# CUDA Parallel Programming Problem Set 7

R08244004 Shin-Rong Tsai

- ## Monte Carlo Integration in 10 Dimension

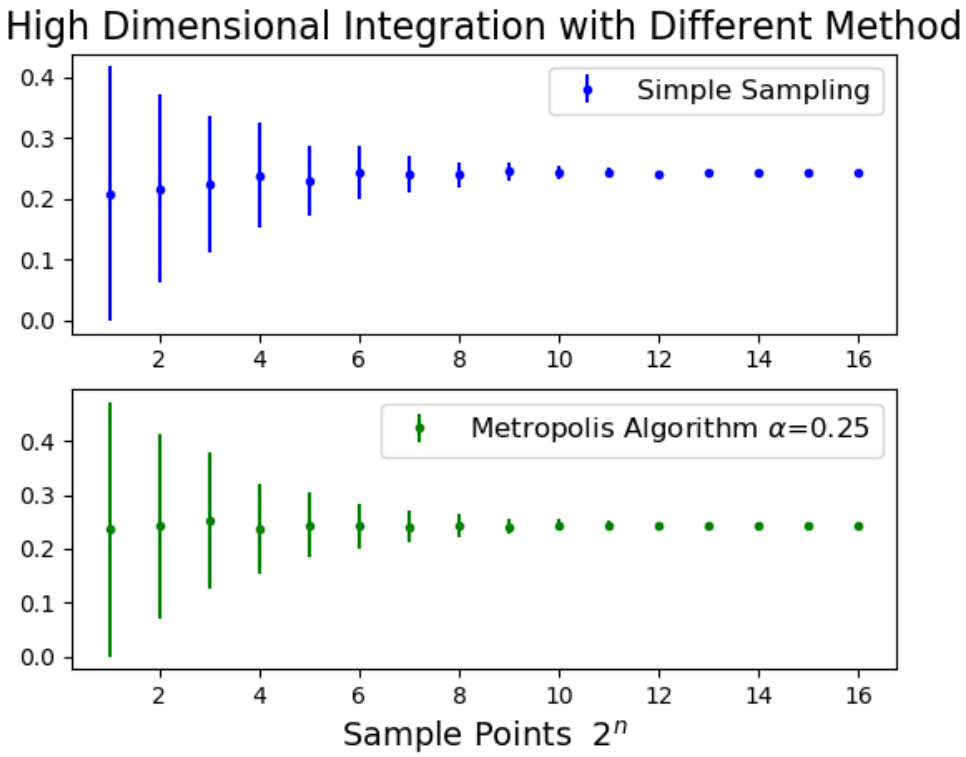
### I. Result

Run and compile *monte_carlo_integration.c* with gsl library on my computer.

```
cindytsai@TURQUOISEA /cygdrive/d/GitHub/CUDA_Parallel_Programming/Assignment/ProblemSet7
$ gcc -o q1.exe monte_carlo_integration.c -lgsl

cindytsai@TURQUOISEA /cygdrive/d/GitHub/CUDA_Parallel_Programming/Assignment/ProblemSet7
$ ./q1.exe
```

After finding the most proper $\alpha = 0.25$ I think, plot the results of Simple Sampling and Metropolis Algorithm with *plot_result.py*. How I find the weight function parameter $\alpha$ is in the discussion.




### II. Discussion

A. Finding the weight function parameter $\alpha$

Since $\frac{1}{1+x_1^2+x_2^2+\cdots+x_{10}^2}$, every $x_i$ is symmetric, so weight function parameter of each dimension of $x$ should be the same.

