Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

High Performance Computing Lab

Class: Final Year (Computer Science and Engineering) **Year:** 2022-23

PRN: 2019BTECS00089 – Piyush Pramod Mhaske **Batch**: B3

Practical 9

Github link: https://github.com/Piyush4620/2019BTECS00089HPCLab

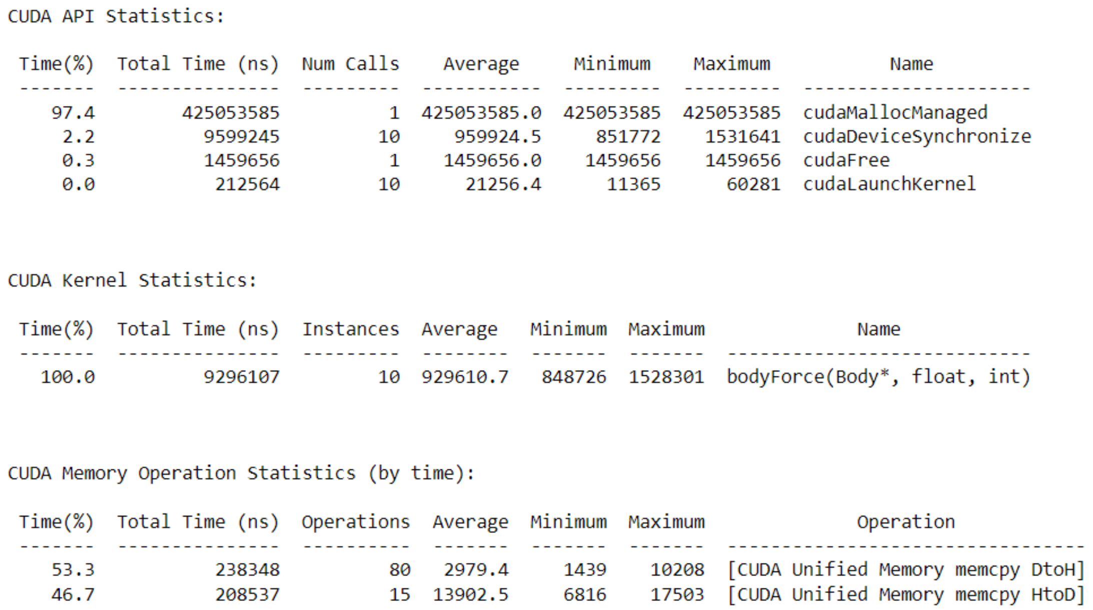
Hosted Link : https://better-sidecar-c10.notion.site/HPC-038e2693a633408c8604841fc50f74e2

Que 1 : Nbody

```
#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 tid = blockIdx.x * blockDim.x + threadIdx.x;

   if (tid < 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[tid].x;
      float dy = p[j].y - p[tid].y;
      float dz = p[j].z - p[tid].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[tid].vx += dt*Fx; p[tid].vy += dt*Fy; p[tid].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.
   */
```

```
    int threads_per_block = 128;
    int number_of_blocks = (nBodies / 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);

}
```

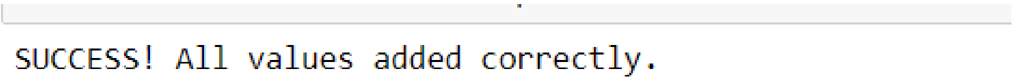


```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average      Minimum    Maximum           Name
 -------  ---------------  ---------  -----------  ---------  ---------  ---------------------
    97.4        425053585          1  425053585.0  425053585  425053585  cudaMallocManaged
     2.2          9599245         10     959924.5     851772    1531641  cudaDeviceSynchronize
     0.3          1459656          1    1459656.0    1459656    1459656  cudaFree
     0.0           212564         10      21256.4      11365      60281  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances  Average   Minimum  Maximum              Name
 -------  ---------------  ---------  --------  -------  -------  ----------------------------
   100.0          9296107         10  929610.7   848726  1528301  bodyForce(Body*, float, int)


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum              Operation
 -------  ---------------  ----------  -------  -------  -------  ---------------------------------
    53.3           238348          80   2979.4     1439    10208  [CUDA Unified Memory memcpy DtoH]
    46.7           208537          15  13902.5     6816    17503  [CUDA Unified Memory memcpy HtoD]
```

