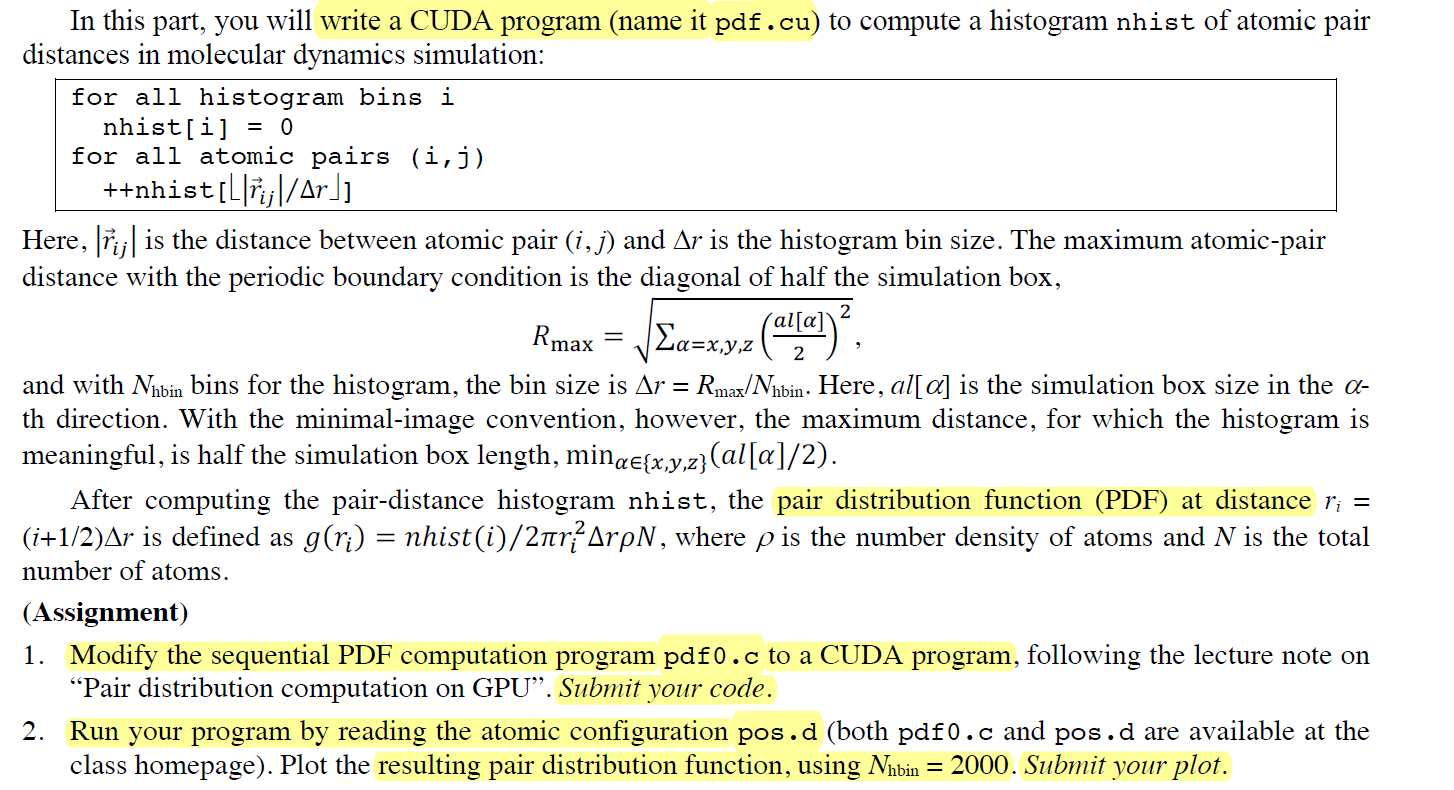
Minh Tran

Assignment 6 CSCI 596: Hybrid MPI + OpenMP + CUDA Programming

1. Pair-Distribution Computation with CUDA



**Solution:**

The new program pfd.cu is modifed from pfd0.c. The changes are marked by //###

/\*----------------------------------------------------------------------

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;

// start ###

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

// DEVICE KERNEL

\_\_device\_\_ float d\_SignR(float v,float x) {if (x > 0) return v; else return -v;}

// read-only constant memory: faster access

// D means device

\_\_constant\_\_ float DALTH[3]; // Simulation box lengths

\_\_constant\_\_ int DN; // number of atoms

\_\_constant\_\_ float DDRH; // bin size

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

\_\_global\_\_ void gpu\_histogram\_kernel(float \*r,float \*nhis)

{

int i,j,a,ih;

float rij,dr;

// offset indexes to perform block spatial decomposition

int iBlockBegin = (DN/gridDim.x)\*blockIdx.x;

int iBlockEnd = min((DN/gridDim.x)\*(blockIdx.x+1),DN);// handle end blocks

int jBlockBegin = (DN/gridDim.y)\*blockIdx.y;

int jBlockEnd = min((DN/gridDim.y)\*(blockIdx.y+1),DN);

