CS 475/575 -- Spring Quarter 2022

Project #5

CUDA: Monte Carlo Simulation

100 Points

Due: May 18

This page was last updated: June 1, 2022

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 CUDA-based Monte Carlo simulation of it, determining how likely a particular output is to happen. You will then take timing results and compare them with what you got with OpenMP in Project #1.

The Scenario

Use the same scenario from Project #1.

Compiling and Running on *rabbit* and the DGX System

On both *rabbit* and the DGX system, here is how to compile:

```
/usr/local/apps/cuda/cuda-10.1/bin/nvcc   -o proj05  proj05.cu
./proj05
```

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

proj05: proj05.cu
                $(CUDA_NVCC) -o proj05  proj05.cu
```

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

```
#!/bin/bash
for t in 1024 4096 16384 65536 262144 1048576 2097152 4194304
do
        for b in 8 32 128
        do
                /usr/local/apps/cuda/cuda-10.1/bin/nvcc -DNUMTRIALS=$t -DBLOCKSIZE=$b -o proj05  proj05.cu
                ./proj05
        done
done
```

You can (and should!) write scripts to run the benchmark combinations. If you want to pass in benchmark parameters, the **-DNUMTRIALS=$t** notation works fine in nvcc.

Before you use the DGX, do your development on the *rabbit* system. It is a lot friendlier because you don't have to run your program through a batch submission. you can take your final performance numbers on *rabbit*, but you will enjoy the numbers you get on the DGX more!

You can also take your benchmark numbers on your own machine.

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

This requires a special setup so that Visual Studio knows to run nvcc in the right place. See our CUDA noteset for instructions.

Requirements:

1. Use these as the ranges of the input parameters when you choose random parameter values:

Variable	Description	Minimum	Maximum
tx	Truck X starting location in feet	-10.	10.
txv	Truck X velocity in feet/second	15.	35.
ty	Truck Y location in feet	40.	50.
sv	Snowball overall velocity in feet/second	5.	30.
theta	Snowball horizontal launch angle in degrees	10.	70.
halflen	Truck half-length in feet	15.	30.

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

2. Run this for at least three BLOCKSIZEs (i.e., the number of threads per block) of 8, 32, and 128, combined with NUMTRIALS sizes of at least 1024, 4096, 16384, 65536, 262144, 1048576, 2097152, and 4194304. You can use more if you want.

3. Be sure each NUMTRIALS is a multiple of 1024. All of the ones above already are.

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 .h files:
    - helper_functions.h
    - helper_cuda.h
    - helper_image.h

- helper_string.h
- helper_timer.h
- exception.h

Just keep them in your project folder.

9. Your commentary PDF should:
    1. Tell what machine you ran this on
    2. What do you think this new probability is?
    3. Show the table and the two graphs
    4. What patterns are you seeing in the performance curves?
    5. Why do you think the patterns look this way?
    6. Why is a BLOCKSIZE of 8 so much worse than the others?
    7. How do these performance results compare with what you got in Project #1? Why?
    8. What does this mean for the proper use of GPU parallel computing?

Sample Test Code

Here is the complete array-multiplication CUDA program we looked at in class: arrayMul.cu. As a first step, you might try getting on *rabbit* and compiling and running it to be sure you understand the process.

Skeleton Code for Project #5

proj05.cu
This code has both the old table of parameters and the new table listed in it. The line:
```
#define PROJECT1
```
will turn on the old values. Changing it to:
```
//#define PROJECT1
```
will turn on the new values. When you think your code is ready to go, try it with the old values and see if you get a correct probability of ~29%. If you don't, then something is wrong with your code and you might as well fix it before you try the new values.

Grading:

Feature	Points
Correct probability	10
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	20
Potential Total	**100**