Minh Tran

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

I. Pair-Distribution Computation with CUDA

In this part, you will write a CUDA program (name it `pdf.cu`) to compute a histogram `nhist` of atomic pair distances in molecular dynamics simulation:

```
for all histogram bins i
  nhist[i] = 0
for all atomic pairs (i,j)
  ++nhist[$\lfloor|\vec{r}_{ij}|/\Delta r\rfloor$]
```

Here, $|\vec{r}_{ij}|$ is the distance between atomic pair (i, j) and Δr is the histogram bin size. The maximum atomic-pair distance with the periodic boundary condition is the diagonal of half the simulation box,

$$R_{\max} = \sqrt{\sum_{\alpha=x,y,z}\left(\frac{al[\alpha]}{2}\right)^2},$$

and with N_{hbin} bins for the histogram, the bin size is $\Delta r = R_{\max}/N_{\text{hbin}}$. Here, $al[\alpha]$ is the simulation box size in the α-th direction. With the minimal-image convention, however, the maximum distance, for which the histogram is meaningful, is half the simulation box length, $\min_{\alpha\in\{x,y,z\}}(al[\alpha]/2)$.

After computing the pair-distance histogram `nhist`, the pair distribution function (PDF) at distance $r_i = (i+1/2)\Delta r$ is defined as $g(r_i) = nhist(i)/2\pi r_i^2 \Delta r \rho N$, where ρ is the number density of atoms and N is the total number of atoms.

(Assignment)

1. Modify the sequential PDF computation program `pdf0.c` to a CUDA program, following the lecture note on "Pair distribution computation on GPU". *Submit your code*.
2. Run your program by reading the atomic configuration `pos.d` (both `pdf0.c` and `pos.d` are available at the class homepage). Plot the resulting pair distribution function, using $N_{\text{hbin}} = 2000$. *Submit your plot*.

Solution:

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

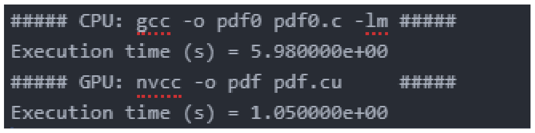
```
/*----------------------------------------------------------------------
Program pdf0.c computes a pair distribution function for n atoms
given the 3D coordinates of the atoms.
----------------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include 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

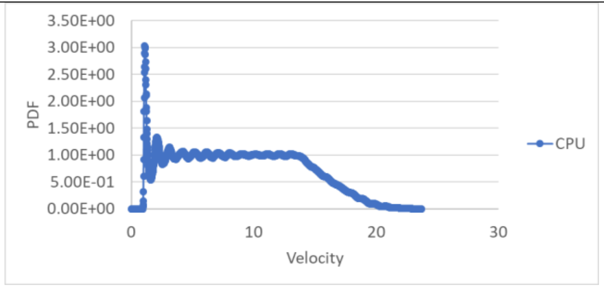
To complete this part, run the following on cluster:

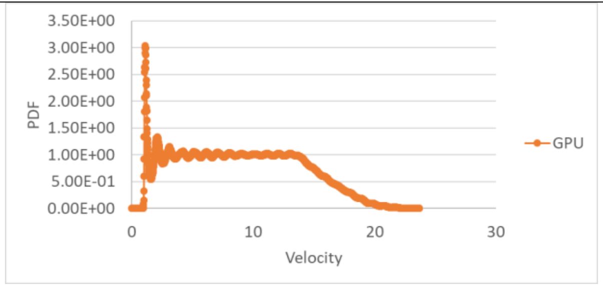
- Compile the programs by: (The input file pos.d is read)
  - **Gcc -o pdf0 pdf0.c -lm**
  - **Nvcc -o pdf pdf.cu -lm**
- Run sbatch:
  - **Sbatch pdf.sl**

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



