CS6640: Advanced Computer Architecture

GPU-Accelerated Monte Carlo Simulations

Programming Assignment #1 Report

Harikrishnan KV CS24S030

Contents

1 Summary

The study of probability and statistics often leads to intriguing problems that can be explored through simulation. One such classic problem is Buffon's Needle, which estimates the value of π through the random dropping of a needle onto a floor marked with parallel lines. This problem not only serves as an engaging mathematical challenge but also highlights the principles of geometric probability.

In recent years, the advent of parallel computing has significantly enhanced the efficiency and scalability of numerical simulations. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) developed by NVIDIA, enabling developers to leverage the power of GPUs for general-purpose computing. By offloading computationally intensive tasks to the GPU, simulations can be executed much faster compared to traditional CPU-based methods.

This report presents a detailed exploration of Buffon's Needle simulation using CUDA. We will delve into the mathematical formulation of the problem, outline the parallel algorithm implemented on the GPU, and analyze the results obtained from the simulation.

2 Introduction

Preliminaries

Monte Carlo methods generally follow these steps:

1. Define input domain
2. Generate inputs randomly
3. Perform deterministic computations
4. Aggregate results.

Setup

1. A floor has parallel lines spaced ℓ apart.
2. A needle of length $L < \ell$ is randomly dropped.
3. The probability of intersection is estimated.

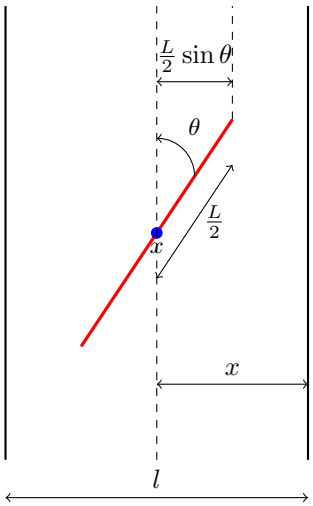


Figure 1: Needle crossing a vertical line

2.1 Mathematical Formulation

A needle's position is (x, y) with orientation θ. Due to symmetry, we analyze $0 \le x \le \ell/2$. The Probability Density Functions:

$$f_X(x) = \begin{cases} 2/\ell, & 0 \le x \le \ell/2 \\ 0, & \text{elsewhere} \end{cases}, \quad f_\Theta(\theta) = \begin{cases} 2/\pi, & 0 \le \theta \le \pi/2 \\ 0, & \text{elsewhere} \end{cases}$$

Since x and θ are independent:

$$f_{X,\Theta}(x,\theta) = f_X(x) f_\Theta(\theta) = \begin{cases} \frac{4}{\ell\pi}, & 0 \le x \le \ell/2, 0 \le \theta \le \pi/2 \\ 0, & \text{elsewhere} \end{cases}$$

Needle crosses a line if $x \le \frac{L}{2}\sin\theta$. The probability is:

$$P(\text{cross}) = \int_0^{\pi/2} \int_0^{(L/2)\sin\theta} \frac{4}{\ell\pi} dx d\theta = \int_0^{\pi/2} \frac{4}{\ell\pi} \cdot \frac{L}{2} \sin\theta d\theta = \frac{2L}{\ell\pi} \left[-\cos\theta\right]_0^{\pi/2} = \frac{2L}{\ell\pi}(1) = \frac{2L}{\ell\pi}$$

Using $\ell = 2L$, we get $\mathbf{P}(\text{cross}) = \frac{1}{\pi}$, enabling easier π estimation.

3 Programming Language, Hardware Details, and Libraries

The simulation is implemented using the CUDA programming model, which allows for efficient parallel processing on NVIDIA GPUs. The programming language used is C/C++ with CUDA extensions.

The simulations were executed on an NVIDIA Tesla T4 GPU, available on Kaggle's cloud environment. The key specifications of the Tesla T4 GPU are:

- **Architecture:** Turing
- **CUDA Cores:** 2560
- **Memory:** 16GB GDDR6
- **Peak FP32 Performance:** 8.1 TFLOPS
- **Compute Capability:** 7.5

For random number generation, we utilize the cuRAND library, which is optimized for CUDA to provide fast and efficient random number generation.

4 Simulation Design

4.1 Serial Approach

The serial implementation follows a simple iterative method:

1. Generate a random needle position and angle.
2. Check if the needle crosses a line.
3. Repeat for N trials.
4. Compute π using the probability estimate.

4.2 Parallel Approach

GPU Parallelization Strategy

- Each thread simulates multiple needle drops independently.
- Random numbers are generated using the cuRAND library.
- The results from all threads are aggregated.
- The final probability estimate is used to compute π.
- **Avoidance of Atomics and Shared Memory:** Atomic operations introduce synchronization overhead, and shared memory is unnecessary since each thread's work is independent. Instead, the simulation is executed in batches, utilizing an optimal number of threads to balance parallel efficiency and resource utilization.
- The block size is set to $8 \times 8 \times 16$, ensuring it remains within the hardware limit of 1024 threads per block.

5 Implementation details

5.1 Parallel Implementation

The parallel simulation of needle drops utilizes the following functions:

```
long int parallelSimulationRunner(long int simulationCount);
```

This function orchestrates the parallel simulation process. It manages GPU memory allocation, executes simulations in batches, and accumulates results until the specified number of simulations is completed.

```
__global__ void parallelNeedleSimulation(double needleLength, int *gpuResultArray, long int experiments
```

This kernel is executed on the GPU, where each thread independently performs the needle simulation using the `simulate` function. It checks whether the needle crosses a line and records the result in the `gpuResultArray`.

```
__global__ void serialSum(int *gpuResultArray, long int *localDSum, long int currentBatchSize);
```

This kernel aggregates the results from the `gpuResultArray` into a single value stored in `localDSum`, representing the total number of needle crossings for the current batch of simulations.

```
__device__ int simulate(long int seed, double needleLength);
```

This device function executes a single needle drop simulation, returning a boolean value indicating whether the needle crosses a line. It uses random number generation to determine the position and angle of the needle.

Note: The `simulate()` function is the same in both the parallel and serial implementations.

5.2 Serial Implementation

The serial version of the simulation comprises the following function:

```
__global__ long int serialSimulationRunner(long int simulationCount);
```

This function performs the needle drop simulation sequentially, iterating through the specified number of simulations and summing the results directly in the device memory, then copying them back to the host.

```
__global__ int simulate(long int seed, double needleLength);
```

This function conducts a single needle drop simulation in a serial manner, similar to the device version. It determines if the needle crosses a line and returns the result accordingly.

Note: The `simulate()` function is the same in both the parallel and serial implementations.

6 Performance Analysis

6.1 Finding Optimal Block Configuration

We explored various block and grid configurations to identify the optimal settings for execution time. The configurations tested, along with their corresponding execution times, are summarized in Table 1.

Table 1: Execution Times for Different Block Configurations($n = 1000000$)

Configuration	Total Run Time (s)	CUDA Run Time (s)
1024, 1, 1	0.402758	0.268937
512, 2, 1	0.121132	0.121230
256, 4, 1	0.117798	0.117857
128, 8, 1	0.112331	0.112308
64, 16, 1	0.115334	0.115304
32, 32, 1	0.117768	0.117747
16, 64, 1	0.111630	0.111671
8, 8, 16	0.114281	0.114304
16, 4, 16	0.114559	0.114529
4, 4, 64	0.115653	0.115612

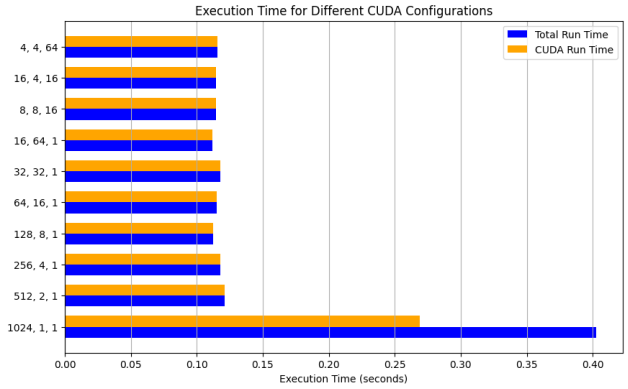


Figure 2: Runtimes for different Block sizes

The analysis of the execution times indicates that the configuration yielding the lowest execution times was **128, 8, 1** for the first run time and **16, 64, 1** for the second run time. The slowest configuration being (1024,1,1) was consistent for other values of n too.

The results demonstrate that reducing the block size to 128 threads while increasing the number of blocks to 8 resulted in significant performance improvements.

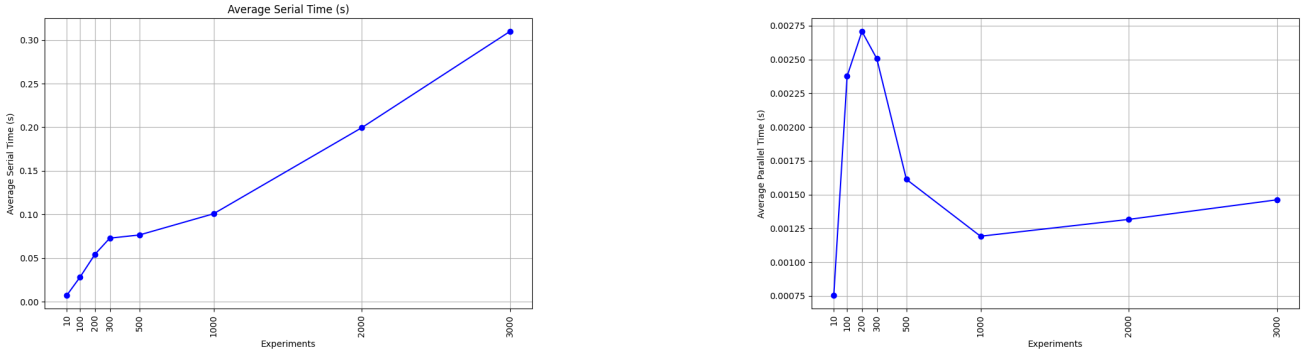
The performance of the Buffon's Needle simulation is evaluated based on three key parameters: execution time, accuracy, and speedup.

Each experiment was repeated for k =**10 epochs**, and the reported parameters are the average across these runs.

6.2 Execution Time

The execution times for both the serial and parallel implementations were recorded for different values of N. The serial implementation executes each experiment sequentially, leading to longer execution times, whereas the parallel implementation distributes computations across multiple GPU threads, resulting in significantly faster execution.

We can also see that in the parallel execution, until a specific value of n, is having more overhead distributing tasks, rather than doing a computation. After $n = 1000$, we can see that the overhead becomes less than the computation time.



(a) Serial Execution Time

(b) Parallel Execution Time

Figure 3: Performance comparison between serial and parallel implementations

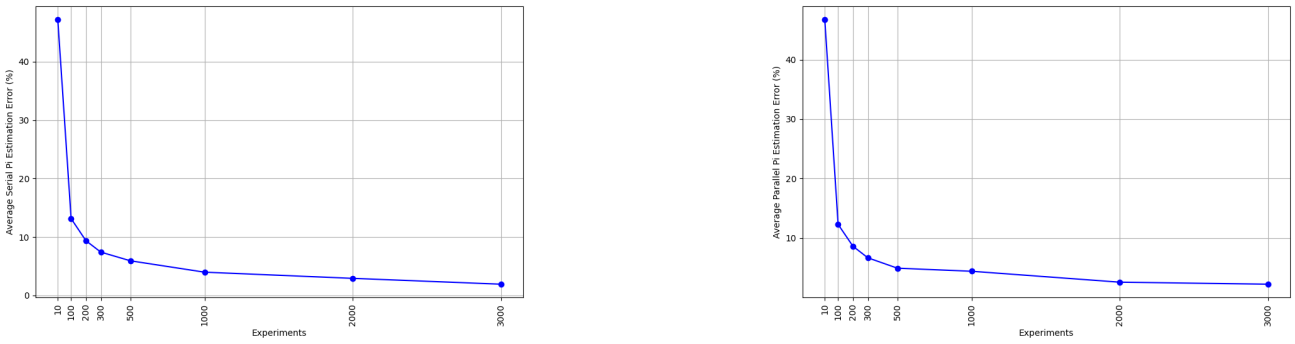
6.3 Accuracy

The accuracy of the simulation is determined by comparing the estimated value of π to its actual value (≈ 3.1415926535). Due to the stochastic nature of the Monte Carlo method, small sample sizes produce large deviations from the true value. As N increases, the law of large numbers ensures convergence towards the true value of π. The percentage error, given by:

$$\text{Percentage Error} = \left| \frac{\pi_{\text{estimated}} - \pi}{\pi} \right| \times 100, \tag{1}$$

decreases as N grows.

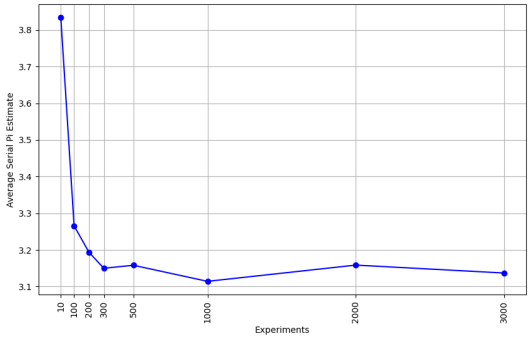
We can see that the values converge to actual value π, as n increases, in both the serial and parallel versions.



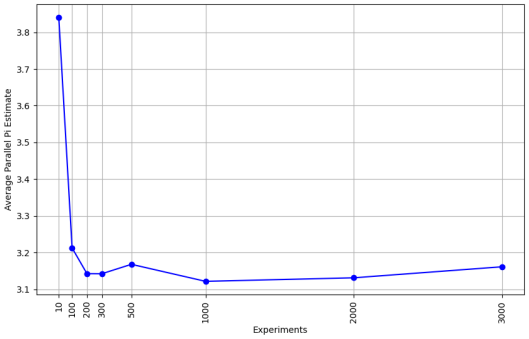
(a) Serial Execution

(b) Parallel Execution

Figure 4: Performance comparison between serial and parallel implementations



(a) Serial π Estimate



(b) Parallel π Estimate

Figure 5: π Estimate between serial and parallel implementations

6.4 Speedup

Speedup is computed as:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}. \tag{2}$$

For small values of N, GPU kernel launch overhead affects performance, leading to lower speedup. However, as N increases, the parallel efficiency improves, and the speedup reaches over $200\times$ for large N, demonstrating the scalability of the parallel implementation.

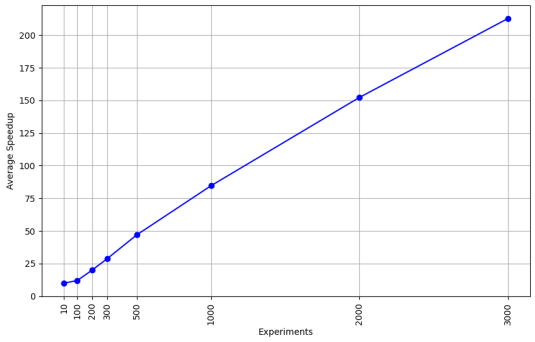


Figure 6: Speedup

6.5 Observations

- The estimated value of π improves with larger N due to statistical averaging.
- The parallel CUDA implementation significantly outperforms the serial version as N increases.
- The speedup is moderate for small N but becomes substantial for large N due to better GPU resource utilization.

7 Instructions on Running the Code

This section provides a step-by-step guide for compiling and executing the Buffon's Needle simulation, along with software dependencies and performance measurement methods.

7.1 Software Dependencies

Ensure the following software and libraries are installed before execution:

- **CUDA Toolkit** (for GPU execution)
- **NVIDIA Drivers** (for GPU access)
- **Python + Matplotlib** (for generating performance graphs)

On a Linux system(Debian based), install the following dependencies:

```
sudo apt update
sudo apt install python3 python3-pip bc
pip install matplotlib numpy
```

7.2 Compiling the Code

7.2.1 Simulation Runner

To compile the simulation Runner:

```
nvcc -arch=sm_75 -o simulation simulation.cu
```

7.3 Running the Simulation

```
./runner.sh
```

The file `graphing_helper.sh` has a variable `n_values`. This variable is an array of values of n, for which times the simulation should run, and the value k indicates the epochs for each n, and the output graph is an average over this weighted k.

7.4 Generating Performance Graphs

After execution, a Python script is used to visualize speedup, accuracy, and error and CUDA device only runtimes:

```
python generate_plots.py
```

This script generates plots showing execution time, speedup, and error rates.

7.5 Running on Kaggle with a T4 GPU

If running on Kaggle, enable the **GPU accelerator** in the notebook settings. The code can be executed within a Jupyter Notebook with:

```
!nvcc --version  # Verify CUDA installation
!nvcc -arch=sm_75 -o simulation simulation.cu
!bash runner.sh
!python generate_plots.py
```

8 Conclusion

In this report, we explored the parallelization of Buffon's Needle problem using CUDA to leverage GPU acceleration. The key takeaways from our analysis are as follows:

- **Speedup:** The parallel implementation achieved significant speedup over the serial version, particularly as the number of simulations increased. We were able to see as much as a 200 speedup, on an input of size just 3000. The results show that GPU acceleration provides a substantial performance boost, making Monte Carlo simulations more efficient for large-scale problems.
- **Accuracy:** Both serial and parallel implementations converge toward the expected value of π. However, the estimation error decreases with larger experiment sizes, as expected in Monte Carlo methods. The parallel implementation maintains accuracy comparable to the serial approach, which is the expected behaviour, as the tosses are independent of each other.
- **Efficiency Considerations:** The implementation avoids atomic operations and shared memory to reduce overhead. Instead, a batched execution strategy was used, ensuring efficient utilization of GPU threads while keeping memory access overhead minimal.

9 References

[1] Wikipedia, "Buffon's needle." Available at: `https://en.wikipedia.org/wiki/Buffon%27s_needle`.

[2] NVIDIA, "CUDA Toolkit Documentation." Available at: `https://docs.nvidia.com/cuda/`.

[3] NVIDIA, "CUDA C Programming Guide." Available at: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[4] D. Bertsekas and J. Tsitsiklis, *Introduction to Probability*, 2nd ed., Athena Scientific, 2008. ISBN: 9781886529236. Available at: `http://www.athenasc.com/probbook.html`.