Multicore Computing
Project #4 – problem2

Department : Software
Name : Jeong eui chan
Student ID : 20195914

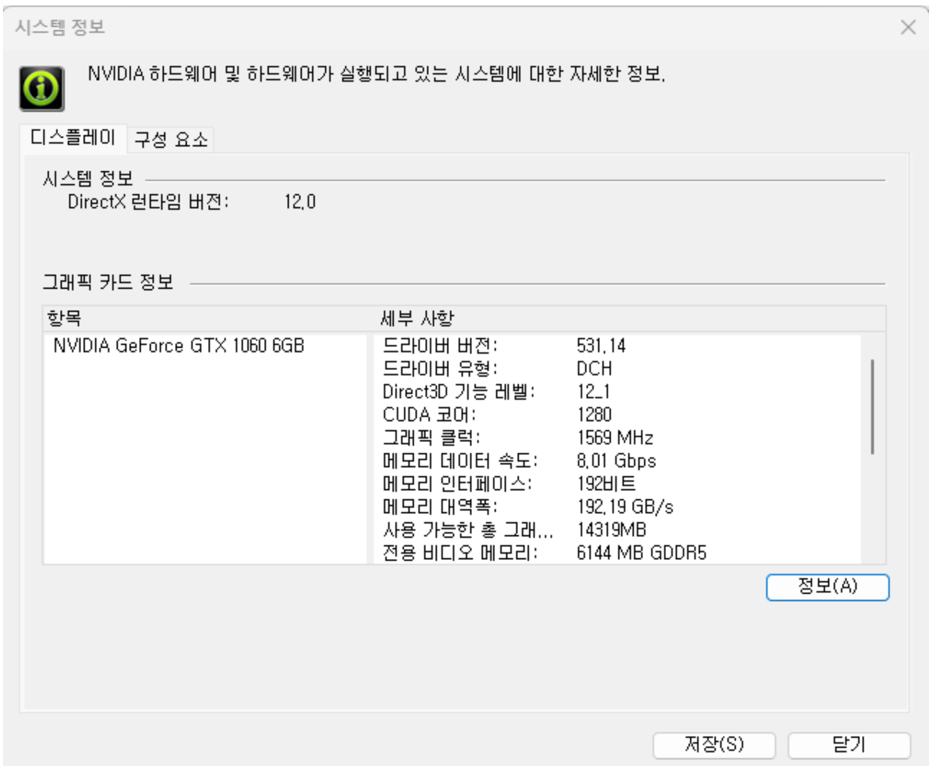
<Excution environment>

For openMP I used a MAC m1 laptop. The computer information of my MAC m1 is as shown in the picture below.

```
 jeong-uichan  ~  system_profiler SPHardwareDataType
Hardware:

    Hardware Overview:

      Model Name: MacBook Air
      Model Identifier: MacBookAir10,1
      Model Number: Z1250002VKH/A
      Chip: Apple M1
      Total Number of Cores: 8 (4 performance and 4 efficiency)
      Memory: 16 GB
      System Firmware Version: 8422.121.1
      OS Loader Version: 8422.121.1
      Serial Number (system): FVFFC3ADQ6LT
      Hardware UUID: EE341825-7FB8-52C6-8F1A-E68A1656E926
      Provisioning UDID: 00008103-001539513CEA001E
      Activation Lock Status: Enabled
```

For CUDA, I used my desktop with an NVIDIA graphics card. The information on the desktop is like the picture below.
You can check the information of the type of OS and the CPU being used through the picture.

You can check the information of the NVIDIA graphics card used in my desktop, and you can check the number of CUDA cores.



