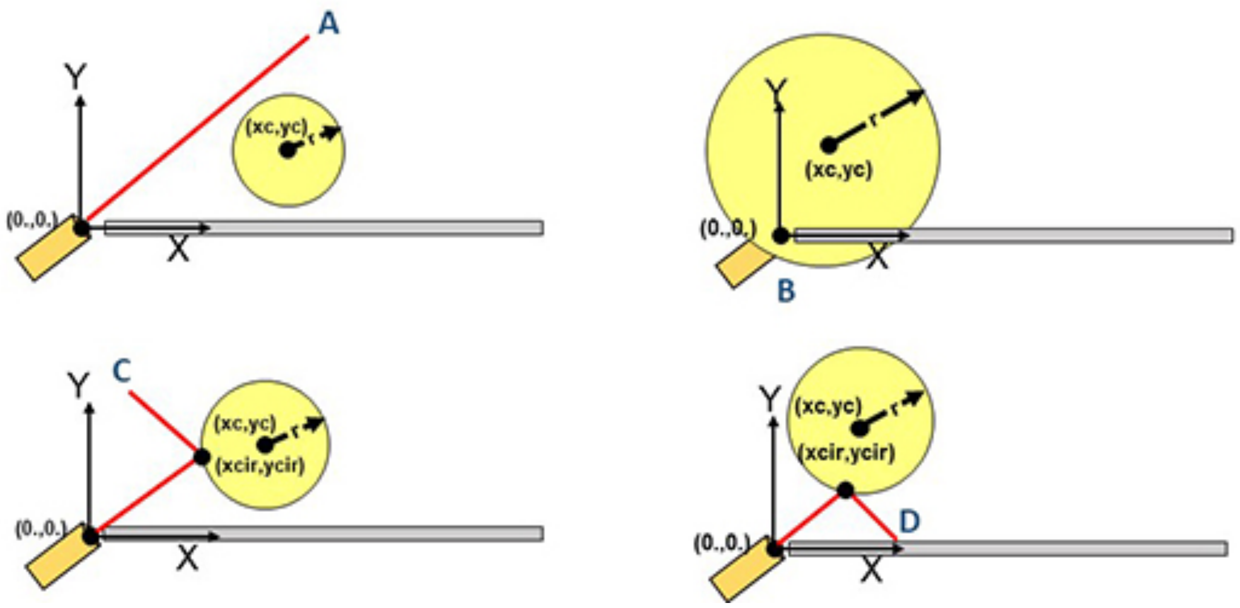
CS 475/575 -- Spring Quarter 2020

Project #5

CUDA Monte Carlo

100 Points

Due: May 20



Liang Zhao

933-667-879

zhaolia@oregonstate.edu

This project is exciting, and the calculation speed of OSU's DGX system is too fast. I checked on Google, and the price of each DGX is $ 270000. I don't know if this is true. This project is very similar to the first project, only converted to GPU for calculation. Judging from the results, the GPU is much stronger than the CPU in some aspects.

Monte Carlo simulation is used to determine the range of outcomes for a series of parameters, each of which has a probability distribution showing how likely each option is to happen. In this project, you will take a scenario and develop a Monte Carlo simulation of it, determining how likely a particular output is to happen.

1. Tell what machine you ran this on

   My program runs on the OSU college of Engineering DGX system.