$$w(x_i) = Ce^{-\alpha x_i}$$

And weight function must satisfy

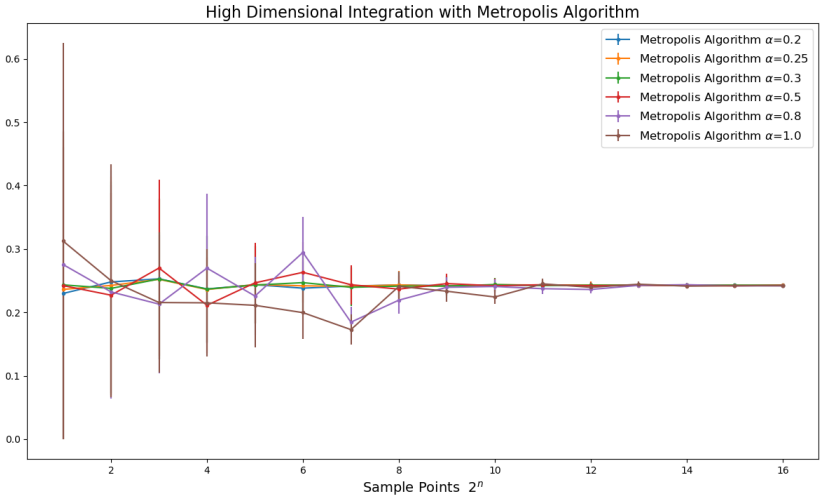
$$\int_0^1 w(x) = 1$$

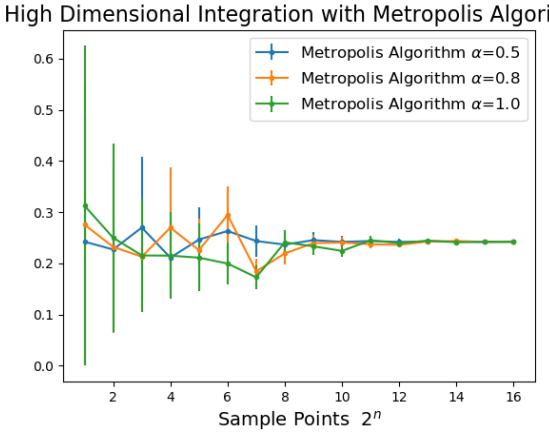
We only have to determine $\alpha$, and calculate $C$ through

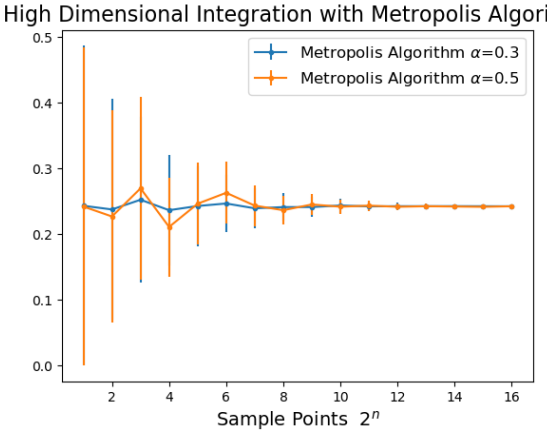
$$C = \frac{\alpha}{1 - e^{-\alpha}}$$

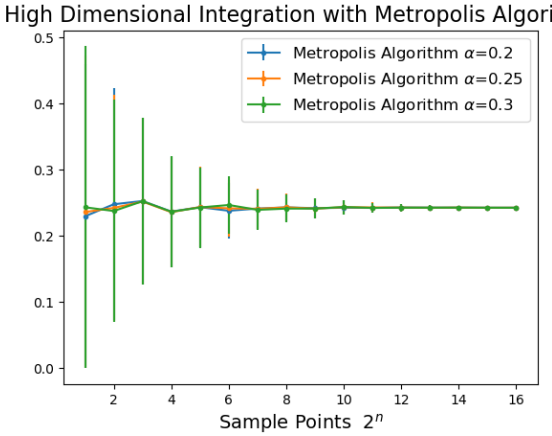
Because a better weight function would guess the distribution of $f(x)$ more correctly, which means it does not vary that much when the number of sample points $N$ are small and it converges faster as $N$ becomes larger.

So through trying a series of $\alpha$, and plot them (in file *finding_alpha.py*), we can clearly see that it is reasonable to choose $\alpha = 0.25$.



B. Compare the result of Simple Sampling and Metropolis Algorithm

We can see that with a proper α Metropolis Algorithm converge faster than Simple Sampling. And we can check the central limit theorem by dividing each of their error with $\sqrt{2}$, then we will approximately get the error of the next row below.

