Assignment06

From Siqi Liang (liangsiq@usc.edu)

Part I

1. Submit the code

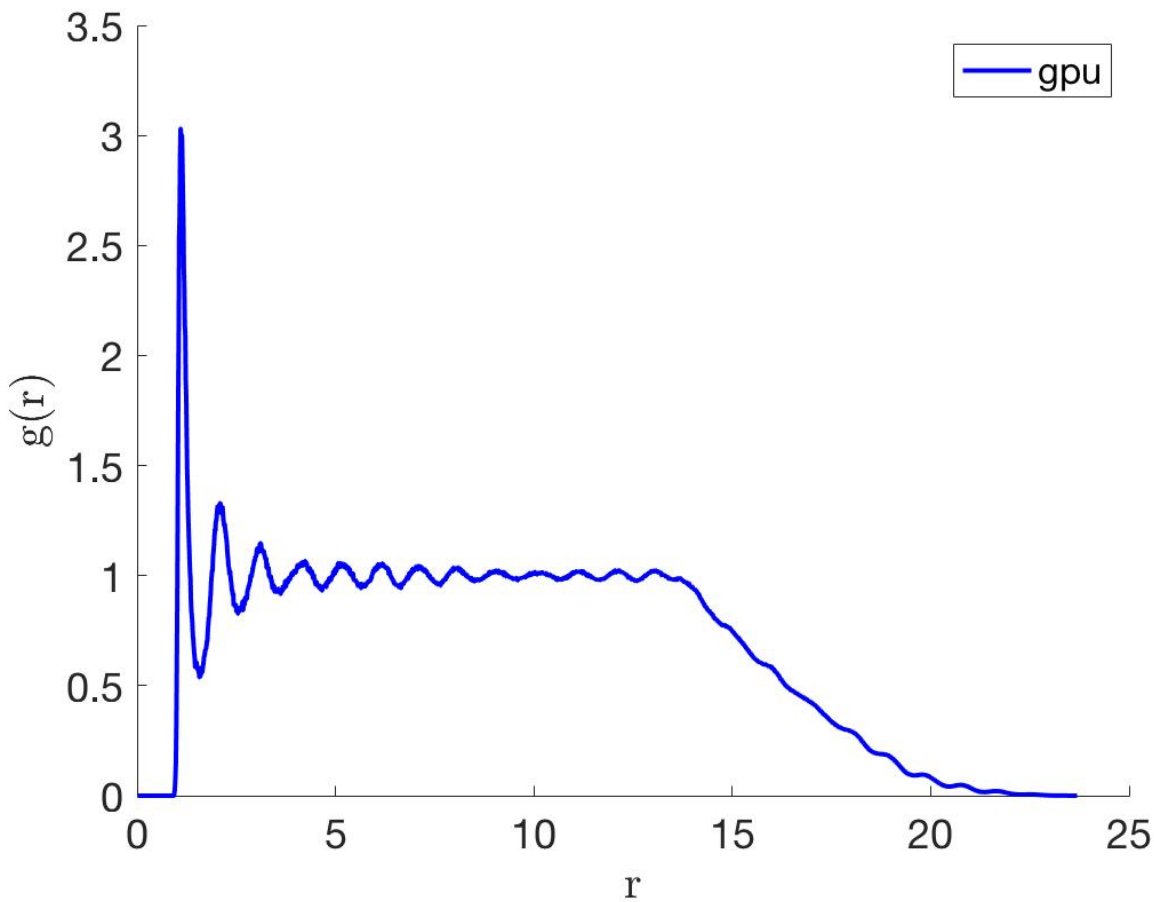
```
/*----------------------------------------------------------------------
Program pdf0.c computes a pair distribution function for n atoms
given the 3D coordinates of the atoms.
--------------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define NHBIN 2000  // Histogram size

float al[3];        // Simulation box lengths
int n;              // Number of atoms
float *r;           // Atomic position array
FILE *fp;

__device__ float d_SignR(float v,float x) {if (x > 0) return v; else return -
v;} // jd

__constant__ float DALTH[3];
__constant__ int DN;
 __constant__ float DDRH;

float SignR(float v,float x) {if (x > 0) return v; else return -v;}

//jd
__global__ void gpu_histogram_kernel(float *r,float *nhis)
{
 int i,j,a, ih;
 float rij,dr;

 int iBlockBegin = (DN/gridDim.x)*blockIdx.x;
 int iBlockEnd = min((DN/gridDim.x)*(blockIdx.x+1),DN);
 int jBlockBegin = (DN/gridDim.y)*blockIdx.y;
```

```
 int jBlockEnd = min((DN/gridDim.y)*(blockIdx.y+1),DN);
 for (i=iBlockBegin+threadIdx.x; i<iBlockEnd; i+=blockDim.x)
 {
 for (j=jBlockBegin+threadIdx.y; j<jBlockEnd; j+=blockDim.y)
 {
    if (i<j) { // Process (i,j) atom pair
    rij = 0.0;
    for (a=0; a<3; a++) {
        dr = r[3*i+a]-r[3*j+a];
        /* Periodic boundary condition */
        dr = dr - d_SignR(DALTH[a],dr-DALTH[a]) -
d_SignR(DALTH[a],dr+DALTH[a]);
        rij += dr*dr;
      }
    rij = sqrt(rij); /* Pair distance */
    ih = rij/DDRH;
   atomicAdd(&nhis[ih],1.0);
   } // end if i<j
 } // end for j
 } // end for i
 }
 //

/*------------------------------------------------------------------------*/
void histogram() {
/*------------------------------------------------------------------------
Constructs a histogram NHIS for atomic-pair distribution.
------------------------------------------------------------------------*/
  float alth[3];
  float* nhis;  // Histogram array
  float rhmax,drh,density,gr;