// perform interleaving threads: skipping thread by block dimension

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); // avoiding race condition due to reading, writting from memory

} // end if i<j

} // end for j

} // end for i

}

//

// end ###

/\*--------------------------------------------------------------------\*/

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);

// start ###

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

// memory allocation for r and nhis

cudaMalloc((void\*\*)&dev\_r,sizeof(float)\*3\*n);

cudaMalloc((void\*\*)&dev\_nhis,sizeof(float)\*NHBIN);

// memory copy: copy r from host to deviece

cudaMemcpy(dev\_r,r,3\*n\*sizeof(float),cudaMemcpyHostToDevice);

cudaMemset(dev\_nhis,0.0,NHBIN\*sizeof(float)); // reset nhis to device

// toSymbol: copy to read-only constants; 0 for memory offset

cudaMemcpyToSymbol(DALTH,alth,sizeof(float)\*3,0,cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(DN,&n,sizeof(int),0,cudaMemcpyHostToDevice);

cudaMemcpyToSymbol(DDRH,&drh,sizeof(float),0,cudaMemcpyHostToDevice);

// user-defined topology

dim3 numBlocks(8,8,1);

dim3 threads\_per\_block(16,16,1);

// Compute dev\_nhis on GPU: dev\_r[] ® dev\_nhis[]

gpu\_histogram\_kernel<<<numBlocks,threads\_per\_block>>>(dev\_r,dev\_nhis);

// copy back results to host

cudaMemcpy(nhis,dev\_nhis,NHBIN\*sizeof(float),cudaMemcpyDeviceToHost);

// free memory

cudaFree(dev\_r);

cudaFree(dev\_nhis);

// for (i=0; i<n-1; i++) {

// for (j=i+1; j<n; j++) { // loop through pair of atoms

// rij = 0.0;

// for (a=0; a<3; a++) { // 3 directions

// dr = r[3\*i+a]-r[3\*j+a]; // relative position vector

// /\* Periodic boundary condition \*/

// dr = dr-SignR(alth[a],dr-alth[a])-SignR(alth[a],dr+alth[a]);

// rij += dr\*dr;

// }

// rij = sqrt(rij); /\* Pair distance \*/

// ih = rij/drh; // divided by bin size

// nhis[ih] += 1.0; /\* Entry to the histogram, potential race condition \*/

// } // End for j

// } // Endo for i

// end ###

density = n/(al[0]\*al[1]\*al[2]);

/\* Print out the histogram \*/

fp = fopen("pdf.d","w");

for (ih=0; ih<NHBIN; ih++) {

// gr function: histogram divided by normalization factor

gr = nhis[ih]/(2\*M\_PI\*pow((ih+0.5)\*drh,2)\*drh\*density\*n);

// 1st column: bin position

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);

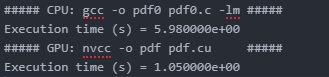
return 0;

}

To complete this part, run the following on cluster:

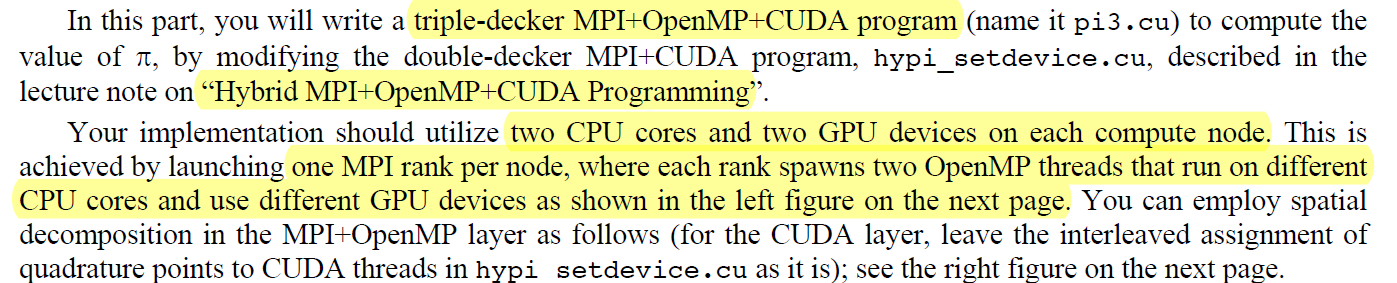
* Compile the programs by: (The input file pos.d is read)
  + **Gcc -o pdf0 pdf0.c -lm**
  + **Nvcc -o pdf pdf.cu -lm**
* Run sbatch:
  + **Sbatch pdf.sl**