```
##### CPU: gcc -o pdf0 pdf0.c -lm #####
Execution time (s) = 5.980000e+00
##### GPU: nvcc -o pdf pdf.cu     #####
Execution time (s) = 1.050000e+00
```

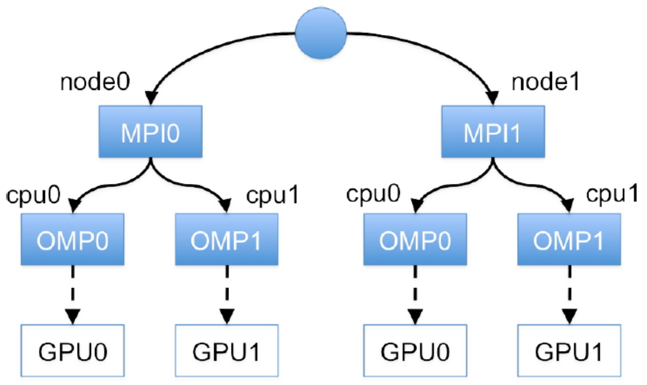

3.50E+00
3.00E+00
2.50E+00
2.00E+00
1.50E+00
1.00E+00
5.00E-01
0.00E+00
PDF
0
10
20
30
Velocity
CPU

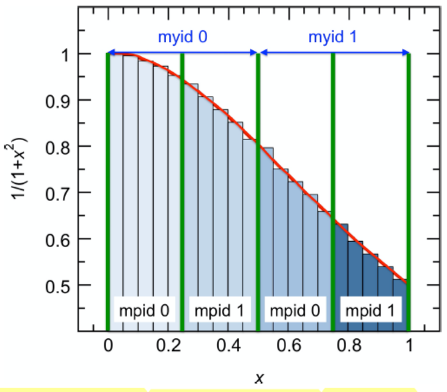

3.50E+00
3.00E+00
2.50E+00
2.00E+00
1.50E+00
1.00E+00
5.00E-01
0.00E+00
PDF
0
10
20
30
Velocity
GPU

II. MPI+OpenMP+CUDA Computation of Pi

In this part, you will write a triple-decker MPI+OpenMP+CUDA program (name it `pi3.cu`) to compute the value of π, by modifying the double-decker MPI+CUDA program, `hypi_setdevice.cu`, described in the lecture note on "Hybrid MPI+OpenMP+CUDA Programming".

Your implementation should utilize two CPU cores and two GPU devices on each compute node. This is achieved by launching one MPI rank per node, where each rank spawns two OpenMP threads that run on different CPU cores and use different GPU devices as shown in the left figure on the next page. You can employ spatial decomposition in the MPI+OpenMP layer as follows (for the CUDA layer, leave the interleaved assignment of quadrature points to CUDA threads in `hypi setdevice.cu` as it is); see the right figure on the next page.





Make sure to list all variables that need private copies in the private clause for the `omp parallel` directive.

The above OpenMP multithreading will introduce a race condition for variable `pi`. This can be circumvented by data privatization, *i.e.*, by defining `float pid[NUM_DEVICE]` and using the array elements as dedicated accumulators for the OepnMP threads (or GPU devices).

To report which of the two GPUs have been used for the run, insert the following lines within the OpenMP parallel block:

```
cudaGetDevice(&dev_used);
printf("myid = %d; mpid = %d: device used = %d; partial pi = %f\n",
myid,mpid,dev_used,pid[mpid]);
```

where `int dev_used` is the ID of the GPU device (0 or 1) that was used, `myid` is the MPI rank, `mpid` is the OpenMP thread ID, `pid[mpid]` is a partial sum per OpenMP thread.

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
```

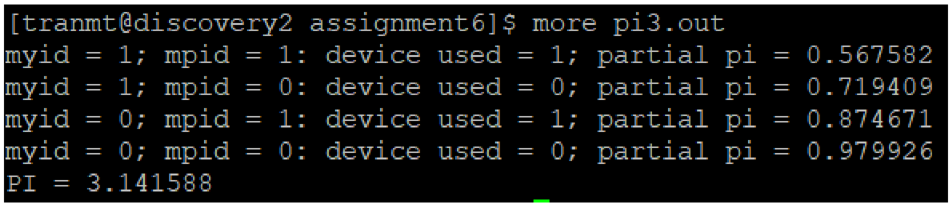
```
#define NUM_THREAD 192  // Number of threads per block
#define NUM_DEVICE 2 // ### Number of GPU devices

// 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 -o pi3 -I${OPENMPI_ROOT}/include -

L${OPENMPI_ROOT}/lib -lmpi -lgomp
```

The output is:



```
[tranmt@discovery2 assignment6]$ more pi3.out
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
```