Assignment 6- Hybrid MPI + MP + CUDA Programming
CSCI 596: Scientific Computing & Visualization

Anup V Kanale

November 1, 2017

The goal of this assignment is understand the basics of cuda through two examples– for calculating the pair distribution function, and to calculate π.

1 Task I– Pair-Distribution Computation with CUDA

The pair distribution function (PDF) is the probability of finding any two atoms, which are a given distance r apart, in a given volume. It is popularly used to measure the order, especially crystalline materials.

To compute the pdf, we fist calculate the histogram of atomic pair distances, i.e., calculate the distance between all pairs of atoms i and j, say r_{ij}, put them in the appropriate '*bins*' Δr to generate a histogram. Note that the maximum distance between pairs under periodic boundary condition is the diagonal of half the simulation box. The algorithm is as follows:

```
for all histogram bins i
    nhist[i] = 0
for all atomic pairs (i,j)
    ++nhist[$r_{ij}/r$]
```

Once this histogram is obtained, the pdf may then be computed using the following formula:

$$g(r_k) = \frac{\text{nhist}(k)}{2\pi r_k^2 \Delta r \rho N} \tag{1}$$

where r_k is the distance between two atoms, ρ is the number density of atoms, N is the total number of atoms. Notice that the obtaining the histogram involves computing distance between all pairs of atoms, which can get expensive with a large number of atoms. The parallel processing power of the GPUs was used to carry out these computations.

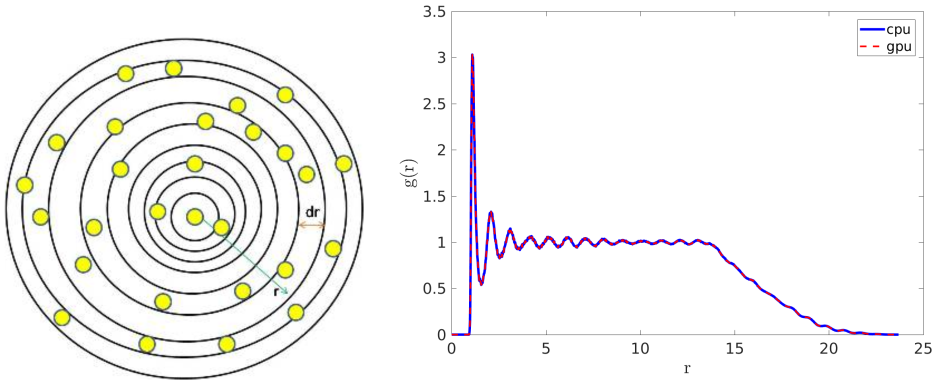


Figure 1: (a)Schematic for obtaining the histogram (b)PDF computation– CPU vs GPU

The code is attached in Appendix A. The pair distribution function computed using both the CPU and the GPUs shown for comparison in Fig 1 (the trailing off after $r = 14$ is explained in the notes.). The cpu implentation ran in 2.62 seconds whereas the gpu-parallel implementation ran in 0.28 seconds (almost 10 times faster!).

2 Task II– Parallel Computation of π using MPI+OMP+CUDA

In this part, a triple-decker MPI+OpenMP+CUDA program was written to compute the value of π, by modifying the double-decker MPI+CUDA program provided in class, described in the lecture note on “Hybrid MPI+OpenMP+CUDA Programming". The formula used is

$$\pi = \int_0^1 \frac{4}{1+x^2}dx \approx \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2} \tag{2}$$

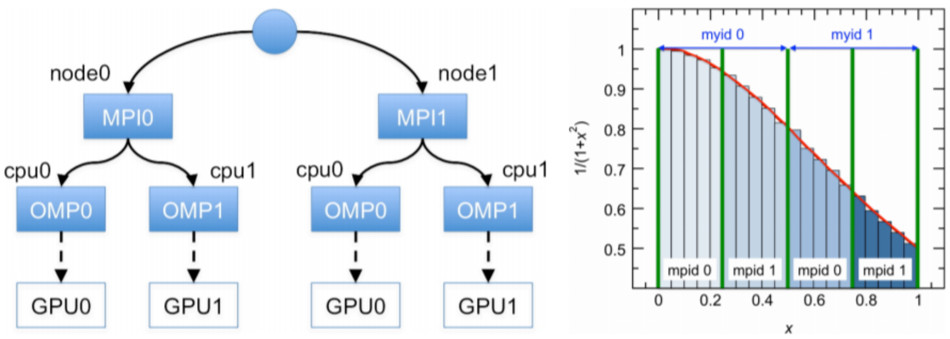
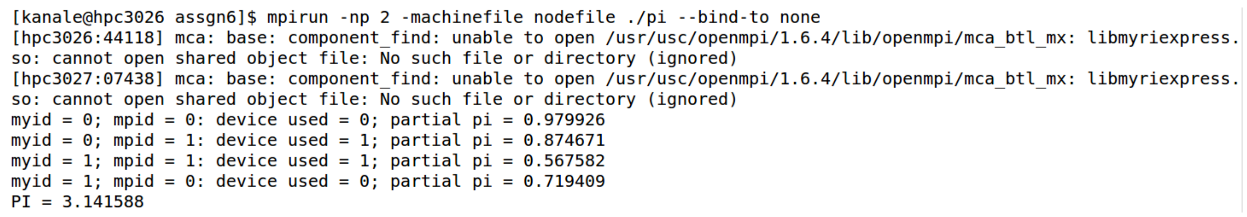


