17级并行与分布式计算 - Final project

HW7 17341137 宋震鹏

$1 实验要求

- 选题：Parallel Monte Carlo algorithm
- 使用CUDA实现 `并行蒙特卡罗算法`。
- 在本例中，将使用并行蒙特卡罗算法实现圆周率π的计算。

$2 实验分析

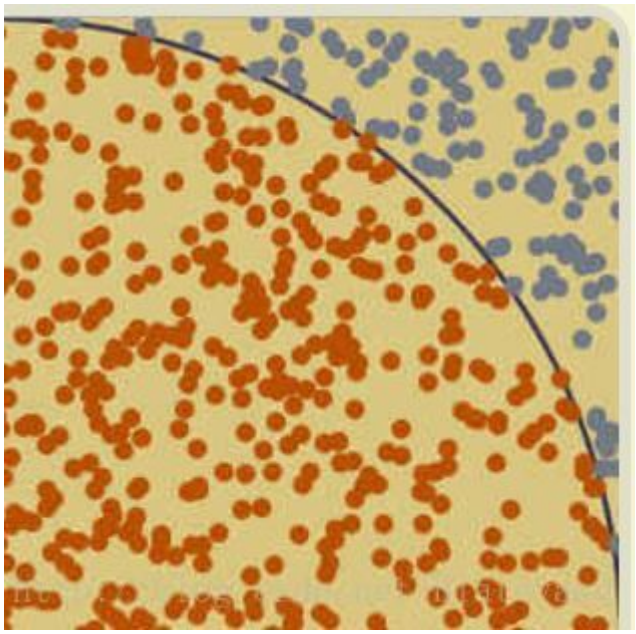
- **Monte Carlo algorithm**
  - **简述**

    Monte Carlo method，也称随机抽样法、统计模拟方法，是二十世纪四十年代中期由于科学技术的发展和电子计算机的发明，而被提出的一种以概率统计理论为指导的一类非常重要的数值计算方法。是指使用随机数（或更常见的伪随机数）来解决很多计算问题的方法。

  - **思想**

    当所求解问题是**某种随机事件出现的概率**，或者是**某个随机变量的期望值**时，通过某种“实验”的方法，以这种事件出现的频率估计这一随机事件的概率，或者得到这个随机变量的某些数字特征，并将其作为问题的解。

  - **实验设计**

  - 在本例中，设图中正方形边长为1，易知该内切圆面积S_O与正方形面积S_C的比值为$\frac{\pi}{4}$，因此，设计实验：
    - 利用随机数，在该正方形中随机取点，统计落在内切圆中的点数N_O。落在内切圆内部的点数以及所取总点数N的比值即为$\frac{\pi}{4}$.

    - $\frac{N_O}{N} = \frac{S_O}{S_C} = \frac{\pi}{4}$

- **算法过程**

    1. 随机数取[0,1]区间小数，每次取两个随机数：
        - $posx$：落在图上的x坐标。
        - $posy$：落在图上的y坐标。
    2. 根据判断式：

    $posx^2 + posy^2 \leq 1 \ (*)$

    3. 维护变量：
        - $Count$：落在范围内的数量，满足*式时加1。
    4. 计算比率：
        - $\frac{N_O}{N} = \frac{Count}{Total}$
    5. 将结果乘4得到pi的估计值：
        - $pi = 4 \times \frac{N_O}{N}$

- **并行化**

    - 先在主进程完成随机数的计算，并复制到device中。

```
double generateRand() {
    return (rand() % 10000) * 0.0001;
}

// generate random position.
for(int i = 0;i<N;i++){
    rand_x_h[i] = generateRand();
    rand_y_h[i] = generateRand();
}

// memory copy from Host to Device.
CUDA_CHECK(
    cudaMemcpy( (void*)rand_x_d, (void*)rand_x_h,
                N * sizeof(double), cudaMemcpyHostToDevice)
);
CUDA_CHECK(
    cudaMemcpy( (void*)rand_y_d, (void*)rand_y_h,
                N * sizeof(double), cudaMemcpyHostToDevice)
);
```

    - 调用内核，完成计算并复制回主进程。

```
// GPU Kernel called.
plot <<< grid, threads >>> (C_count_d, rand_x_d, rand_y_d, N);

// memory copy from Device to Host.
CUDA_CHECK(
    cudaMemcpy( (void*)C_count_h, (void*)C_count_d,
                N * sizeof(int), cudaMemcpyDeviceToHost)
);

// pi calculation.
for(int i = 0;i<N;i++){
    C_count += C_count_h[i];
}
```

