Final Year B. Tech, Sem VII 2022-23
PRN – 2020BTECS00211
Name – Aashita Narendra Gupta
High Performance Computing Lab
Batch: B4

Practical no – 9

Github Link for Code - https://github.com/Aashita06/HPC_Practicals

Q.1) Implement Vector-Vector addition using CUDA C. State and justify the speedup using different size of threads and blocks.

→

Code:

```
%%cu
#include <math.h>
#include <time.h>
#include <iostream>
#include <stdexcept>
#include "cuda_runtime.h"

// declare the vectors' number of elements and their size in bytes
static const int n_el = 10000000; // 10 millions
static const size_t size = n_el * sizeof(float);

// function for computing sum on CPU
void CPU_sum(const float* A, const float* B, float* C, int n_el) {
  for (int i=0; i<n_el; i++) {
    C[i]=A[i]+B[i];
  }
}

// kernel
__global__ void kernel_sum(const float* A, const float* B, float* C, in
t n_el)
{
  // calculate the unique thread index
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  // perform tid-th elements addition
  if (tid < n_el) C[tid] = A[tid] + B[tid];
}

// function which invokes the kernel
void GPU_sum(const float* A, const float* B, float* C, int n_el) {
```

```
  // declare the number of blocks per grid and the number of threads pe
r block
  int threadsPerBlock,blocksPerGrid;

  // use max 512 threads per block
  threadsPerBlock = min(512,n_el);
  blocksPerGrid   = ceil(double(n_el)/double(threadsPerBlock));

  // invoke the kernel
  kernel_sum<<<blocksPerGrid,threadsPerBlock>>>(A, B, C, n_el);
}

int main(){
  // declare and allocate input vectors h_A and h_B in the host (CPU) m
emory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  float* h_C = (float*)malloc(size);

  // initialize input vectors
  for (int i=0; i<n_el; i++){
    h_A[i]=sin(i);
    h_B[i]=cos(i);
  }

  /************ CPU Version ***********/

  clock_t tstart,tend;
  float cpu_duration;
  // compute on CPU
  tstart = clock();

  /////////////////////////////////
  // call kernel function
  /////////////////////////////////
  CPU_sum(h_A, h_B, h_C, n_el);
  /////////////////////////////////

  tend = clock();
  cpu_duration = ((float)(tend-tstart))/CLOCKS_PER_SEC;
  printf("Total  time for sum on CPU: %f seconds\n",cpu_duration);

  /************ GPU Version ***********/

  clock_t tstart_total;
  tstart_total = clock();
```

```
  //////////////////////////////////
  // transfer data from CPU to GPU
  //////////////////////////////////
  // declare device vectors in the device (GPU) memory
  float *d_A,*d_B,*d_C;
  // allocate device vectors in the device (GPU) memory
  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);
  // copy input vectors from the host (CPU) memory to the device (GPU)
memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  float gpu_duration;
  tstart = clock();

  //////////////////////////////////
  // call kernel function
  //////////////////////////////////
  GPU_sum(d_A, d_B, d_C, n_el);
  // wait for everything to finish
  cudaDeviceSynchronize();
  //////////////////////////////////

  tend = clock();
  gpu_duration = ((float)(tend-tstart))/CLOCKS_PER_SEC;
  printf("Kernel time for sum on GPU: %f seconds\n",gpu_duration);

  //////////////////////////////////
  // transfer data from GPU to CPU
  //////////////////////////////////
  // copy the output (results) vector from the device (GPU) memory to t
he host (CPU) memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  // free device memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);

  // wait for everything to finish
  cudaDeviceSynchronize();

  tend = clock();
  gpu_duration = ((float)(tend-tstart_total))/CLOCKS_PER_SEC;
  printf("Total  time for sum on GPU: %f seconds\n",gpu_duration);

  /************ Check correctness using RMS Error ***********/
```

```
  // compute the squared error of the result
  // using double precision for good accuracy
  double err=0;
  for (int i=0; i<n_el; i++) {
    double diff=double((h_A[i]+h_B[i])-h_C[i]);
    err+=diff*diff;
    // print results for manual checking.
    //printf("%f=%f,",h_A[i]+h_B[i],h_C[i]);
  }
  // compute the RMS error
  err=sqrt(err/double(n_el));
  printf("error: %f\n",err);

  printf("speed-up: %.2fx",cpu_duration/gpu_duration);

  // free host memory
  free(h_A);
  free(h_B);
  free(h_C);

  return 0;
}
```

Output:

```
Total  time for sum on CPU: 0.045169 seconds
Kernel time for sum on GPU: 0.000527 seconds
Total  time for sum on GPU: 0.239267 seconds
error: 0.000000
speed-up: 0.19x
```