| | N | SS | SSerror | MA | MAerror |
|---|---|---|---|---|---|
| 0 | 2 | 0.208967 | 0.209095 | 0.236287 | 0.236325 |
| 1 | 4 | 0.217261 | 0.153889 | 0.242548 | 0.171610 |
| 2 | 8 | 0.224875 | 0.112991 | 0.252137 | 0.126175 |
| 3 | 16 | 0.239284 | 0.085564 | 0.236144 | 0.083605 |
| 4 | 32 | 0.229877 | 0.058243 | 0.243853 | 0.061048 |
| 5 | 64 | 0.242917 | 0.043370 | 0.241930 | 0.042804 |
| 6 | 128 | 0.241457 | 0.030531 | 0.241188 | 0.030177 |
| 7 | 256 | 0.239467 | 0.021522 | 0.243438 | 0.021543 |
| 8 | 512 | 0.245331 | 0.015570 | 0.241504 | 0.015109 |
| 9 | 1024 | 0.244152 | 0.010963 | 0.243642 | 0.010778 |
| 10 | 2048 | 0.243964 | 0.007731 | 0.243224 | 0.007608 |
| 11 | 4096 | 0.240969 | 0.005400 | 0.243052 | 0.005376 |
| 12 | 8192 | 0.242928 | 0.003850 | 0.242785 | 0.003797 |
| 13 | 16384 | 0.242899 | 0.002722 | 0.243009 | 0.002688 |
| 14 | 32768 | 0.243325 | 0.001928 | 0.242757 | 0.001898 |
| 15 | 65536 | 0.242700 | 0.001360 | 0.242842 | 0.001343 |

Through the result, we can say that this 10-dimension integral is approximately 0.243.

- **GPU accelerated Monte Carlo integration in 10-dimension**

**I. Result**

I write the GPU code for Simple Sampling and Metropolis Algorithm in files *MCintegration_SimpleSampling_ngpu.cu* and *MCintegration_Metropolis_ngpu.cu* respectively. The results of different number of sample points N for GPU with proper block size and grid size are basically the same as CPU.

The integration results are already shown in previous section, so I'll discuss more on GPU's performance.

```
r08244002@twqcd135:~/PS7$ nvcc -arch=compute_52 -code=sm_52,sm_52 -m64 --compiler-options
-fopenmp -o q2-MA.exe MCintegration_Metropolis_ngpu.cu
r08244002@twqcd135:~/PS7$ ./q2-MA.exe

* Initial parameters for GPU:
  Enter the number of GPUs: 2
2
  Enter the GPU ID (0/1/...): 0
0
  Enter the GPU ID (0/1/...): 1
1

* Solve Monte Carlo Integration in 10-dim:
  Enter the number of sample points: 65536
65536
  Enter the power (m) of threads per block (2^m): 10
10
threads per block = 1024
  Enter the number of blocks per grid: 10
10
```

## II. Discussion

### A. Implementation of getting random number for individual threads.

For simplicity, I send current time plus the OpenMP threads number times whatsoever to GPU, and then plus its own unique id as seed, to initialize cuda random number generator.

For this I assume that each seeds are not correlated, if they are, then this will affect the results of Monte Carlo Integration. From the results, it seems that they don't, so I guess that's OK.

### B. Implementation of Simple Sampling and Metropolis Algorithm

1. Simple Sampling

   Since the repeated part is generating $\{x_1 \cdots x_{10}\}$, computing $f(x_1 \cdots x_{10})$ and $f^2$, and finally, sum them up respectively. So the strategy I take:

   i. Let all the threads compute their own $f$ and $f^2$
   ii. Add $\text{BlockSize} \times \text{GridSize}$ to threads' i.
   iii. Go to next round(i.) if current i is smaller than N.
   iv. Finally, do the parallel reduction for each block.
   v. Pass the mean and sigma result back to CPU, and do some further computation.

2. Metropolis Algorithm

   The structure is generally the same as Simple Sampling, but with dividing the weight function, and determine the acceptance.

   So basically, we only have to change step i. to the metropolis algorithm, and the rest is the same.

### C. The optimal block size and grid size for Simple Sampling(SS)

Speed up rate of SS-1GPU-65536:

| BlockSize / GridSize | Simple Sampling 1GPU N = 65536 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 1.69 | 4.14 | 7.68 | 14.11 | 22.86 | 32.33 | 36.62 | 44.00 | 35.29 | 22.89 |
| $10^2$ | 15.86 | 25.42 | 34.46 | 41.89 | 43.80 | 37.89 | 24.38 | 11.06 | 5.02 | |
| $10^3$ | 17.67 | 16.94 | 15.31 | 13.20 | 9.33 | 4.05 | | | | |
| $10^4$ | 2.33 | 1.55 | | | | | | | | |

Speed up rate of SS-2GPU-65536:

| BlockSize / GridSize | Simple Sampling 2GPU N = 65536 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 2.13 | 7.30 | 13.27 | 19.95 | 27.68 | 32.10 | 41.07 | 40.66 | 33.49 | 21.38 |
| $10^2$ | 21.17 | 26.27 | 32.95 | 38.37 | 37.48 | 32.57 | 20.61 | 9.82 | | |
| $10^3$ | 19.05 | 15.50 | 13.82 | 12.57 | 8.87 | | | | | |
| $10^4$ | 2.27 | | | | | | | | | |

1. It seems that time used to run through sample points $N = 65536$ and $N = \frac{65536}{2}$ are the same, so 2GPU did not accelerate here.
2. We know that the optimal block size and grid size are determined by the computational strength. The optimal block size and grid size for Simple Sampling is where $BlockSize \times GridSize$ (Threads Per Grid) falls between 1600 to 3200. This is the same in either 1GPU or 2GPU, since it is computational strength that matters.

Speed up rate of SS-1GPU-81920000:

| BlockSize / GridSize | Simple Sampling 1GPU N = 81920000 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 2.58 | 5.14 | 10.20 | 20.16 | 39.29 | 77.40 | 144.86 | 262.05 | 419.92 | 539.65 |
| $10^2$ | 24.16 | 45.99 | 97.80 | 169.56 | 388.32 | 558.24 | 667.04 | 597.00 | 622.63 | 540.56 |
| $10^3$ | 46.78 | 93.50 | 152.28 | 344.78 | 578.59 | 659.92 | 582.54 | 410.43 | 20.07 | 6.81 |
| $10^4$ | 53.45 | 98.21 | 181.25 | 328.02 | 76.74 | 13.64 | 5.12 | 2.26 | 0.90 | 0.33 |

Speed up rate of SS-2GPU-81920000:

| BlockSize / GridSize | Simple Sampling 2GPU N = 81920000 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 5.13 | 10.18 | 20.17 | 39.15 | 75.52 | 137.37 | 269.15 | 479.54 | 790.93 | 934.48 |
| $10^2$ | 47.13 | 97.64 | 163.94 | 384.59 | 585.71 | 961.03 | 1106.07 | 991.91 | 897.60 | 799.41 |
| $10^3$ | 79.54 | 154.88 | 330.17 | 498.55 | 988.83 | 1007.39 | 864.79 | 351.20 | 20.16 | 6.79 |
| $10^4$ | 106.09 | 173.98 | 326.14 | 455.17 | 75.65 | 13.61 | 5.10 | 2.25 | 0.90 | 0.33 |

1. With a proper block size and grid size, 2GPU accelerates for almost two times faster. The workloads for running through $N = 81920000$ and $N = \frac{81920000}{2}$ are so different that it reflects on the speed up rate in 1GPU and 2GPU.

2. The optimal block size and grid size are the same in 1GPU and 2GPU. I marked them with yellow blocks.
3. The speed up rate tables in 1GPU and 2GPU are alike, since it is computational strength that matters.

D. <u>The optimal block size and grid size for Metropolis Algorithm(MA)</u>

Speed up rate of MA-1GPU-65536:

| BlockSize / GridSize | Metropolis Algorithm 1GPU N = 65536 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 6.41 | 15.00 | 28.12 | 50.33 | 79.55 | 111.01 | 131.73 | 144.33 | 123.32 | 100.35 |
| $10^2$ | 57.58 | 88.52 | 116.22 | 141.50 | 143.03 | 134.06 | 98.05 | 43.37 | 21.65 | |
| $10^3$ | 64.11 | 64.87 | 61.69 | 54.73 | 38.45 | 16.74 | | | | |
| $10^4$ | 9.39 | 6.11 | | | | | | | | |

Speed up rate of MA-2GPU-65536:

| BlockSize / GridSize | Metropolis Algorithm 2GPU N = 65536 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 7.93 | 27.21 | 46.44 | 77.59 | 105.96 | 127.16 | 139.25 | 139.07 | 106.65 | 87.45 |
| $10^2$ | 82.59 | 101.18 | 114.58 | 138.54 | 117.58 | 128.33 | 80.95 | 40.25 | | |
| $10^3$ | 64.25 | 62.56 | 55.17 | 52.25 | 35.01 | | | | | |
| $10^4$ | 9.48 | | | | | | | | | |

1. Again, time used for computing $N = 65536$ and $N = \frac{65536}{2}$ are almost the same, so 2GPU did not accelerate, even though Metropolis Algorithm is more complicated than Simple Sampling.
2. The optimal block size and grid size for 1 and 2GPU are slightly different, but their distribution are the same, see the yellow blocks.

Speed up rate of MA-1GPU-81920000:

| BlockSize / GridSize | Metropolis Algorithm 1GPU N = 81920000 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 9.50 | 18.84 | 37.47 | 74.10 | 147.30 | 281.65 | 501.89 | 994.57 | 1646.14 | 2013.32 |
| $10^2$ | 89.48 | 176.60 | 328.99 | 631.75 | 1205.47 | 2030.02 | 2474.78 | 2210.94 | 2460.58 | 2087.50 |
| $10^3$ | 177.82 | 325.57 | 616.10 | 1059.47 | 1586.86 | 2334.31 | 2274.44 | 1823.69 | 92.02 | 31.08 |
| $10^4$ | 190.73 | 352.30 | 650.45 | 1169.34 | 341.62 | 62.05 | 23.25 | 10.31 | 4.12 | 1.51 |

Speed up rate of MA-2GPU-81920000:

| BlockSize / GridSize | Metropolis Algorithm 2GPU N = 81920000 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
| $10^1$ | 20.59 | 40.59 | 80.93 | 159.39 | 303.86 | 562.70 | 1068.01 | 1979.43 | 3256.52 | 3926.42 |
| $10^2$ | 189.83 | 347.28 | 670.38 | 1251.53 | 2406.64 | 3986.20 | 3101.18 | 3672.04 | 2963.49 | 3196.62 |
| $10^3$ | 339.69 | 645.22 | 1236.40 | 2136.46 | 3968.33 | 4113.02 | 3838.92 | 2358.98 | 101.72 | 34.32 |
| $10^4$ | 389.10 | 722.36 | 1317.53 | 1936.87 | 385.51 | 68.29 | 25.73 | 11.37 | 4.54 | 1.66 |

1. 2GPU accelerates for almost 2 times the speed up rate of 1GPU.
2. The optimal block size and grid size for 1GPU and 2GPU are in the yellow blocks. We can see that they are different, because that calculate $N = 81920000$ and $N = \frac{81920000}{2}$ of Metropolis Algorithm is indeed a lot different than in Simple Sampling. So the optimal settings are different.

E. Errors and their time used compared with GPU

I compared the results from optimal block size and grid size only. Defined the error as

$$\text{error} = \frac{|MeanGPU - MeanCPU|}{MeanCPU}$$

how much mean value calculate by GPU is different from CPU.

Errors:

| | Simple Sampling | | Metropolis Algorithm | |
|---|---|---|---|---|
| Sample Points N | 65536 | 81920000 | 65536 | 81920000 |
| 1GPU | $1.76 \times 10^{-3}$ | $8.24 \times 10^{-6}$ | $3.62 \times 10^{-4}$ | $1.19 \times 10^{-4}$ |
| 2GPU | $1.69 \times 10^{-3}$ | $3.71 \times 10^{-5}$ | $8.53 \times 10^{-4}$ | $1.11 \times 10^{-4}$ |

1. Cause Monte Carlo integration is determined by the random numbers at each run, so the more sample points, it should be more close to the result from CPU.
2. Numbers of GPU used basically won't make the error smaller, which should be the case in Simple Sampling. And as the sample points becomes larger, the error is smaller.
3. For Metropolis Algorithm, the algorithm requires a series of random number such that they are not uniform. But the way I implement it makes each threads start their own series, which means by increasing sample points, the error won't decrease that much as in Simple Sampling.

Time used (ms):

| | Simple Sampling | | Metropolis Algorithm | |
|---|---|---|---|---|
| Sample Points N | 65536 | 81920000 | 65536 | 81920000 |
| 1GPU | 4.4 | 59.8 | 4.7 | 73.0 |
| 2GPU | 9.0 | 35.1 | 10.4 | 45.4 |
| CPU | 31.0 | 38153.2 | 136.6 | 173681.2 |

1. Metropolis Algorithm spends more time than Simple Sampling, as it spends some time to test for the acceptance.
2. Compared from the error, time used, and the implementation, I think it is better to use Simple Sampling when doing integration. Because determining the weight function is time consuming and the implementation of Metropolis Algorithm in GPU is not that ideal (in my case), and the worst of all is that as sample points increase, the error won't drop that much as in Simple Sampling.