Figure 2: Output of program for computation of π

This program runs on two CPU cores, and 2 GPU devices on each compute node. This is achieved by launching one MPI rank per node, where each rank spawns two OpenMP threads, and each thread uses a different GPU device. This is illustrated in Fig 2. The above OpenMP multithreading introduces a race condition for variable pi. This can be circumvented by data privatization using `reduction(+:pi)`, which creates a `pi` variable for each thread. The ‘+:’ means that the ‘addition reduction’ is to be performed on this variable.

The program can be found in appendix B, and the output of the program is as shown below in Fig 2. `myid` refers to the process rank, `mpid` is the thread number within the processor, and `device used` refers to the gpu.



```
[kanale@hpc3026 assgn6]$ mpirun -np 2 -machinefile nodefile ./pi --bind-to none
[hpc3026:44118] mca: base: component_find: unable to open /usr/usc/openmpi/1.6.4/lib/openmpi/mca_btl_mx: libmyriexpress.
so: cannot open shared object file: No such file or directory (ignored)
[hpc3027:07438] mca: base: component_find: unable to open /usr/usc/openmpi/1.6.4/lib/openmpi/mca_btl_mx: libmyriexpress.
so: cannot open shared object file: No such file or directory (ignored)
myid = 0; mpid = 0: device used = 0; partial pi = 0.979926
myid = 0; mpid = 1: device used = 1; partial pi = 0.874671
myid = 1; mpid = 1: device used = 1; partial pi = 0.567582
myid = 1; mpid = 0: device used = 0; partial pi = 0.719409
PI = 3.141588
```

Figure 3: Output of program for computation of π

Important points

1. Finding compatible version of mpi, openMP and cuda is not trivial. And it varies from system to system.
2. pay attention to how the sequential thread index is calculated across blocks.

A Appendix– Calculating the PDF

Note that there were no major changes in the `main()` function, so it is not shown.

```
/*------------------------------------------------------------------------
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;

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

__device__ float d_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;

  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;
        // nhis[ih] += 1.0; /* Entry to the histogram */
        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[] Âő dev_nhis[]
  cudaMemcpy(nhis,dev_nhis,NHBIN*sizeof(float),cudaMemcpyDeviceToHost);

  density = n/(al[0]*al[1]*al[2]);
  /* Print out the histogram */
  fp = fopen("pdf_gpu.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);
}
```

pdf1writeup.cu

B Appendix– Computation of π

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

#define NUM_DEVICE 2 // # of GPU devices = # of OpenMP threads
#define NBIN  10000000  // Number of bins
#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,dev_used,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

  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
  omp_set_num_threads(NUM_DEVICE);
  nbin = NBIN/(nproc*NUM_DEVICE);  // Number of bins per MPI process
  step = 1.0/(float)(nbin*nproc*NUM_DEVICE);  // Step size with redefined number of bins

  #pragma omp parallel private(mpid,offset,sumHost,tid) reduction(+:pi)
  {
      mpid = omp_get_thread_num();
      offset = (NUM_DEVICE*myid+mpid)*step*nbin;  // Quadrature-point offset

      cudaSetDevice(mpid%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;
      // CUDA cleanup
      free(sumHost);
      cudaFree(sumDev);

      // Reduction over MPI processes
      cudaGetDevice(&dev_used);
      printf("myid = %d; mpid = %d: device used = %d; partial pi = %f\n", myid,mpid,dev_used,pi);
  } // End omp parallel
  MPI_Allreduce(&pi,&pig,1,MPI_FLOAT,MPI_SUM,MPI_COMM_WORLD);
  if (myid==0) printf("PI = %f\n",pig);
  MPI_Finalize();
  return 0;
}
```

piCudaMpiOmpwriteup.cu