CS 475/575 -- Spring Quarter 2021

Project #5

CUDA Monte Carlo

100 Points

Due: May 19

This page was last updated: May 4, 2021

Note: The flip machines do not have GPU cards in them, so CUDA will not run there. If your own system has a GPU, you can use that. You can also use *rabbit* or the DGX machine, but please be good about sharing them.

Introduction

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 the Project #1 scenario and develop a Monte Carlo simulation of it, determining how likely a particular output is to happen.

The Scenario

Use the same scenario from Project #1.

Using Linux for this Project

On both *rabbit* and the DGX system, here is a working Makefile:

```
CUDA_PATH       =       /usr/local/apps/cuda/cuda-10.1
CUDA_BIN_PATH   =       $(CUDA_PATH)/bin
CUDA_NVCC       =       $(CUDA_BIN_PATH)/nvcc

montecarlo:     montecarlo.cu
                $(CUDA_NVCC) -o montecarlo  montecarlo.cu
```

Before you use the DGX, do your development on the *rabbit* system (Slide #3 of the DGX noteset). It is a little friendlier because you don't have to run your program through a batch submission. But, don't take any final performance numbers from *rabbit*, just get your program running there.

But, if you decide to use Visual Studio on your own machine, you must first install the CUDA Toolkit!

If you are trying to run CUDA on your own Visual Studio system, make sure your machine has the CUDA toolkit installed. It is available here: https://developer.nvidia.com/cuda-downloads

Running CUDA in Visual Studio

Get the NewCudaArrayMul2019.zip file. Un-zip the file and double-click on the .sln file. This is the CUDA version of an array multiply program. Modify it for this assignment.

Requirements:

1. The ranges are:

Variable	Meaning	Range
g	Ground distance to the cliff face	20. - 30.
h	Height of the cliff face	10. - 40.
d	Upper deck distance to the castle	10. - 20.
v	Cannonball initial velocity	30. - 50.
θ	Cannon firing angle	70. - 80.

   Note: these are not the same numbers as we used before!

2. Run this for at least four BLOCKSIZEs (i.e., the number of threads per block) of 16, 32, 64, and 128, combined with NUMTRIALS sizes of at least 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, 512K, and 1M.

3. Be sure the NUMTRIALS are in multiples of 1024, that is, for example, for "32K" use 32,768, not 32,000.

4. Record timing for each combination. For performance, use some appropriate units like MegaTrials/Second.

5. For this one, use CUDA timing, not OpenMP timing.

6. Do a table and two graphs:
    1. Performance vs. NUMTRIALS with multiple curves of BLOCKSIZE
    2. Performance vs. BLOCKSIZE with multiple curves of NUMTRIALS

7. Like Project #1 before, fill the arrays ahead of time with random values. Send them to the GPU where they can be used as look-up tables.

8. You will also need these six .h files:
    - helper_functions.h
    - helper_cuda.h
    - helper_image.h
    - helper_string.h
    - helper_timer.h
    - exception.h

9. Your commentary PDF should:
    1. Tell what machine you ran this on
    2. Show the table and the two graphs
    3. What patterns are you seeing in the performance curves?
    4. Why do you think the patterns look this way?
    5. Why is a BLOCKSIZE of 16 so much worse than the others?
    6. How do these performance results compare with what you got in Project #1? Why?
    7. What does this mean for the proper use of GPU parallel computing?

Grading:

Feature	Points
Monte Carlo performance table	20
Graph of performance vs. NUMTRIALS with multiple curves of BLOCKSIZE	25
Graph of performance vs. BLOCKSIZE with multiple curves of NUMTRIALS	25
Commentary	30
Potential Total	**100**