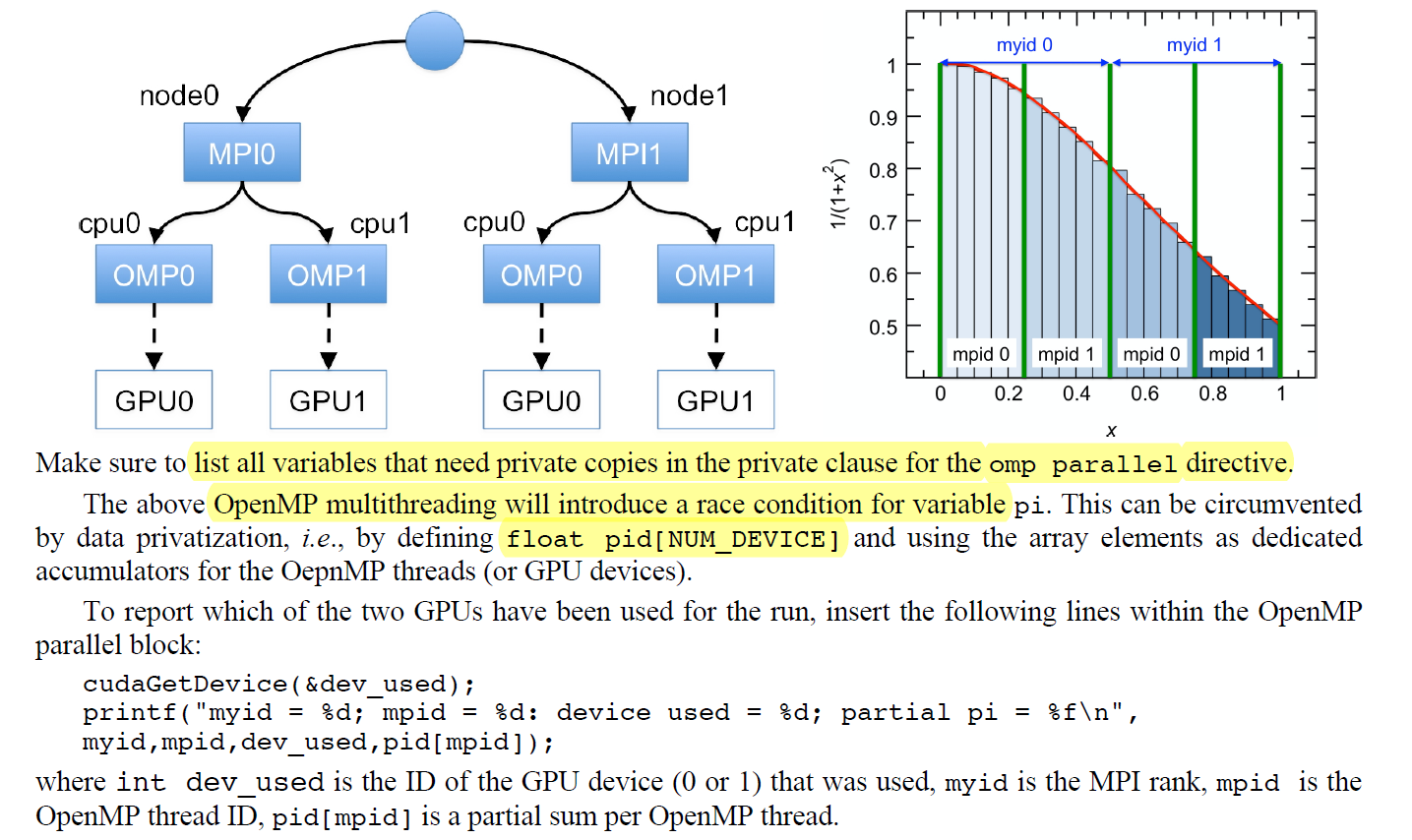
The output are the histograms (pdf.d and pdf0.d) and the output file pdf.out



|  |  |
| --- | --- |
|  |  |

1. MPI+OpenMP+CUDA Computation of Pi





**Solution:**

The pi3.cu is below:

// Hybrid MPI+CUDA computation of Pi

#include <stdio.h>

#include <mpi.h>

#include <cuda.h>

#include <omp.h>//###

#define NBIN 10000000 // Number of bins

#define NUM\_BLOCK 13 // Number of thread blocks

#define NUM\_THREAD 192 // Number of threads per block

#define NUM\_DEVICE 2 // ### Number of GPU devices

// 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;

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

/// start ###

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);

// reduction to avoid rac

#pragma omp parallel private(offset, sumHost, sumDev, tid, dev\_used) reduction(+:pi)

{

int 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

// end ###

// 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;

}

To compile in this triple decker environment:

nvcc -Xcompiler -fopenmp [pi3.cu](http://pi3.cu/) -o pi3 -I${OPENMPI\_ROOT}/include -L${OPENMPI\_ROOT}/lib -lmpi -lgomp

The output is:

