Estimating π Value — Report

Ranathunga N. D.
200517U

February 25, 2025

Contents

1 Monte Carlo Method: Approach and Implementation

The Monte Carlo method for estimating π relies on sampling random points in $[-1, 1] \times [-1, 1]$ and counting how many lie within the inscribed quarter circle:

$$\pi \approx 4 \times \frac{\text{Number of points inside the circle}}{\text{Total number of points}}.$$

A single-threaded approach is excluded due to its long runtime for large sample sizes. Instead, multi-threaded (pthreads, OpenMP) and GPU-accelerated (CUDA) implementations are used. The sample size is determined using a **binomial** model.

1.1 Determining Sample Size Using a Binomial Model

- **Variance Bound:** Each point can be viewed as a Bernoulli trial (success if $\sqrt{x^2 + y^2} \leq 1$), but the total number of successes follows a binomial distribution with worst-case variance $p(1 - p)$ with $p = 0.5$ yields $\sigma^2_{\max} = 0.25$.
- **Samples for Accuracy:**

  $$N \geq \frac{4}{\varepsilon^2}.$$

  For $\varepsilon = 10^{-5}$, $N \approx 4 \times 10^{10}$. For $\varepsilon = 10^{-10}$, $N \approx 4 \times 10^{20}$.

1.1.1 Practical Limit and Diminishing Returns

Statistical Nature of the Estimator. When π is estimated by randomly sampling points in a unit square, the estimator $\hat{\pi} = 4 \times (\text{points in circle}/N)$ is unbiased but has a variance that scales as $1/N$. By the central limit theorem, its standard error decreases proportionally to $1/\sqrt{N}$.

Diminishing Returns on Accuracy. Because the error decreases as $1/\sqrt{N}$:

- Doubling N only reduces the error by about $1/\sqrt{2} \approx 29\%$.
- Each additional decimal place of precision requires roughly 100 times more samples.

Inherent Random Variance. Random fluctuations remain even for large N, diminishing only at the rate of $1/\sqrt{N}$. Practical constraints on pseudo-random generation and the rapid growth in required samples justify fixing the total sample size at 4×10^{10} for the experiments under Question 1.

1.2 Task Decomposition and Random Number Generation

- **Common Strategy:** The total sample size (4×10^{10}) is divided into smaller chunks; each thread (or GPU thread) processes one or more chunks. Local counters are maintained to minimize synchronization.
- **Random Numbers:**
  - `std::mt19937` (Mersenne Twister) is used with `std::uniform_real_distribution` over $[-1, 1]$.
  - Distinct seeds are assigned to each thread to reduce correlation.
- **Data Types:**
  - `unsigned long long` is used for counting inside-circle samples.
  - `long double` is used for storing and returning the final π value.
- **Thread/Block Configuration:**
  - **CPU Threads:** Up to 12 threads are used on the CPU, as specified by a configuration file.
  - **GPU Threads:** In CUDA, a one-dimensional grid is launched with `threadsPerBlock` = 1024 and

    $$\texttt{blocks} = \frac{\texttt{threadCount} + \texttt{threadsPerBlock} - 1}{\texttt{threadsPerBlock}},$$

    where `threadCount` = 40000.

1.3 Parallel Implementations: Overview

1.3.1 Pthreads-Based Implementation

- **Thread Creation:** `pthread_create` spawns a fixed number of threads, each assigned one or more chunks.

- **Local Counting:** Within each chunk, the condition $(x^2 + y^2) \leq 1$ is checked. A local counter is incremented whenever a point lies inside the circle.
- **Global Summation:** `pthread_join` is used to wait for all threads; local counts are then summed to compute the final estimate of π.

1.3.2 OpenMP-Based Implementation

- **Parallel Loop:** A `#pragma omp parallel for` distributes chunks among OpenMP threads.
- **Reduction:** A `reduction(+ : totalInsideCount)` clause accumulates local counters into a global variable.
- **Thread Seeding:** Each thread is assigned a seed offset via `omp_get_thread_num()`.

1.3.3 CUDA-Based Implementation

- **Kernel Launch:** A kernel (`MCKernel`) is invoked with the chosen block and thread configuration. Each GPU thread processes a portion of the samples.
- **curand_init:** Each thread initializes a local PRNG state, generating points in $[-1, 1] \times [-1, 1]$.
- **Atomic Accumulation:** Each thread's local count is added to a global counter using `atomicAdd`. The final count is copied back to the CPU.

1.4 Hardware Specifications

1.5 Implementation Details

- **Time Measurement:** The total runtime is measured using `std::chrono::high_resolution_clock`. The start time is recorded immediately before the estimation begins, and the end time is recorded after all threads (or GPU kernels) have completed.

CPU Model	Intel (R) Core (TM) i7–9750H CPU @ 2.60GHz
Cores/Threads	6 Cores / 12 Threads
RAM	16 GiB DDR4
GPU Model	NVIDIA GeForce (CUDA Version 11.4)
GPU Memory	3911 MiB

Table 1: Summary of CPU and GPU hardware used in the experiments.

- **Compilation:** A `CMakeLists.txt` file and a Makefile are used to compile and run the code, though these details do not affect the methodology directly.

2 Part A: Experimental Tasks and Results

2.1 Task A.1: Evaluating π to Different Decimal Places

Table 2: Precision-based results for Pthreads

Precision	Pi Estimate	Time (s)
5	3.14159	233.626
10	3.1415886455	238.334
15	3.141602155100000	232.330
20	3.14158805580000000004	231.193

Table 3: Precision-based results for OpenMP

Precision	Pi Estimate	Time (s)
5	3.14159	244.388
10	3.1415845871	249.834
15	3.141601863300000	242.200
20	3.14158336399999999996	239.260

Table 4: Precision-based results for CUDA

Precision	Pi Estimate	Time (s)
5	3.14158	9.036
10	3.1415923941	8.762
15	3.141592781500000	8.786
20	3.14158480229999999994	8.786

2.2 Task A.2: Reassessing Strategies for Specific Trial Counts

Table 5: 2^n-based results for Pthreads

Experiment	Total Samples	Pi Estimate	Time (s)	Error
2^{24} trials	16777216	3.14161205291748046875	0.094	0.000019
2^{26} trials	67108864	3.14167696237564086914	0.520	0.000084
2^{28} trials	268435456	3.14164182543754577637	1.794	0.000049

Table 6: 2^n-based results for OpenMP

Experiment	Total Samples	Pi Estimate	Time (s)	Error
2^{24} trials	16777216	3.14237880706787109375	0.083	0.000786
2^{26} trials	67108864	3.14174300432205200195	0.450	0.000150
2^{28} trials	268435456	3.14153531193733215332	1.867	0.000057

Table 7: 2^n-based results for CUDA

Experiment	Total Samples	Pi Estimate	Time (s)	Error
2^{24} trials	16777216	3.13798832893371582031	0.004	0.003604
2^{26} trials	67108864	3.14036542177200317383	0.016	0.001227
2^{28} trials	268435456	3.14094561338424682617	0.060	0.000647

2.3 Task A.3: Profiling the Best-Performing Program

After running the experiments, the GPU-based Monte Carlo approach was identified as the best performer for large-scale sampling. Profiling was performed using Nsight Compute (`ncu`), and the following key observations were made:

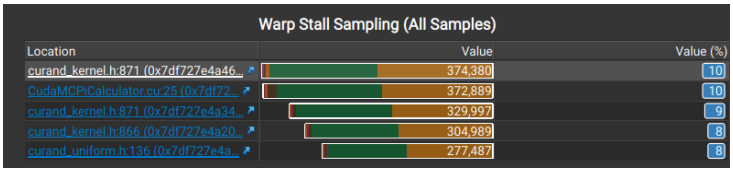

Figure 1: Warp Stall Sampling view

Warp Stall Sampling. Figure 1 shows that functions in `curand_kernel.h` account for significant warp stalls (10% each for two instances), with `CudaMCPiCalculator.cu` also registering around 8%–9% of stalls. These stalls are primarily related to random number generation and the `if (x * x + y * y <= 1.0f)` check within the kernel.

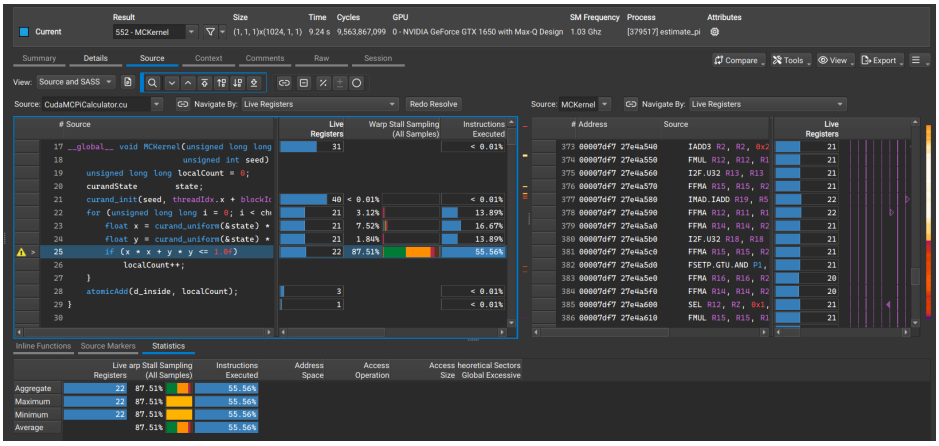

Figure 2: Source and SASS Analysis

Instruction-Level Analysis. Figure 2 indicates that the conditional check `if (x * x + y * y <= 1.0f)` causes a significant share of warp stalls. In a GPU's SIMT (Single Instruction, Multiple Threads) architecture, threads are grouped into warps that execute the same instruction in lockstep. When a branching instruction (like an `if`-statement) occurs, some threads may take one path while others take another, leading to *divergence*. Divergent warps must serialize the execution of each path, which stalls threads on the inactive path. This serialization can reduce overall throughput, especially when the conditional is evaluated many times per thread. Although the dis-

tance check is essential for the Monte Carlo approach, reducing divergence can mitigate these stalls.

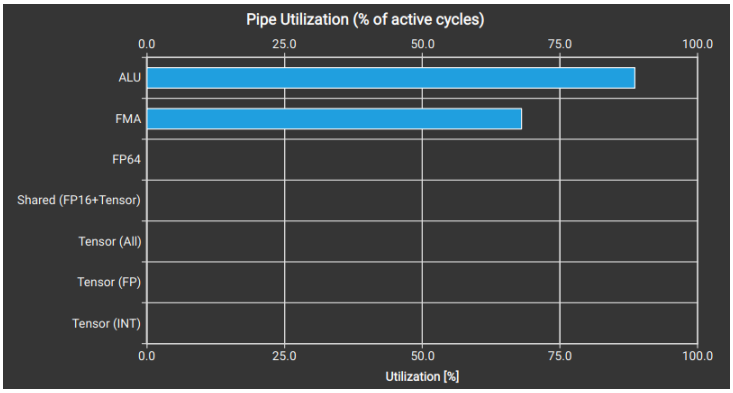

Figure 3: Pipe utilization

Pipe Utilization. Figure 3 presents the pipe utilization. The ALU is utilized most heavily, followed by the FMA (fused multiply-add) units. This is expected given the arithmetic nature of the distance check ($x^2 + y^2$) and other basic floating-point operations.

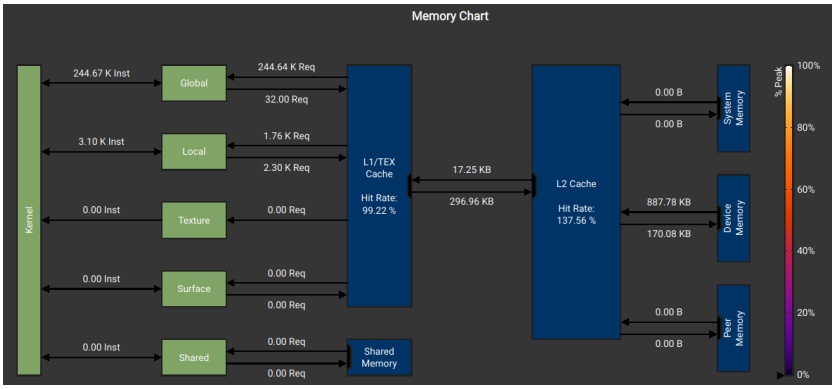

Figure 4: Memory chart

Memory Chart. Figure 4 shows the memory chart. Global memory requests are relatively small, since only a single atomic addition is performed

per thread block (to accumulate the inside-circle count). Consequently, the memory subsystem does not appear to be a major bottleneck. L2 cache hit rates are high, and only minimal data is transferred between host and device, so bandwidth is not saturated.