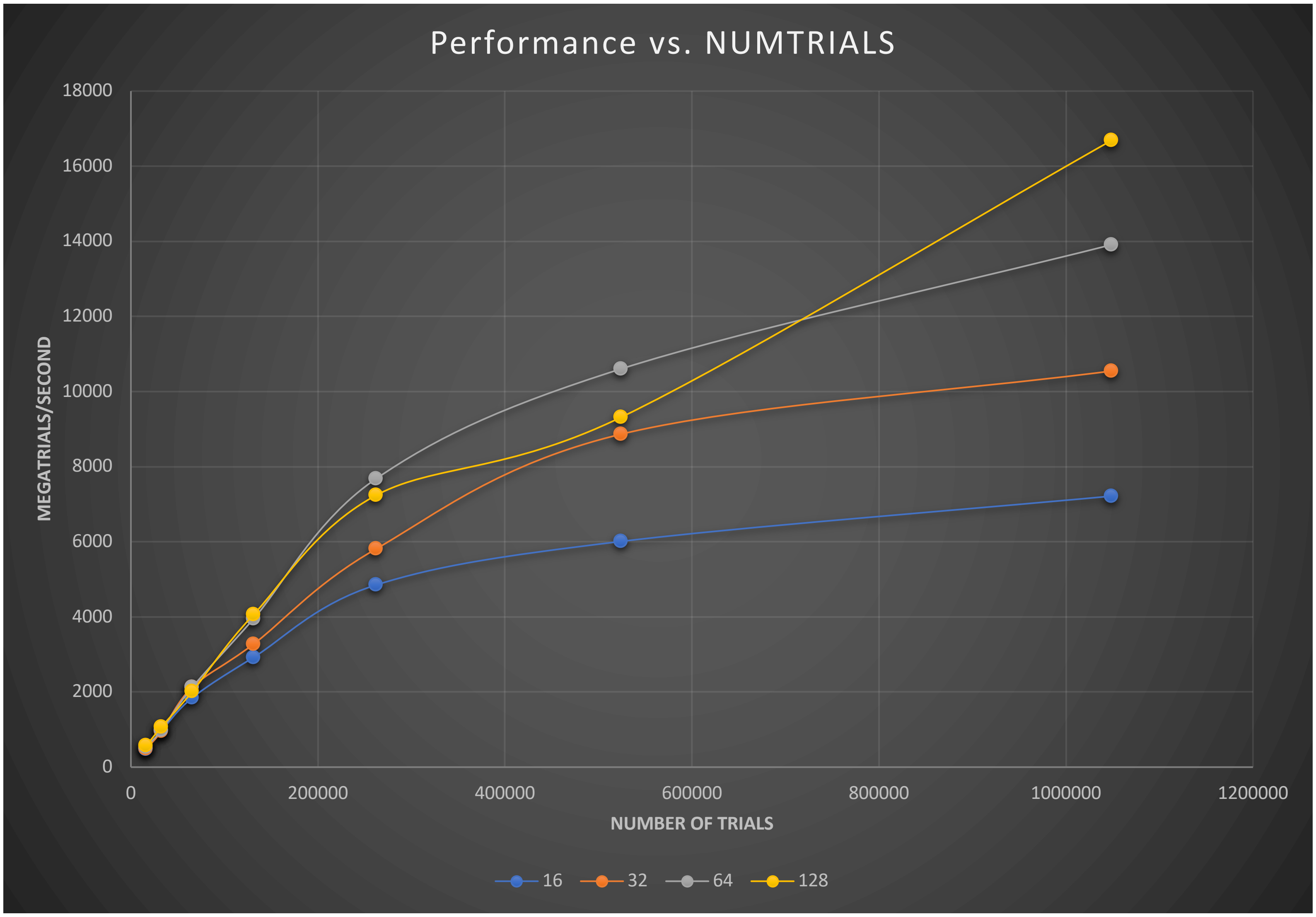
2. Show the table and the two graphs

   Monte Carlo performance table:

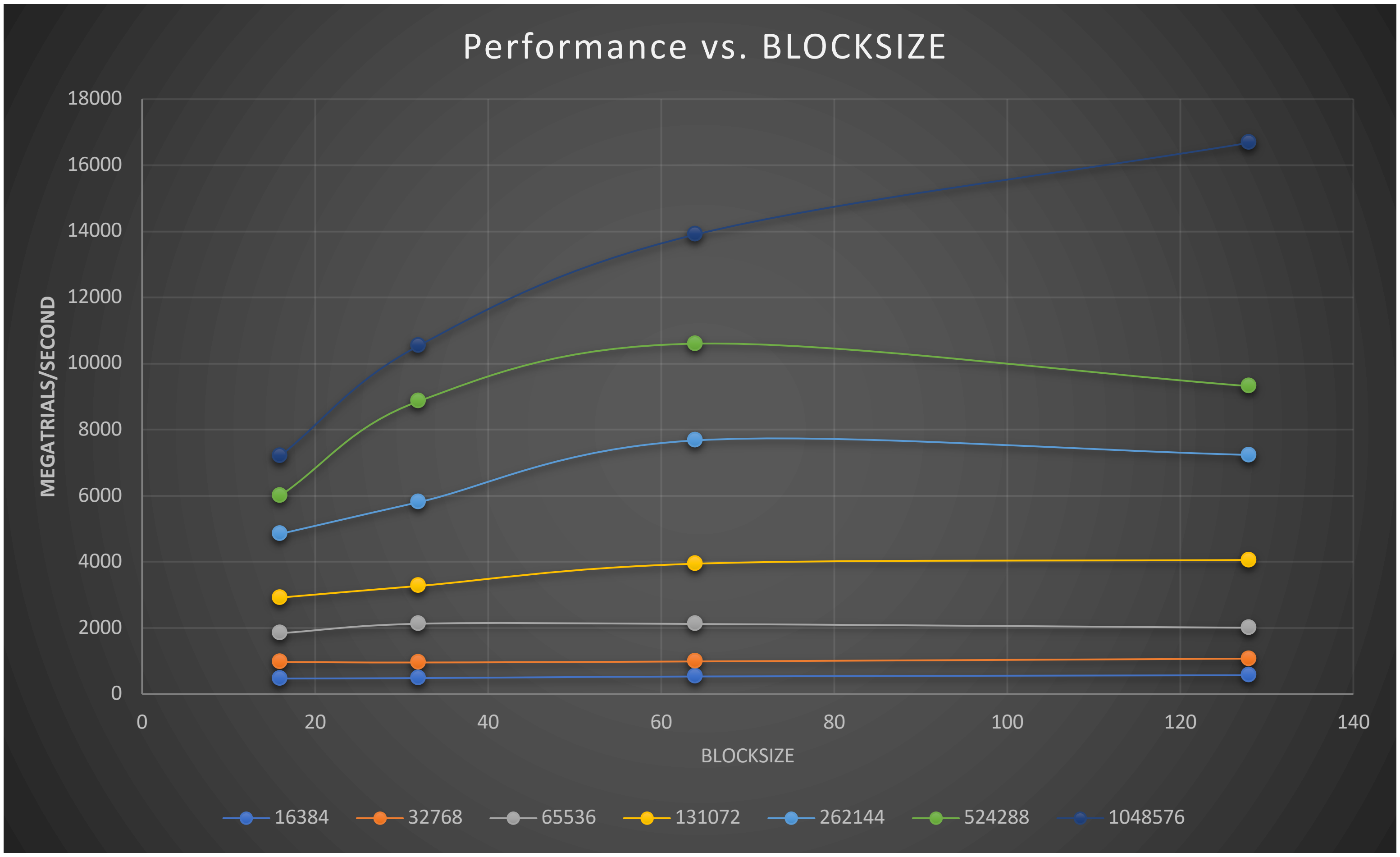
Number of Trials	BLOCKSIZE	MegaTrials/Second	Probability
16384	16	470.7241	41.75%
16384	32	484.8485	41.74%
16384	64	533.3333	42.05%
16384	128	571.4286	42.66%
32768	16	970.6161	42.31%
32768	32	957.0093	41.39%
32768	64	988.417	42.01%

32768	128	1073.3753	42.50%
65536	16	1843.3844	42.09%
65536	32	2128.8981	41.86%
65536	64	2122.2798	42.02%
65536	128	2007.8432	41.99%
131072	16	2921.5407	42.07%
131072	32	2274.2921	42.01%
131072	64	3946.0501	41.92%
131072	128	4059.4648	41.95%
262144	16	4850.2071	42.20%
262144	32	5809.929	42.00%
262144	64	7677.6009	41.84%
262144	128	7236.7491	42.16%
524288	16	6010.2714	42.06%
524288	32	8865.8009	41.90%
524288	64	10604.5307	42.00%
524288	128	8316.7508	42.01%
1048576	16	7216.0319	42.11%
1048576	32	10546.5082	42.01%
1048576	64	13908.3191	42.00%
1048576	128	16684.3173	41.94%

Graph of performance vs. NUMTRIALS with multiple curves of BLOCKSIZE:



Graph of performance vs. BLOCKSIZE with multiple curves of NUMTRIALS:



3. What patterns are you seeing in the performance curves?

We can see from the figure above that as block size and trials increase, the performance of the calculation will be improved. However, it seems that as block size increases, performance increases faster. With the infinite increase in block size and trials, performance will also tend to be stable.

4. Why do you think the patterns look this way?

   A block is made up of a grid of threads. Of course, more block size means more threads so that performance will increase exponentially. With the increase of trials, the time used to call the thread will be relatively reduced, so performance will improve.

5. Why is a BLOCKSIZE of 16 so much worse than the others?

   16 block size will have fewer threads than others. The fewer the number of threads, the slower the performance.

6. How do these performance results compare with what you got in Project #1? Why?

   It goes without saying that the first assignment is much slower than this assignment. Because open MP mainly uses CPU to calculate, while CUDA uses GPU to calculate. We all know that GPUs have larger TDP and higher frequency memory than CPUs. At the design level of the chip, the GPU has more ALUs because it does not require many logical operation units like the CPU.