Vector Addition:

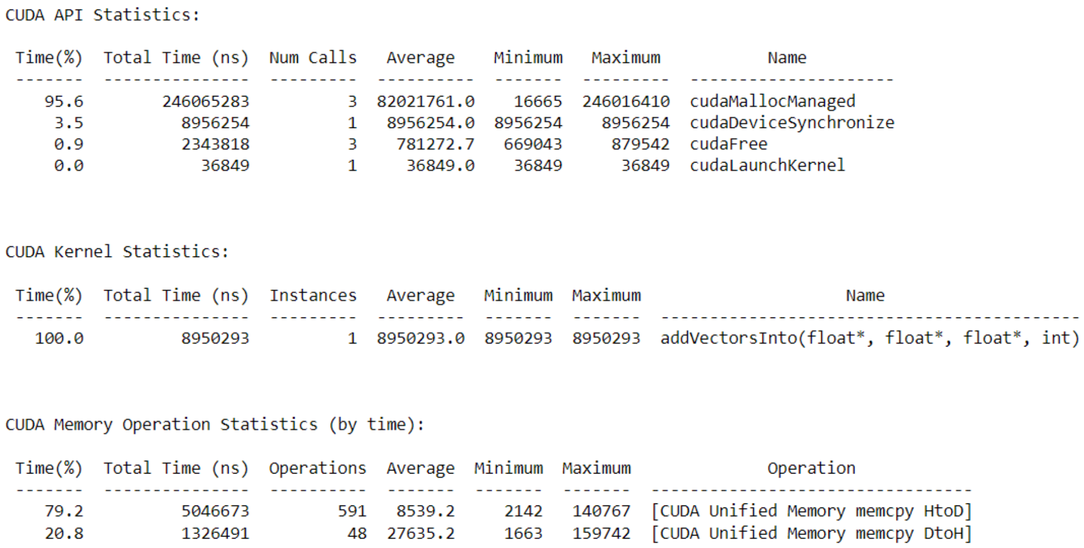
```
#include <stdio.h>

void initWith(float num, float *a, int N)
{
    for (int i = 0; i < N; ++i)
    {
        a[i] = num;
    }
}

__global__ void addVectorsInto(float *result, float *a, float *b, int N)
{
    int start = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = gridDim.x * blockDim.x;

    for (int i = start; i < N; i += stride)
    {
        result[i] = a[i] + b[i];
    }
}

void checkElementsAre(float target, float *array, int N)
{
    for (int i = 0; i < N; i++)
    {
        if (array[i] != target)
        {
            printf("FAIL: array[%d] - %0.0f does not equal %0.0f\n", i, array[i], target);
            exit(1);
        }
    }
    printf("SUCCESS! All values added correctly.\n");
}

int main()
{
    const int N = 2 << 20;
    size_t size = N * sizeof(float);

    float *a;
    float *b;
    float *c;

    cudaMallocManaged(&a, size);
    cudaMallocManaged(&b, size);
    cudaMallocManaged(&c, size);

    initWith(3, a, N);
    initWith(4, b, N);
    initWith(0, c, N);

    addVectorsInto<<<100, 1024>>>(c, a, b, N);
    cudaDeviceSynchronize();
    checkElementsAre(7, c, N);

    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
}
```

Output:



```
SUCCESS! All values added correctly.
```

Profiling ;



```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls   Average    Minimum   Maximum           Name
 -------  ---------------  ---------  ----------  -------  ---------  ---------------------
    95.6        246065283          3  82021761.0    16665  246016410  cudaMallocManaged
     3.5          8956254          1   8956254.0  8956254    8956254  cudaDeviceSynchronize
     0.9          2343818          3    781272.7   669043     879542  cudaFree
     0.0            36849          1     36849.0    36849      36849  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                   Name
 -------  ---------------  ---------  ---------  -------  -------  -------------------------------------------
   100.0          8950293          1  8950293.0  8950293  8950293  addVectorsInto(float*, float*, float*, int)


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum              Operation
 -------  ---------------  ----------  -------  -------  -------  ---------------------------------
    79.2          5046673         591   8539.2     2142   140767  [CUDA Unified Memory memcpy HtoD]
    20.8          1326491          48  27635.2     1663   159742  [CUDA Unified Memory memcpy DtoH]
```