Multicore Computing Assignment 4

Joshua BRIONNE - 50241647

🧰 Compilation & Execution

Compilation has been performed on **Ubuntu 22.04.3 LTS (Linux)**.

---

1. Environment

Property	**Value**
OS Name	Linux
OS Version	6.8.0-52-generic
Architecture	amd64
Available processors (cores)	8
Max memory (MB)	3936
Thrust Environment	Google Colab

---

2. Compilation

2.1 Thrust

```
nvcc -arch=sm_75 thrust_ex.cu

./a.out
```

⏱️ Execution Time Table (in seconds)

Reference

```
Execution Time : 6.8606073940sec

pi=3.1415926536
```

Using Thrust

Type	**Time (s)**
Sequential (one thread)	6.860
Thrust	0.412

3. Performance Analysis

3.1 Sequential Single-Threaded

The original version of the program performs a straightforward loop over 1 billion iterations to approximate π using numerical integration. All computations are done on a single CPU thread.

- **Execution time**: 6.86 seconds
- **Accuracy**: pi ≈ 3.1415926536
- **Downside**: No parallelism at all — this version is entirely CPU-bound, and only uses a single core, making it inefficient for high workloads.

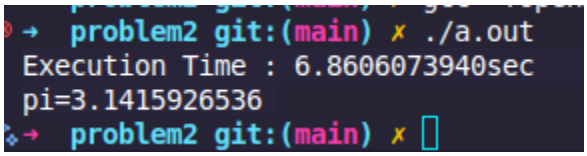
3.2 Thrust (CUDA Accelerated)

The Thrust-based version leverages CUDA to run the same computation on the GPU, distributing the work across thousands of threads.

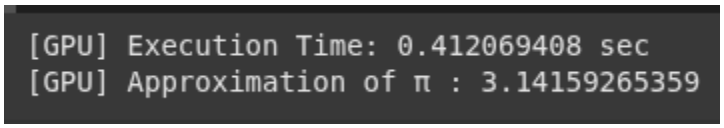
- **Execution time**: 0.412 seconds
- **Same accuracy** as the sequential version.
- **Speed-up**: ~16× faster than the CPU version.
- **Why it's faster**: The computation of each iteration is independent, making it a perfect candidate for **data-parallel** execution. Thrust handles GPU resource management under the hood, offering a high-level API to achieve massive parallelism with little effort when the workload is suitable, meaning it can efficiently utilize the GPU's many cores.

4. Screenshots

4.1 Compilation Output - Mono Threaded



4.2 Compilation Output - Thrust



5. Source Code

```
// %%writefile thrust_ex.cu
#include <thrust/count.h>
#include <thrust/transform_reduce.h>
#include <thrust/execution_policy.h>
#include <iostream>
#include <chrono>


struct BerthelotPiKernel
{
   double dx;

   __host__ __device__
   BerthelotPiKernel(double interval) : dx(interval) {}

   __host__ __device__
   double operator()(long long idx) const
   {
       double midpoint = (idx + 0.5) * dx;
       return 4.0 / (1.0 + midpoint * midpoint);
   }
};

int main()
{
   constexpr long long totalSlices = 1'000'000'000LL;
   const double delta = 1.0 / static_cast<double>(totalSlices);

   // We start the timing refrences here
   auto tic = std::chrono::high_resolution_clock::now();

   // reducing the pi calculus
   double sum = thrust::transform_reduce(
       thrust::device,
       thrust::make_counting_iterator(0LL),
       thrust::make_counting_iterator(totalSlices),
       BerthelotPiKernel(delta),
       0.0,
       thrust::plus<double>()
   );

   double piApprox = sum * delta;

   auto toc = std::chrono::high_resolution_clock::now();
   double duration = std::chrono::duration<double>(toc - tic).count();

   // logging our results :)
   std::cout.precision(12);
   std::cout << "[GPU] Execution Time: " << duration << " sec\n";
   std::cout << "[GPU] Approximation of π : " << piApprox << "\n";

   return 0;
}
```