Some Sample Code:

```
// degrees-to-radians -- callable from the device:
__device__
float
Radians( float d )
{
	return (M_PI/180.f) * d;
}

// the kernel:
__global__
void
MonteCarlo( float *dvs, float *dths, float *dgs, float *dhs, float *dds, int *dhits )
{
	unsigned int gid      = blockIdx.x*blockDim.x + threadIdx.x;

	// randomize everything:
	float v   = dvs[gid];
	float thr = Radians( dths[gid] );
	float vx  = v * cos(thr);
	float vy  = v * sin(thr);
	float  g  =  dgs[gid];
	float  h  =  dhs[gid];
	float  d  =  dds[gid];

	int numHits = 0;

	// see if the ball doesn't even reach the cliff:
	float t = -vy / ( 0.5*GRAVITY );
	float x = vx * t;
	if( x > g )
	{
		...
				numHits = 1;
	}

	dhits[gid] = numHits;
}



// these two #defines are just to label things
// other than that, they do nothing:
#define IN
#define OUT

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

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

	// better to define these here so that the rand() calls don't get into the thread timing:
```

```
float *hvs   = new float [NUMTRIALS];
float *hths  = new float [NUMTRIALS];
float *hgs   = new float [NUMTRIALS];
float *hhs   = new float [NUMTRIALS];
float *hds   = new float [NUMTRIALS];
int   *hhits = new int   [NUMTRIALS];

// fill the random-value arrays:

?????


// allocate device memory:
float *dvs, *dths, *dgs, *dhs, *dds;
int   *dhits;

cudaMalloc( &dvs,   NUMTRIALS*sizeof(float) );
cudaMalloc( &dths,  NUMTRIALS*sizeof(float) );
cudaMalloc( &dgs,   NUMTRIALS*sizeof(float) );
cudaMalloc( &dhs,   NUMTRIALS*sizeof(float) );
cudaMalloc( &dds,   NUMTRIALS*sizeof(float) );
cudaMalloc( &dhits, NUMTRIALS*sizeof(int) );
CudaCheckError( );

// copy host memory to the device:
cudaMemcpy( dvs,  hvs,  NUMTRIALS*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( dths, hths, NUMTRIALS*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( dgs,  hgs,  NUMTRIALS*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( dhs,  hhs,  NUMTRIALS*sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( dds,  hds,  NUMTRIALS*sizeof(float), cudaMemcpyHostToDevice );
CudaCheckError( );

// setup the execution parameters:
dim3 grid( NUMBLOCKS, 1, 1 );
dim3 threads( BLOCKSIZE, 1, 1 );

// allocate cuda events that we'll use for timing:
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop  );
CudaCheckError( );

// let the gpu go quiet:
cudaDeviceSynchronize( );

// record the start event:
cudaEventRecord( start, NULL );
CudaCheckError( );

// execute the kernel:
MonteCarlo<<< grid, threads >>>( IN dvs, IN dths, IN dgs, IN dhs, IN dds,   OUT dhits );

// record the stop event:
cudaEventRecord( stop, NULL );
CudaCheckError( );

// wait for the stop event to complete:
cudaDeviceSynchronize( );
cudaEventSynchronize( stop );
CudaCheckError( );

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

// compute and print the performance

?????
```

```
	// copy result from the device to the host:
	cudaMemcpy( hhits, dhits, NUMTRIALS*sizeof(int), cudaMemcpyDeviceToHost );
	CudaCheckError( );

	// add up the hhits[ ] array: :

	?????

	// compute and print the probability:

	?????

	// clean up host memory:
	delete [ ] hvs;
	delete [ ] hths;
	delete [ ] hgs;
	delete [ ] hhs;
	delete [ ] hds;
	delete [ ] hhits;

	// clean up device memory:
	cudaFree( dvs );
	cudaFree( dths );
	cudaFree( dgs );
	cudaFree( dhs );
	cudaFree( dds );
	cudaFree( dhits );
	CudaCheckError( );

	return 0;
}

void
CudaCheckError( )
{
	cudaError_t e = cudaGetLastError( );
	if( e != cudaSuccess )
	{
		fprintf( stderr, "CUDA failure %s:%d: '%s'\n", __FILE__, __LINE__, cudaGetErrorString(e) );
	}
}
```