1. Execution Environment

Google Colab

2. Calculate Pie using thrust

- Prob2

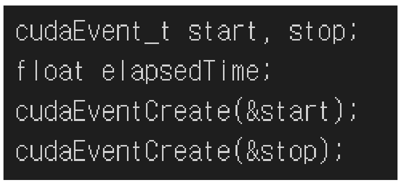
CUDA can be programmed in C/C++, but it does not provide direct access to the STL (Standard Template Library) in the device code. However, Thrust is a CUDA-based vectorized algorithm library that allows programmers to write code that runs on the GPU using C++ syntax. With Thrust, you can leverage the power of CUDA to perform computations, such as calculating Pi.

Thrust has the advantage of providing a similar interface to C++'s STL, making it easier for programmers familiar with the STL to learn and use. As someone with experience in C++, I found it relatively easy to code using Thrust.

The algorithm is as follows:

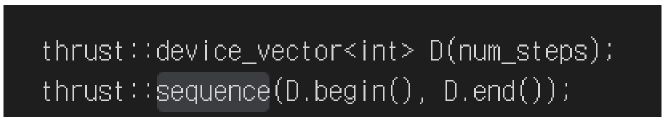
- Insert numbers from 0 to N-1 into a vector in sequential order.
- Apply the integral operation to each element.
- Sum all the elements.
- Divide the sum by N.

3. Code



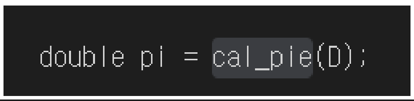
```
cudaEvent_t start, stop;
float elapsedTime;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

Thrust automatically classifies code into the device region during compilation, so there is no need to explicitly use keywords like global. It is common practice to declare start and end variables to measure time in a program and print the execution time when the task is completed.

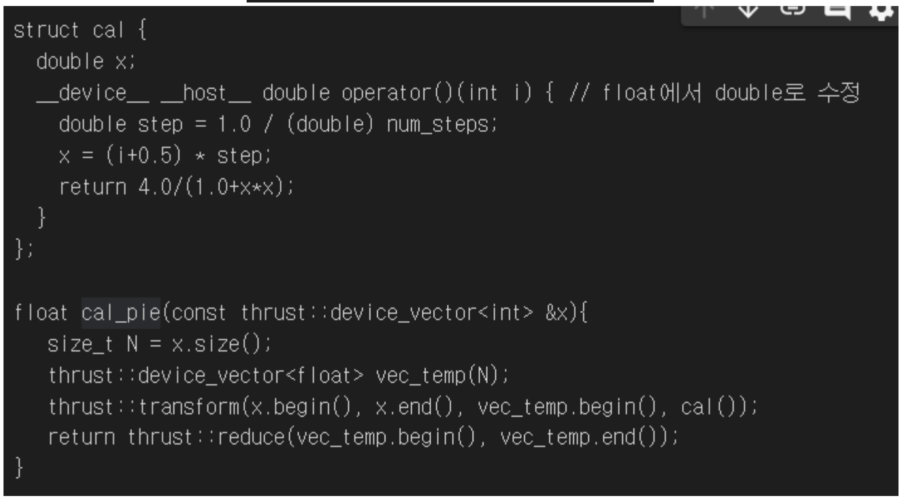


```
thrust::device_vector<int> D(num_steps);
thrust::sequence(D.begin(), D.end());
```

To begin, we create a device_vector. A device_vector automatically resizes itself and is similar to the vector in the STL. We specify the size using the desired step. Next, we use the sequence function. The sequence function populates the device_vector with values from 0 to N-1 in sequential order.

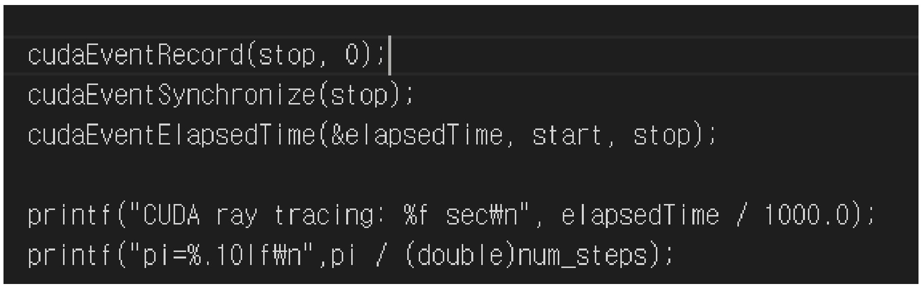