- 完成计算

```
pi = 4.0 * (double(C_count)/double(T_count));
```

$3 实验代码

实验取了三个不同的N值，以比较N的规模对估计精度的影响，同时考虑CUDA并发计算的速度受到的影响。

展示以N=100000时的代码。

```
/* ./MonteCarlo(n=100000).cu */

/*
*   Monte Carlo Algorithm
*   for Pi calculating.
*
*   随机取点次数：1000, 10000, 100000
*   本例中，N=100000
*/
#include <sys/time.h>
#include <cstdio>
#include <cstdlib>
#include "error_checks.h"
#define PI 3.141592653589793

// GPU kernel
__global__
void plot(int *C_count, double* ranx, double* rany, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    double pos_x = ranx[idx];
    double pos_y = rany[idx];
    if (idx < N) {
        if(pos_x*pos_x + pos_y*pos_y <= 1.0) {
            C_count[idx]++;
        }
    }
```

```
}

double generateRand() {
    return (rand() % 10000) * 0.0001;
}

int main() {
    timeval t1, t2; // Structs for timing
    // const int N = 1000;
    // const int N = 10000;
    const int N = 100000;
    // Result-variables.
    int C_count = 0;
    int T_count = N;
    double pi = 0.0;

    // Host-variables.
    int *C_count_h = new int[N];
    double rand_x_h[N];
    double rand_y_h[N];

    // Device-variables.
    int *C_count_d;
    double *rand_x_d, *rand_y_d;

    // memory allocation for Device-variables.
    CUDA_CHECK( cudaMalloc( (void**)&C_count_d, N * sizeof(int)) );
    CUDA_CHECK( cudaMalloc( (void**)&rand_x_d, N * sizeof(double)) );
    CUDA_CHECK( cudaMalloc( (void**)&rand_y_d, N * sizeof(double)) );

    // generate random position.
    for(int i = 0;i<N;i++){
        rand_x_h[i] = generateRand();
        rand_y_h[i] = generateRand();
    }

    // memory copy from Host to Device.
    CUDA_CHECK( cudaMemcpy( (void*)rand_x_d, (void*)rand_x_h, N * sizeof(double),
cudaMemcpyHostToDevice) );
    CUDA_CHECK( cudaMemcpy( (void*)rand_y_d, (void*)rand_y_h, N * sizeof(double),
cudaMemcpyHostToDevice) );

    // GPU version
    dim3 threads(128), grid((N+threads.x-1)/threads.x);

    // GPU Kernel called.
    gettimeofday( & t1, NULL);
    plot <<< grid, threads >>> (C_count_d, rand_x_d, rand_y_d, N);
    gettimeofday( & t2, NULL);

    // memory copy from Device to Host.
    CUDA_CHECK( cudaMemcpy( (void*)C_count_h, (void*)C_count_d, N * sizeof(int),
cudaMemcpyDeviceToHost) );
```

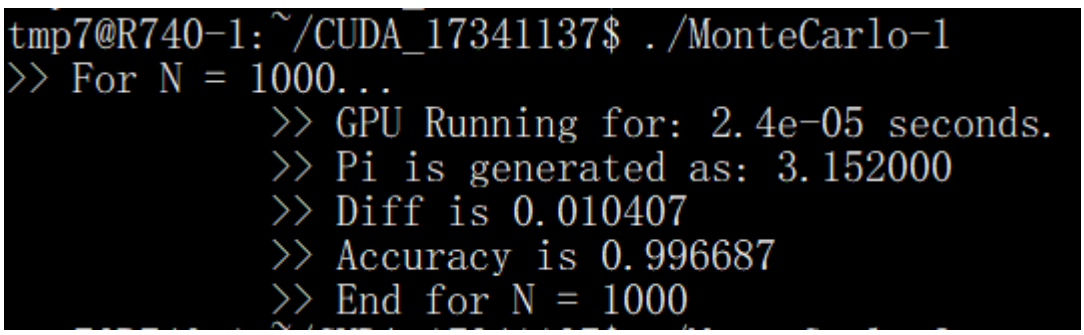
```
    // pi calculation.
    for(int i = 0;i<N;i++){
        C_count += C_count_h[i];
    }

    pi = 4.0 * (double(C_count)/double(T_count));

    // Resulting...
    printf(">> For N = %d...\n\
            >> GPU Running for: %g seconds.\n\
            >> Pi is generated as: %lf\n\
            >> Diff is %lf\n\
            >> Accuracy is %lf\f\
            >> End for N = %d\n",
           N, t2.tv_sec - t1.tv_sec +
           (t2.tv_usec - t1.tv_usec)/1.0e6, pi, abs(pi-PI), 1-(abs(pi-PI)/PI), N);

    return EXIT_SUCCESS;
}
```

