HPC_Spring 2017

CUDA Mini Project

Kaiqin Huang

For this CUDA project, I continue working on using Monte Carlo algorithm to estimate the pi value, same as the OpenMP project. More details about how the algorithm works has been described in my OpenMP project write-up.

In the CUDA code below, I include comments on CUDA process flow such as allocate memory on GPU, invoke the kernel, copy from GPU back to CPU, free the memory on GPU, and more. To compile, I use nvcc –o cuda_pi cuda_monte_carlo_pi.cu.

Later I changed the numbers of blocks, threads, and trials per thread to compare the accuracy. I found that the more trials each thread has, the larger error it leads to, which is kind of unexpected. For the numbers of blocks and threads, the more blocks or threads we delegate, the higher accuracy we get. Basically, this is because the code is calculating a pi estimate in each thread, and then takes the average among all blocks and threads.

Compared with the previous OpenMP version, CUDA returns much higher accuracy. As for timing, although it is not a huge project that takes minutes or hours to run, I can still tell CUDA is faster than OpenMP. The process finishes rapidly.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <curand_kernel.h> // CURAND lib header file
#define TRIALS_PER_THREAD 1024 // Set the value for global variables
#define BLOCKS 512
#define THREADS 512
#define PI 3.1415926535 // Known value of pi, to calculate error

__global__ void pi_mc(float *estimate, curandState *states){
	unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;
	int points_in_circle = 0;
	float x, y;

	// Initialize CURAND
	curand_init(tid, 0, 0, &states[tid]);
	for(int i = 0; i < TRIALS_PER_THREAD; i++){
		x = curand_uniform(&states[tid]);
		y = curand_uniform(&states[tid]);
		// Count if x & y is in the circle
		points_in_circle += (x*x + y*y <= 1.0f);
	}
	estimate[tid] = 4.0f * points_in_circle / (float) TRIALS_PER_THREAD;
}

int main(int argc, char *argv[]){
	float host[BLOCKS * THREADS];
	float *dev;
```

```
	curandState *devStates;

	// Allocate memory on GPU
	cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float));
	cudaMalloc((void **) &devStates, BLOCKS * THREADS * sizeof(curandState));

	// Invoke the kernel
	pi_mc<<<BLOCKS, THREADS>>>(dev, devStates);

	// Copy from device back to host
	cudaMemcpy(host, dev, BLOCKS * THREADS * sizeof(float),
cudaMemcpyDeviceToHost);

	// Free the memory on GPU
	cudaFree(dev);
	cudaFree(devStates);

	// Get the average estimate pi value among all blocks and threads, and calculate error
	float pi_gpu = 0.0;
	for(int i = 0; i < BLOCKS * THREADS; i++){
		pi_gpu += host[i];
	}
	pi_gpu /= (BLOCKS * THREADS);
	printf("Trials per thread is: %d, number of blocks is: %d, number of threads is: %d\n",
	 TRIALS_PER_THREAD, BLOCKS, THREADS);
	printf("CUDA estimate of PI = %f [error of %f]\n", pi_gpu, pi_gpu - PI);

	return 0;
}
```

Result

TRIALS_PER_THREAD	2048	1024	1024	1024	10000
NUM_BLOCKS	256	256	512	512	512
NUM_THREADS	256	256	256	512	512
Estimate PI	3.141675	3.141660	3.141623	3.141602	3.139749
Error	0.000083	0.000067	0.000030	0.000009	-0.001844

References

https://www.olcf.ornl.gov/tutorials/cuda-monte-carlo-pi/

https://github.com/JRWynneIII/Monte-Carlo-Pi--Cuda-OpenACC-/blob/master/thrust.cu

https://hpcc.usc.edu/files/2013/07/CUDAworkshop.pdf