7. What does this mean for the proper use of GPU parallel computing?

Although the GPU showed super performance in this job, let's not ignore the role of the CPU. GPU is very good at data-parallel computing, CPU is very good at parallel processing. The GPU has thousands of cores, and the CPU has less than 100 cores. The GPU has about 40 hyper-threads per core, and the CPU has about two hyper-threads per core. GPU is challenging to execute recursive code, and CPU has fewer problems. CUDA allows engineers to ignore the underlying graphical concepts and move to more general high-performance computing. In the future, the combination of GPU and CPU seems to be a trend.

```
// System includes
#include <stdio.h>
#include <assert.h>
#include <malloc.h>
#include <math.h>
#include <stdlib.h>

// CUDA runtime
#include <cuda_runtime.h>

// Helper functions and utilities to work with CUDA
#include "helper_functions.h"
#include "helper_cuda.h"


// setting the number of trials in the monte carlo simulation:
#ifndef NUMTRIALS
#define NUMTRIALS   ( 1024*1024 )
#endif


#ifndef BLOCKSIZE
#define BLOCKSIZE       32     // number of threads per block
#endif

#define NUMBLOCKS       ( NUMTRIALS / BLOCKSIZE )


// ranges for the random numbers:
const float XCMIN =  0.0;
const float XCMAX =  2.0;
const float YCMIN =  0.0;
const float YCMAX =  2.0;
const float RMIN  =  0.5;
const float RMAX  =  2.0;

// function prototypes:
float       Ranf( float, float );
int     Ranf( int, int );
void        TimeOfDaySeed( );



__global__  void MonteCarlo( float *Xcs, float *Ycs, float *Rs, int *Hits )
{
    unsigned int wgNumber      = blockIdx.x;
unsigned int wgDimension   = blockDim.x;
unsigned int threadNum     = threadIdx.x;
unsigned int gid           = wgNumber*wgDimension + threadNum;

// all the monte carlo stuff goes in here
// if we make it all the way through, then Hits[gid] = 1

// randomize the location and radius of the circle:
float xc = Xcs[gid];
float yc = Ycs[gid];
float  r =  Rs[gid];

float tn = tanf( (float)( (M_PI/180.) * 30. ) );
Hits[gid] = 0;

// solve for the intersection using the quadratic formula:

float a = 1. + tn*tn;
float b = -2.*( xc + yc*tn );
float c = xc*xc + yc*yc - r*r;
float d = b*b - 4.*a*c;

// cascading if-statements:
//  if you used "continue;" in project #1, change to this style because,
//  if there is no for-loop, then there is nowhere to continue to

if( d >= 0 )
{
    d = sqrt( d );
    float t1 = (-b + d ) / ( 2.*a );
    float t2 = (-b - d ) / ( 2.*a );
    float tmin = t1 < t2 ? t1 : t2;


    if( tmin >= 0 )
    {

        float xcir = tmin;
        float ycir = tmin*tn;


        float nx = xcir - xc;
        float ny = ycir - yc;
        float nxy = sqrt( nx*nx + ny*ny );
        nx /= nxy;
```

```
            ny /= nxy;


            float inx = xcir - 0.;
            float iny = ycir - 0.;
            float in = sqrt( inx*inx + iny*iny );
            inx /= in;
            iny /= in;


            float dot = inx*nx + iny*ny;
            float outy = iny - 2.*ny*dot;

            // find out if it hits the infinite plate:
            float t = ( 0. - ycir ) / outy;
            if( t >= 0. )
            {
                Hits[gid] = 1;
            }
        }
    }
}


// main program:

int
main( int argc, char* argv[ ] )
{
    TimeOfDaySeed( );

    int dev = findCudaDevice(argc, (const char **)argv);

    // allocate host memory:

    float *hXcs  = new float[NUMTRIALS];
    float *hYcs  = new float[NUMTRIALS];
    float * hRs  = new float[NUMTRIALS];
    int   *hHits = new   int[NUMTRIALS];

    // fill the random-value arrays:
    for( int n = 0; n < NUMTRIALS; n++ )
    {
        hXcs[n] = Ranf( XCMIN, XCMAX );
        hYcs[n] = Ranf( YCMIN, YCMAX );
        hRs[n]  = Ranf(  RMIN,  RMAX );