后续展示实验截图并总结。

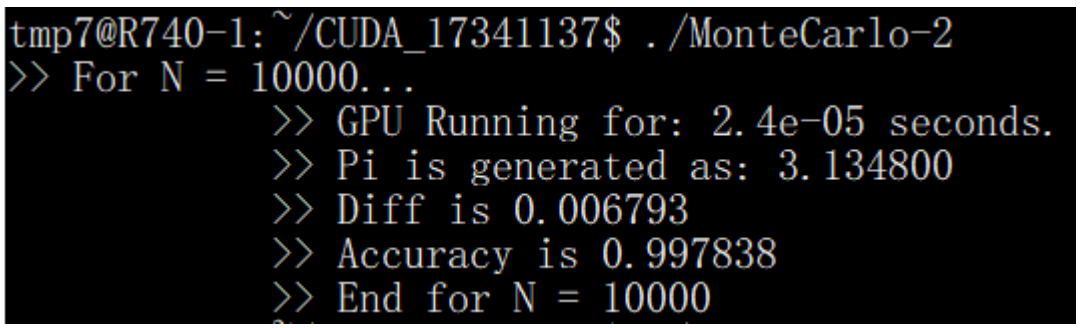
$4 实验截图

运行环境为老师提供的服务器端，作为用户 `tmp7`。

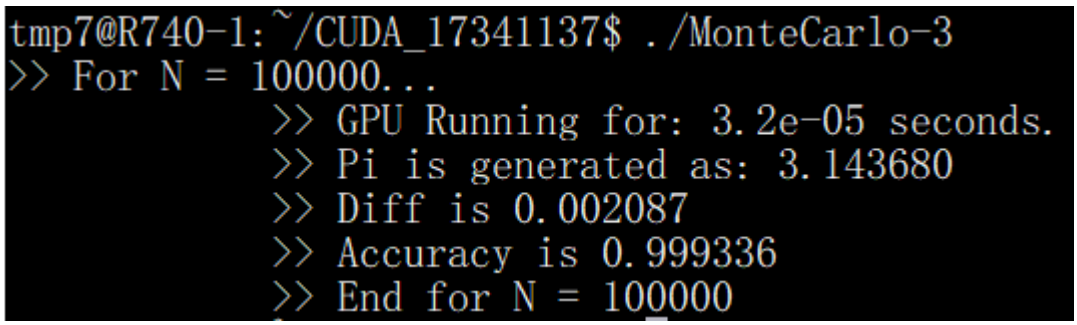
Case1：n = 1000



Case2：n = 10000



Case3：n = 100000



- 由实验截图可知，在100000以下数量级，GPU的运算速度数量级未受影响。
- 同时验证了，N越大的情况下，估计精度越高。

$5 实验总结

本次实验利用CUDA并行编程完成了并行蒙特卡洛算法（Parallel Monte Carlo algorithm）的设计。

首先，我先查阅资料，对蒙特卡罗算法及其串行实现进行了了解与学习，实现了串行算法后，开始考虑该算法的“可分性”，并初步进行并行算法的设计。在并行化设计过程中，最主要的思想就是让每个线程并行地进行“取点”操作，最后再汇总到主线程即可。

遇到的主要问题是，CUDA的子线程（device）中无法调用`__host__`函数（本计划在device中进行随机数的生成）。查阅资料后，考虑采取提前生成好随机数并通过数组结构传入给子线程，问题顺利解决。

总体而言，这次大作业从头开始完成了一个算法的并行设计，是对并行与分布式计算这门课的一个总结，也帮助我在考虑问题时，会多从“能否并发进行”角度进行考虑，我受益匪浅。

经过一个学期的学习，并行与分布式计算这门课给到了我很大的帮助，从OpenMP、MPI到MapReduce、CUDA，我们学习到了多种当下热门的并行编程语言，锻炼了并行思维。希望在日后的学习工作中能学以致用！

17341137 宋震鹏 19/6/20