  int a,ih;

  float* dev_r;     // Atomic positions
  float* dev_nhis;  // Histogram

  /* Half the simulation box size */
  for (a=0; a<3; a++) alth[a] = 0.5*al[a];
  /* Max. pair distance RHMAX & histogram bin size DRH */
  rhmax = sqrt(alth[0]*alth[0]+alth[1]*alth[1]+alth[2]*alth[2]);
  drh = rhmax/NHBIN;  // Histogram bin size

  nhis = (float*)malloc(sizeof(float)*NHBIN);
  //for (ih=0; ih<NHBIN; ih++) nhis[ih] = 0.0; // Reset the histogram

  cudaMalloc((void**)&dev_r,sizeof(float)*3*n);
```

```
  cudaMalloc((void**)&dev_nhis,sizeof(float)*NHBIN);


  cudaMemcpy(dev_r,r,3*n*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemset(dev_nhis,0.0,NHBIN*sizeof(float));
  cudaMemcpyToSymbol(DALTH,alth,sizeof(float)*3,0,cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(DN,&n,sizeof(int),0,cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(DDRH,&drh,sizeof(float),0,cudaMemcpyHostToDevice);

  dim3 numBlocks(8,8,1);
  dim3 threads_per_block(16,16,1);
  gpu_histogram_kernel<<<numBlocks,threads_per_block>>>(dev_r,dev_nhis);//
Compute dev_nhis on GPU: dev_r[] ® dev_nhis[]

  cudaMemcpy(nhis,dev_nhis,NHBIN*sizeof(float),cudaMemcpyDeviceToHost);
  cudaFree(dev_r);
  cudaFree(dev_nhis);

  density = n/(al[0]*al[1]*al[2]);
  /* Print out the histogram */
  fp = fopen("pdf.d","w");
  for (ih=0; ih<NHBIN; ih++) {
    gr = nhis[ih]/(2*M_PI*pow((ih+0.5)*drh,2)*drh*density*n);
    fprintf(fp,"%e %e\n",(ih+0.5)*drh,gr);
  }
  fclose(fp);
  free(nhis);
}

int main() {
  int i;
  float cpu1,cpu2;

  /* Read the atomic position data */
  fp = fopen("pos.d","r");
  fscanf(fp,"%f %f %f",&(al[0]),&(al[1]),&(al[2]));
  fscanf(fp,"%d",&n);
  r = (float*)malloc(sizeof(float)*3*n);
  for (i=0; i<n; i++)
    fscanf(fp,"%f %f %f",&(r[3*i]),&(r[3*i+1]),&(r[3*i+2]));
  fclose(fp);
  /* Compute the histogram */
  cpu1 = ((float) clock())/CLOCKS_PER_SEC;
  histogram();
  cpu2 = ((float) clock())/CLOCKS_PER_SEC;
  printf("Execution time (s) = %le\n",cpu2-cpu1);
```

```
  free(r);
  return 0;
}
```

2.



