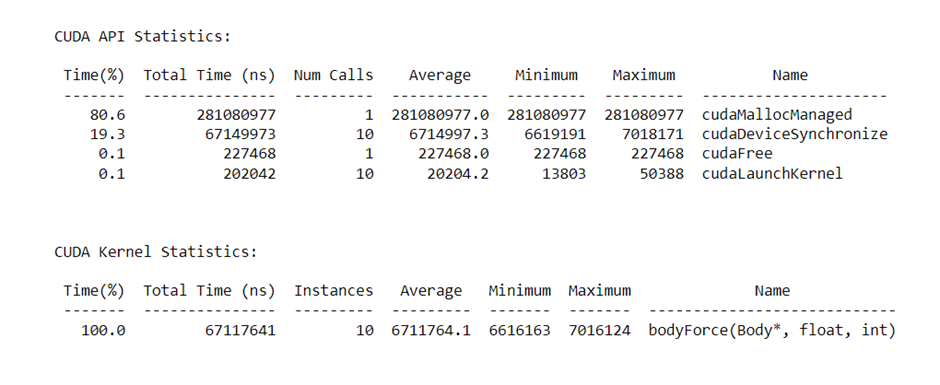
}

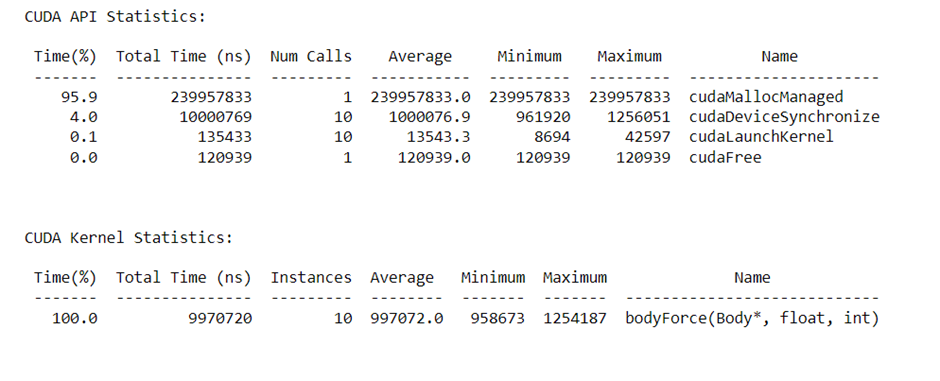
For N = 2^10

For <<<1,1>>>

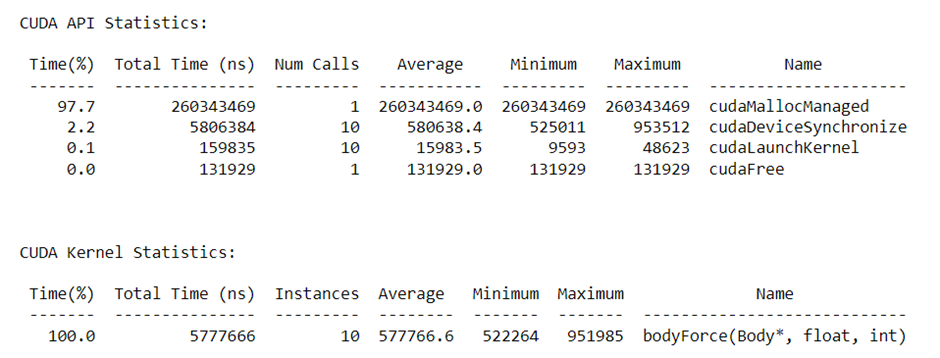


For <<<2,32>>> :

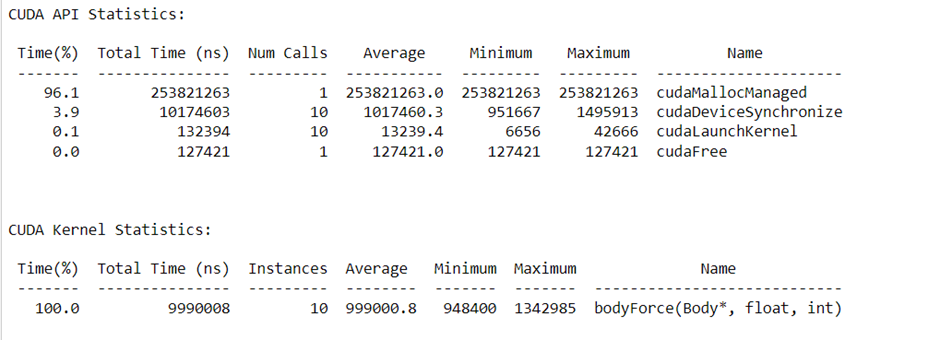
  
For <<<2,1024>>> :

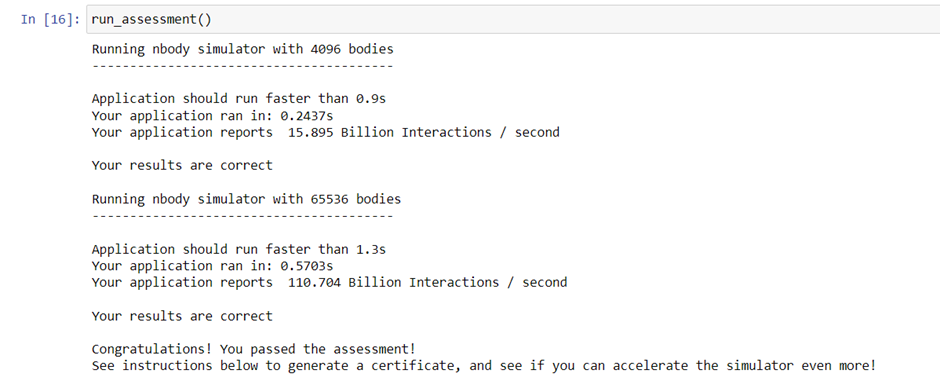


For <<<64,32>>> :



For <<<64,1024>>> :





| No of Blocks | No of Threads | Total Time | Speedup |
| --- | --- | --- | --- |
| 1 | 1 | 10990850068 | - |
| 2 | 32 | 6711764.1 | 1637.55 |
| 2 | 1024 | 9970720 | 1102.3125 |
| 64 | 32 | 5777666 | 1902.29 |
| 64 | 1024 | 9990008 | 1100.1843 |

Github Link:

<https://github.com/OnkarGavali/HPC_Lab/tree/main/Practical_No9>