<Entire source code – thrust_ex.cu>

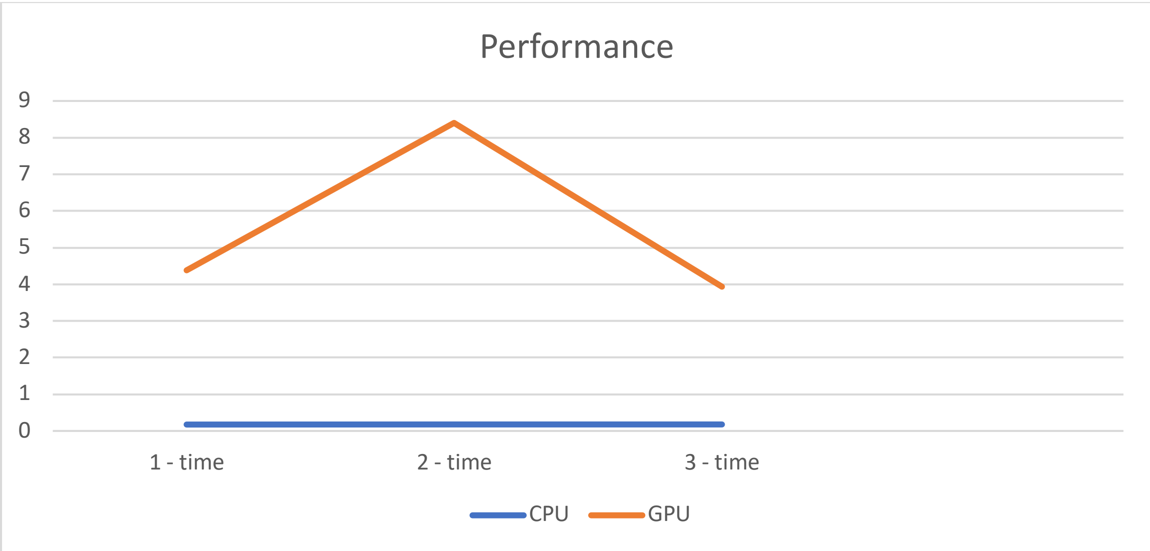
```
#include <thrust/device_vector.h> // 효율적인 GPU 연산을 위한 thrust 라이브러리를 불러온다
#include <thrust/transform_reduce.h>
#include <thrust/functional.h>
#include <thrust/sequence.h>
#include <cmath>
#include <iostream>

// 연산을 수행하는 함수 정의
struct pi_functor
{
    double step;
    pi_functor(double step) : step(step) {} // 구조체 생성자.

    __host__ __device__
        double operator()(const int& i) const // 원주율을 계산하는 식을 정의
    {
        double x = (i + 0.5) * step;
        return 4.0 / (1.0 + x * x);
    }
};

// 원주율을 계산하기 위해 위에서 정의한 함수를 적용하여 합계 구하기
double compute_pi(thrust::device_vector<int>::iterator first,
    thrust::device_vector<int>::iterator last,
    double step)
{
    return step * thrust::transform_reduce(first, last, pi_functor(step), 0.0, thrust::plus<double>());
}

int main()
{
    const long num_steps = 1000000000; // 총 단계 수
    double step = 1.0 / (double)num_steps; // 단계 크기

    // 숫자를 저장할 디바이스 벡터를 선언하고 이에 0부터 num_steps-1까지의 숫자를 생성
    thrust::device_vector<int> d_sequence(num_steps);
    thrust::sequence(d_sequence.begin(), d_sequence.end());

    // CUDA 이벤트를 생성하여 연산 시간을 측정
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    // 위에서 선언한 함수를 호출하여 원주율을 계산
    double pi = compute_pi(d_sequence.begin(), d_sequence.end(), step);

    // 계산이 끝났으므로 CUDA 이벤트를 기록하고 시간을 측정
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
```

```
    float timeDiff;
    cudaEventElapsedTime(&timeDiff, start, stop);

    // 계산에 걸린 시간과 계산 결과를 출력
    std::cout << "Execution Time : " << timeDiff / 1000.0 << " sec" << std::en
dl;
    std::cout << "pi = " << pi << std::endl;

    // 생성했던 CUDA 이벤트를 제거
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

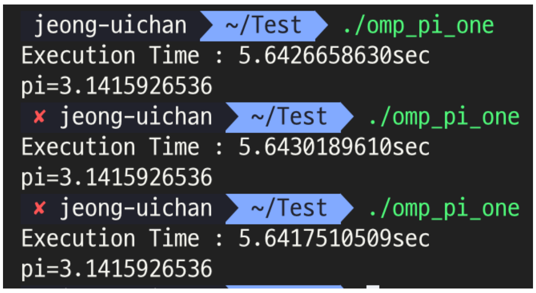
< execution time table and graphs>

	Sequential program using only CPU	Using thrust
Execution time - 1	5.642665	0.228344
Execution time - 2	5.643018	0.119023
Execution time - 3	5.641751	0.254164

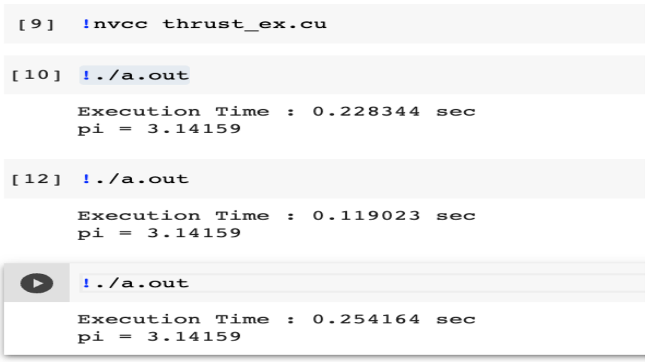
	Sequential program using only CPU	Using thrust
Performance time - 1	0.17722	4.37936
Performance time - 2	0.17721	8.40174
Performance time - 3	0.17725	3.93447



< explanation/interpretation on the result>



The above picture is the program execution screen using the CPU. I ran it 3 times and the execution time came out similar.



The above image is the result of rewriting the code using Thrust and then executing it on the GPU. Similarly, the code was run three times, and similar results were obtained, with execution times around 0.xxx seconds.

This performance is significantly faster than when performing computations using the CPU. The reason for this is that the area of each rectangle can be calculated independently of all the others. GPUs excel at this kind of parallel computation, being able to calculate the areas of many rectangles simultaneously. In contrast, CPUs compute the areas of the rectangles one by one, leading to a stark difference in performance. The parallel computation is performed in the compute_pi function by calling thrust::transform_reduce. Subsequently, pi_functor calculates the area of a single rectangle.