```
double pi = cal_pie(D);
```



```
struct cal {
  double x;
  __device__ __host__ double operator()(int i) { // float에서 double로 수정
    double step = 1.0 / (double) num_steps;
    x = (i+0.5) * step;
    return 4.0/(1.0+x*x);
  }
};

float cal_pie(const thrust::device_vector<int> &x){
   size_t N = x.size();
   thrust::device_vector<float> vec_temp(N);
   thrust::transform(x.begin(), x.end(), vec_temp.begin(), cal());
   return thrust::reduce(vec_temp.begin(), vec_temp.end());
}
```

transform is a convenient Thrust function that allows us to apply a function f(x) to each element x in a vector, resulting in a transformed vector. After transforming the vector elements as desired, we can use reduce to sum all the elements in the vector.



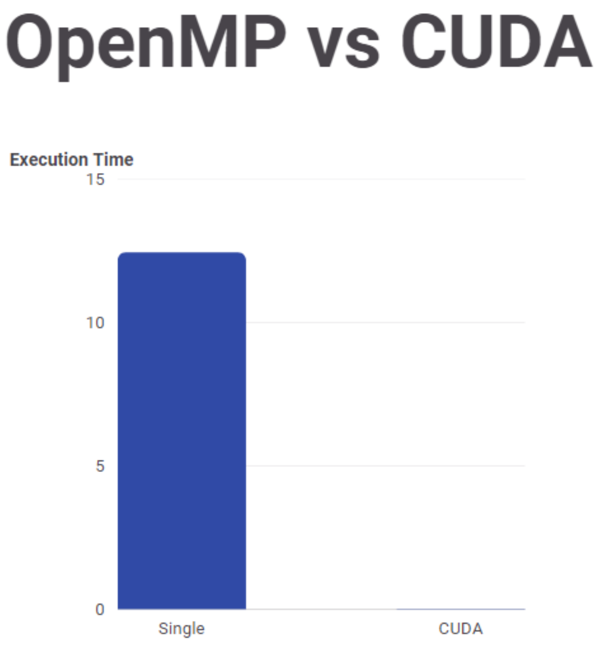
```
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsedTime, start, stop);

printf("CUDA ray tracing: %f sec₩n", elapsedTime / 1000.0);
printf("pi=%.10lf₩n",pi / (double)num_steps);
```

After performing the reduction operation using thrust::reduce and obtaining the sum of the transformed vector, dividing this sum by num_steps will yield an approximation of the mathematical constant π (pi).

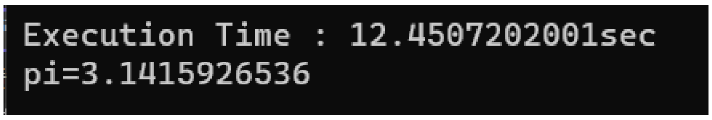
4. Analysis

	CUDA	Single Thread
Execution Time	0.377635	12.4507202001

<Table>

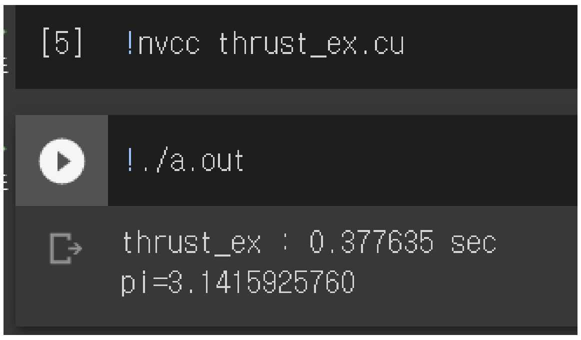


<Graph>



```
Execution Time : 12.4507202001sec
pi=3.1415926536
```

<Single Thread>



```
[5] !nvcc thrust_ex.cu
```

```
!./a.out
```

```
thrust_ex : 0.377635 sec
pi=3.1415925760
```

<CUDA result>

Calculating π (pie) is a problem I have worked on using pthread, OpenMP, and Java multithreading. Load balancing, achieved by distributing the work among threads using loops, was crucial in those implementations. However, with Thrust and CUDA, the implementation becomes much more convenient and the performance is excellent.

Unlike a single thread, CUDA leverages its parallel processing capabilities to efficiently handle large-scale data and complex calculations. Moreover, GPUs are inherently SIMD (Single Instruction, Multiple Data) devices, and the problem of calculating π can be seen as a suitable SIMD problem since each element performs the same operation.