Code:

```
%%cu
#include <math.h>
#include <time.h>
#include <iostream>
#include <stdexcept>
#include "cuda_runtime.h"

// declare the vectors' number of elements and their size in bytes
static const int n_el = 20000000; // 10 millions
static const size_t size = n_el * sizeof(float);

// function for computing sum on CPU
void CPU_sum(const float* A, const float* B, float* C, int n_el) {
  for (int i=0; i<n_el; i++) {
```

```
    C[i]=A[i]+B[i];
  }
}

// kernel
__global__ void kernel_sum(const float* A, const float* B, float* C, in
t n_el)
{
  // calculate the unique thread index
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  // perform tid-th elements addition
  if (tid < n_el) C[tid] = A[tid] + B[tid];
}

// function which invokes the kernel
void GPU_sum(const float* A, const float* B, float* C, int n_el) {

  // declare the number of blocks per grid and the number of threads pe
r block
  int threadsPerBlock,blocksPerGrid;

  // use max 512 threads per block
  threadsPerBlock = min(1024,n_el);
  blocksPerGrid   = ceil(double(n_el)/double(threadsPerBlock));

  // invoke the kernel
  kernel_sum<<<blocksPerGrid,threadsPerBlock>>>(A, B, C, n_el);
}

int main(){
  // declare and allocate input vectors h_A and h_B in the host (CPU) m
emory
  float* h_A = (float*)malloc(size);
  float* h_B = (float*)malloc(size);
  float* h_C = (float*)malloc(size);

  // initialize input vectors
  for (int i=0; i<n_el; i++){
    h_A[i]=sin(i);
    h_B[i]=cos(i);
  }

  /************ CPU Version ***********/

  clock_t tstart,tend;
  float cpu_duration;
  // compute on CPU
  tstart = clock();
```

```
  //////////////////////////////////
  // call kernel function
  //////////////////////////////////
  CPU_sum(h_A, h_B, h_C, n_el);
  //////////////////////////////////

  tend = clock();
  cpu_duration = ((float)(tend-tstart))/CLOCKS_PER_SEC;
  printf("Total  time for sum on CPU: %f seconds\n",cpu_duration);

  /************ GPU Version ***********/

  clock_t tstart_total;
  tstart_total = clock();

  //////////////////////////////////
  // transfer data from CPU to GPU
  //////////////////////////////////
  // declare device vectors in the device (GPU) memory
  float *d_A,*d_B,*d_C;
  // allocate device vectors in the device (GPU) memory
  cudaMalloc(&d_A, size);
  cudaMalloc(&d_B, size);
  cudaMalloc(&d_C, size);
  // copy input vectors from the host (CPU) memory to the device (GPU)
memory
  cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

  float gpu_duration;
  tstart = clock();

  //////////////////////////////////
  // call kernel function
  //////////////////////////////////
  GPU_sum(d_A, d_B, d_C, n_el);
  // wait for everything to finish
  cudaDeviceSynchronize();
  //////////////////////////////////

  tend = clock();
  gpu_duration = ((float)(tend-tstart))/CLOCKS_PER_SEC;
  printf("Kernel time for sum on GPU: %f seconds\n",gpu_duration);

  //////////////////////////////////
  // transfer data from GPU to CPU
  //////////////////////////////////
```

```
  // copy the output (results) vector from the device (GPU) memory to t
he host (CPU) memory
  cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
  // free device memory
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);

  // wait for everything to finish
  cudaDeviceSynchronize();

  tend = clock();
  gpu_duration = ((float)(tend-tstart_total))/CLOCKS_PER_SEC;
  printf("Total  time for sum on GPU: %f seconds\n",gpu_duration);

  /************ Check correctness using RMS Error ***********/

  // compute the squared error of the result
  // using double precision for good accuracy
  double err=0;
  for (int i=0; i<n_el; i++) {
    double diff=double((h_A[i]+h_B[i])-h_C[i]);
    err+=diff*diff;
    // print results for manual checking.
    //printf("%f=%f,",h_A[i]+h_B[i],h_C[i]);
  }
  // compute the RMS error
  err=sqrt(err/double(n_el));
  printf("error: %f\n",err);

  printf("speed-up: %.2fx",cpu_duration/gpu_duration);

  // free host memory
  free(h_A);
  free(h_B);
  free(h_C);

  return 0;
}
```

Output:

```
Total  time for sum on CPU: 0.089007 seconds
Kernel time for sum on GPU: 0.000986 seconds
Total  time for sum on GPU: 0.263134 seconds
error: 0.000000
speed-up: 0.34x
```

Q.2) Implement N-Body Simulator using CUDA C. State and justify the speedup using different size of threads and blocks.

→

Code:

```
#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.
 */

void bodyForce(Body *p, float dt, int n) {
 for (int i = 0; i < n; ++i) {
  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;

  buf = (float *)malloc(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.
   */

    bodyForce(p, dt, nBodies); // compute interbody forces

  /*
   * 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);

  free(buf);
}
```

Output:

```
In [1]: !nvcc -std=c++11 -o nbody 09-nbody/01-nbody.cu
```

It is highly recommended you use the profiler to assist your work. Execute the following cell to generate a report file:

In [2]:
```
!nsys profile --stats=true --force-overwrite=true -o nbody-report ./nbody
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
WARNING: The command line includes a target application therefore the CPU context-switch scope has been set to process-tree.
Collecting data...
0.040 Billion Interactions / second
Processing events...
Saving temporary "/tmp/nsys-report-eed6-03bd-3f28-8cbd.qdstrm" file to disk...

Creating final output files...
Processing [==============================================================100%]
Saved report file to "/tmp/nsys-report-eed6-03bd-3f28-8cbd.qdrep"
Exporting 39 events: [==================================================100%]

Exported successfully to
/tmp/nsys-report-eed6-03bd-3f28-8cbd.sqlite


Operating System Runtime API Statistics:

 Time(%)  Total Time (ns)  Num Calls  Average  Minimum  Maximum   Name
 -------  ---------------  ---------  -------  -------  -------  -------
    32.8            84839          2  42419.5     8537    76302  fopen64
    28.4            73512          1  73512.0    73512    73512  writev
    24.5            63532          1  63532.0    63532    63532  read
    14.3            37129          2  18564.5     2763    34366  fclose

Report file moved to "/dli/task/nbody-report.qdrep"
Report file moved to "/dli/task/nbody-report.sqlite"
```

In [4]:
```
run_assessment()
```

```
Running nbody simulator with 4096 bodies
----------------------------------------

Application should run faster than 0.9s
Your application ran in: 4.1297s
Your application is not yet fast enough
```