Part II

1. Sumbmit code

```
// Hybrid MPI+CUDA computation of Pi
#include <stdio.h>
#include <omp.h>//jd
#include <mpi.h>
#include <cuda.h>

#define NBIN  10000000  // Number of bins
#define NUM_DEVICE 2 // jd
#define NUM_BLOCK   13  // Number of thread blocks
```

```
#define NUM_THREAD 192  // Number of threads per block

// Kernel that executes on the CUDA device
__global__ void cal_pi(float *sum,int nbin,float step,float offset,int
nthreads,int nblocks) {
    int i;
    float x;
    int idx = blockIdx.x*blockDim.x+threadIdx.x;  // Sequential thread index
across the blocks
    for (i=idx; i<nbin; i+=nthreads*nblocks) {  // Interleaved bin assignment
to threads
        x = offset+(i+0.5)*step;
        sum[idx] += 4.0/(1.0+x*x);
    }
}

int main(int argc,char **argv) {
    int myid,nproc,nbin,tid, mpid;
    float step,offset,pi=0.0,pig;
    dim3 dimGrid(NUM_BLOCK,1,1);  // Grid dimensions (only use 1D)
    dim3 dimBlock(NUM_THREAD,1,1);  // Block dimensions (only use 1D)
    float *sumHost,*sumDev;  // Pointers to host & device arrays
    int dev_used;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);  // My MPI rank
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);  // Number of MPI processes
    //nbin = NBIN/nproc;  // Number of bins per MPI process
    //step = 1.0/(float)(nbin*nproc);  // Step size with redefined number of
bins
    //offset = myid*step*nbin;  // Quadrature-point offset

    omp_set_num_threads(NUM_DEVICE);        // One OpenMP thread per GPU device
    nbin = NBIN/(nproc*NUM_DEVICE);         // # of bins per OpenMP thread
    step = 1.0/(float)(nbin*nproc*NUM_DEVICE);
    #pragma omp parallel private(  mpid, offset, sumHost, sumDev, tid,
dev_used  ) reduction(+:pi)
    {
        mpid = omp_get_thread_num();
        offset = (NUM_DEVICE*myid+mpid)*step*nbin;  // Quadrature-point offset
        cudaSetDevice(mpid%2);
        //cudaSetDevice(myid%2);
        size_t size = NUM_BLOCK*NUM_THREAD*sizeof(float);  //Array memory size
        sumHost = (float *)malloc(size);  //  Allocate array on host
        cudaMalloc((void **) &sumDev,size);  // Allocate array on device
        cudaMemset(sumDev,0,size);  // Reset array in device to 0
```

```
        // Calculate on device (call CUDA kernel)
        cal_pi <<<dimGrid,dimBlock>>>
(sumDev,nbin,step,offset,NUM_THREAD,NUM_BLOCK);
        // Retrieve result from device and store it in host array
        cudaMemcpy(sumHost,sumDev,size,cudaMemcpyDeviceToHost);
        // Reduction over CUDA threads
        for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++)
            pi += sumHost[tid];
        pi *= step;// race condition solved by reduction(+:pi)

        // CUDA cleanup
        free(sumHost);
        cudaFree(sumDev);
        cudaGetDevice(&dev_used);
        //printf("myid = %d: device used = %d; partial pi =
%f\n",myid,dev_used,pi);
        printf("myid = %d; mpid = %d: device used = %d; partial pi = %f\n",
myid, mpid, dev_used, pi);
    } // ENd omp parallel jd

    // Reduction over MPI processes
    MPI_Allreduce(&pi,&pig,1,MPI_FLOAT,MPI_SUM,MPI_COMM_WORLD);
    if (myid==0) printf("PI = %f\n",pig);

    MPI_Finalize();
    return 0;
}
```

2.

```
myid = 1; mpid = 1: device used = 1; partial pi = 0.567582
myid = 1; mpid = 0: device used = 0; partial pi = 0.719409
myid = 0; mpid = 1: device used = 1; partial pi = 0.874671
myid = 0; mpid = 0: device used = 0; partial pi = 0.979926
PI = 3.141588
```