Ivan Theodor-Adrian - DSWT II 1A
Savin Vlad-Mihai - DSWT II 1B

PI computation GPU acceleration using Monte Carlo simulation on CUDA

General description

The number π is a mathematical constant. Originally defined as the ratio of a circle's circumference to its diameter, it now has various equivalent definitions and appears in many formulas in all areas of mathematics and physics. It is approximately equal to 3.14159.

Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. One of the basic examples of getting started with the Monte Carlo algorithm is the estimation of π.

The idea is to simulate random (x, y) points in a 2-D plane with domain as a square of side 1 unit. Imagine a circle inside the same domain with same diameter and inscribed into the square. We then calculate the ratio of number points that lied inside the circle and total number of generated points.

We know that area of the square is 1 unit sq while that of circle is:

$$\pi * \left(\frac{1}{2}\right)^2 = \frac{\pi}{4}.$$

Now for a very large number of generated points,

$$\frac{\text{area of the circle}}{\text{area of the square}} = \frac{\text{no. of points generated inside the circle}}{\text{total no. of points generated or no. of points generated inside the square}}$$

that is,

$$\pi = 4 * \frac{\text{no. of points generated inside the circle}}{\text{no. of points generated inside the square}}$$

Parallelization methods

Naive approach

In our first implementation, we used a naive approach to distribute the computation of the value of π on CUDA blocks.

On the host device, we generated two vectors with random (x, y) pairs.

Then we copied the vectors into the memory of the graphics device. After that, we launched the CUDA kernel, which had the task of counting how many points land inside the circle and write the value of the counter in the **sampleCountPerThreads** shared vector. The first thread from each block then sums up the values from the thread counter and stores the result in a **device_sampleCountPerBlock** vector. The host device then does the computation of π, after summing up the values from the **device_sampleCountPerBlock** vector.

Deficitary aspects and optimizations

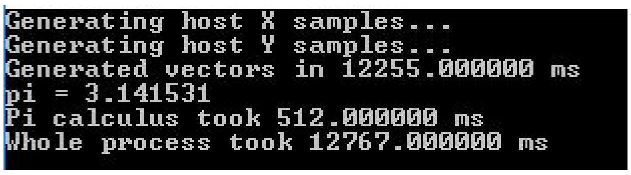
The first and most important optimization would be generating the random samples directly on the GPU threads. This would eliminate the severe overhead on the CPU.

Hardware & software devices used

CPU	GPU	OS	CUDA version
Intel Core i7-4790 @ 3.6 GHz	GeForce GTX 660, 2048MB	Windows 10 Pro	10.2

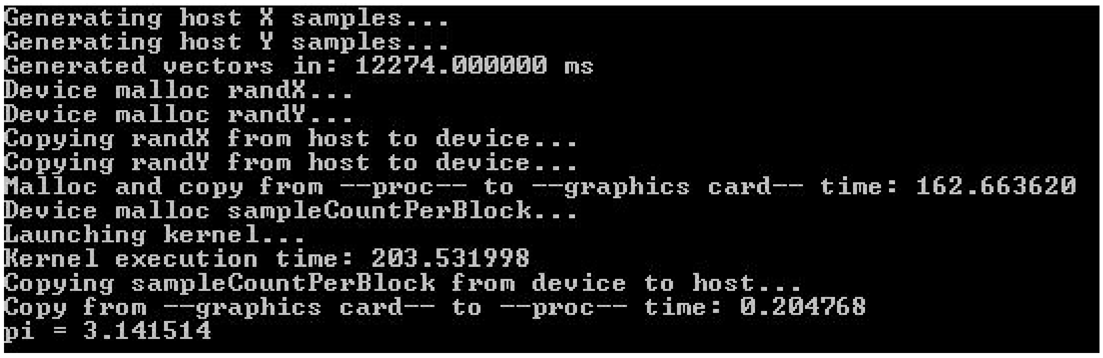
Results analysis

No. of samples	CPU version (ms)		GPU version (ms)				Acceleration
	Samples generation	**Pi computation**	**Samples generation**	**Copy data host -> device**	**Kernel computation**	**Copy data device -> host**	
10^5	**13**	**1**	**12**	**0.2975**	**159.7696**	**0.0945**	**0.08**
10^6	**125**	**7**	**127**	**1.7931**	**159.7395**	**0.0543**	**0.46**
10^7	**1238**	**62**	**1235**	**16.0672**	**160.0351**	**0.0619**	**0.92**
10^8	**12255**	**510**	**12274**	**162.6636**	**203.5319**	**0.2047**	**1.01**



```
Generating host X samples...
Generating host Y samples...
Generated vectors in 12255.000000 ms
pi = 3.141531
Pi calculus took 512.000000 ms
Whole process took 12767.000000 ms
```

Output for the CPU version



```
Generating host X samples...
Generating host Y samples...
Generated vectors in: 12274.000000 ms
Device malloc randX...
Device malloc randY...
Copying randX from host to device...
Copying randY from host to device...
Malloc and copy from --proc-- to --graphics card-- time: 162.663620
Device malloc sampleCountPerBlock...
Launching kernel...
Kernel execution time: 203.531998
Copying sampleCountPerBlock from device to host...
Copy from --graphics card-- to --proc-- time: 0.204768
pi = 3.141514
```

Output for the GPU version