Potential Optimizations.

- **Random Number Generation Bottleneck:** The `curand_kernel.h` calls introduce warp stalls. More efficient RNG strategies or caching random values in shared memory could reduce this overhead.
- **Conditional Check Overhead:** The `if` $(x^2 + y^2 \leq 1.0f)$ line is repeatedly executed. While unavoidable, exploring warp-level primitives or reducing divergence may help.
- **Memory System:** Memory bandwidth is not saturated, so optimizing global memory usage is less critical. However, a shared-memory reduction might still reduce atomic overhead.

2.4 Task A.4: Discussion of Findings and Limitations

Accuracy. Monte Carlo methods are probabilistic; achieving precision beyond a few decimal places quickly becomes expensive due to the $1/\sqrt{N}$ error scaling. Each additional digit of accuracy necessitates roughly 100 times more samples.

Profiling Insights. Profiling the GPU code revealed that:

- `curand_kernel.h` introduces significant warp stalls due to random number generation.
- Repeated distance checks ($x^2 + y^2 \leq 1$) also stall warps, though they are unavoidable for this approach.
- Memory usage is minimal, so bandwidth is not a bottleneck.

Limitations.

- **Randomness Quality:** The estimator depends on good pseudorandom generation.
- **Complexity:** Threading libraries and CUDA kernels add development overhead.

3 Part B: Gregory-Leibniz Series for π

The Gregory-Leibniz series for π is given by:

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}.$$

3.1 Implementation and Observations

- **Implementation**: A dynamic approach was used, where the kernel-based summation accumulates terms until a specified decimal precision is reached. This eliminates random variance but requires explicit tracking of convergence.
- **Convergence**: The series converges relatively slowly compared to some other infinite series for π. However, it still tends to require fewer terms for a given precision than a Monte Carlo approach needs samples, especially as precision grows.
- **Comparison to Monte Carlo**:
  - **Monte Carlo's High Sample Requirements**: Achieving high-precision estimates with Monte Carlo generally demands an exponentially larger number of random samples, leading to slow convergence for many-decimal-place accuracy.
  - **Parallelization Ease**: Monte Carlo is straightforward to parallelize (each sample is independent), whereas Gregory-Leibniz requires careful summation of partial results. Nonetheless, both can benefit from multi-threading or GPU kernels.
  - **Deterministic vs. Stochastic**: Gregory-Leibniz produces deterministic results without random variance, whereas Monte Carlo is inherently probabilistic.

3.2 Experimental Results and Comparison

Table 8: Gregory-Leibniz GPU Results (Dynamically Terminated)

Precision	Pi Estimate	Time (s)
5	3.14159	1.765
6	3.141592	3.934
7	3.1415924	11.876
8	3.14159255	36.433
9	3.141592622	120.241
10	3.1415926436	394.454

Time Comparison with Monte Carlo.

- **Precision vs. Performance**: The Monte Carlo approach requires a rapidly growing number of samples to reach higher decimal places, often making it slower than Gregory-Leibniz for equivalent precision. By contrast, the Gregory-Leibniz series, though still slow, can achieve a target precision with fewer total operations than a naive Monte Carlo at very high decimal places.
- **Parallel Scalability**: Monte Carlo is conceptually simpler to parallelize (each sample is independent), but the overhead of generating large numbers of samples can still be high. Gregory-Leibniz parallelization involves summing partial results, yet each iteration is deterministic and free of variance.

Conclusion

In summary, the Monte Carlo and Gregory-Leibniz methods each provide valid approaches to estimating π. Monte Carlo is conceptually simple to parallelize but requires a rapidly growing number of samples for higher precision. The Gregory-Leibniz series converges deterministically, often requiring fewer total iterations at very high precision, though it still grows slowly and demands careful summation for parallel performance. Ultimately, the choice depends on available hardware, desired accuracy, and implementation complexity.