    }

    // allocate device memory:

    float *dXcs, *dYcs, *dRs;
    int *dHits;

    dim3 dimsXcs(  NUMTRIALS, 1, 1 );
    dim3 dimsYcs(  NUMTRIALS, 1, 1 );
    dim3 dimsRs(   NUMTRIALS, 1, 1 );
    dim3 dimsHits( NUMTRIALS, 1, 1 );


    cudaError_t status;
    status = cudaMalloc( (void **)(&dXcs), NUMTRIALS*sizeof(float) );
    checkCudaErrors( status );

    status = cudaMalloc( (void **)(&dYcs), NUMTRIALS*sizeof(float) );
    checkCudaErrors( status );

    status = cudaMalloc( (void **)(&dRs), NUMTRIALS*sizeof(float) );
    checkCudaErrors( status );

    status = cudaMalloc( (void **)(&dHits), NUMTRIALS *sizeof(int) );
    checkCudaErrors( status );


    // copy host memory to the device:

    status = cudaMemcpy( dXcs, hXcs, NUMTRIALS*sizeof(float), cudaMemcpyHostToDev
ice );
    checkCudaErrors( status );

    status = cudaMemcpy( dYcs, hYcs, NUMTRIALS*sizeof(float), cudaMemcpyHostToDev
ice );
    checkCudaErrors( status );

    status = cudaMemcpy( dRs, hRs, NUMTRIALS*sizeof(float), cudaMemcpyHostToDevic
e );
    checkCudaErrors( status );


    // setup the execution parameters:
```

```
    dim3 threads(BLOCKSIZE, 1, 1 );
    dim3 grid(NUMBLOCKS, 1, 1 );

    // create and start timer

    cudaDeviceSynchronize( );

    // allocate CUDA events that we'll use for timing:

    cudaEvent_t start, stop;
    status = cudaEventCreate( &start );
    checkCudaErrors( status );
    status = cudaEventCreate( &stop );
    checkCudaErrors( status );

    // record the start event:

    status = cudaEventRecord( start, NULL );
    checkCudaErrors( status );

    // execute the kernel:

    MonteCarlo<<< grid, threads >>>( dXcs, dYcs, dRs, dHits );

    // record the stop event:

    status = cudaEventRecord( stop, NULL );
    checkCudaErrors( status );

    // wait for the stop event to complete:

    status = cudaEventSynchronize( stop );
    checkCudaErrors( status );

    float msecTotal = 0.0f;
    status = cudaEventElapsedTime( &msecTotal, start, stop );
    checkCudaErrors( status );

    // compute and print the performance

    double secondsTotal = 0.001 * (double)msecTotal;
    double trialsPerSecond = (float)NUMTRIALS / secondsTotal;
    double megaTrialsPerSecond = trialsPerSecond / 1000000.;
    fprintf( stderr, "%10d\t%10d\t%10.4lf\t", NUMTRIALS, BLOCKSIZE, megaTrialsPer
Second );
```

```
    // copy result from the device to the host:

    status = cudaMemcpy( hHits, dHits, NUMTRIALS *sizeof(int), cudaMemcpyDeviceTo
Host );
    checkCudaErrors( status );
    cudaDeviceSynchronize( );

    // compute the probability:

    int numHits = 0;
    for(int i = 0; i < NUMTRIALS; i++ )
    {
        numHits += hHits[i];
    }

    float probability = 100.f * (float)numHits / (float)NUMTRIALS;
    fprintf(stderr, "%6.3f %%\n", probability );

    // clean up memory:
    delete [ ] hXcs;
    delete [ ] hYcs;
    delete [ ] hRs;
    delete [ ] hHits;

    status = cudaFree( dXcs );
    status = cudaFree( dYcs );
    status = cudaFree( dRs );
    status = cudaFree( dHits );
    checkCudaErrors( status );

    return 0;
}

float
Ranf( float low, float high )
{
    float r = (float) rand();               // 0 - RAND_MAX
    float t = r  /  (float) RAND_MAX;       // 0. - 1.

    return   low  +  t * ( high - low );
}

int
Ranf( int ilow, int ihigh )
```

```
{
    float low = (float)ilow;
    float high = ceil( (float)ihigh );

    return (int) Ranf(low,high);
}

void
TimeOfDaySeed( )
{
    struct tm y2k = { 0 };
    y2k.tm_hour = 0;   y2k.tm_min = 0; y2k.tm_sec = 0;
    y2k.tm_year = 100; y2k.tm_mon = 0; y2k.tm_mday = 1;

    time_t  timer;
    time( &timer );
    double seconds = difftime( timer, mktime(&y2k) );
    unsigned int seed = (unsigned int)( 1000.*seconds );    // milliseconds
    srand( seed );
}
```