WIDS Assignment 1: Analysis of GPU-Accelerable Workload
Monte Carlo Simulation for Estimating π

Matam Kushaal
23B1290

14th December 2025

1 Operation Breakdown

Brief Description

Monte Carlo methods form a cornerstone of computational science, enabling statistical estimation of complex quantities through random sampling. Estimating π via Monte Carlo integration serves as a canonical example due to its simplicity and clear parallel structure. This workload exemplifies parallel computation, making it exceptionally well-suited for GPU acceleration.

1.1 Computational Objective

The algorithm estimates π by leveraging the area relationship between a unit circle and its bounding square. Random points uniformly distributed within the square are sampled, and the proportion falling inside the circle provides an estimate of $\pi/4$. As sample count increases, the estimate converges to the true value according to the law of large numbers.

1.2 Mathematical Formulation

Given N independent random samples where each point (x_i, y_i) is drawn from a uniform distribution over $[-1, 1]^2$, the indicator function for each sample is:

$$I_i = \begin{cases} 1 & \text{if } x_i^2 + y_i^2 \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

The Monte Carlo estimator for π is:

$$\pi_{\text{est}} = \frac{4}{N} \sum_{i=1}^{N} I_i$$

1.3 Algorithmic Structure

```
count_inside = 0
for i in range(num_samples):
    x = random_uniform(-1, 1)
    y = random_uniform(-1, 1)
    if (x*x + y*y <= 1.0):
        count_inside += 1
pi_estimate = 4.0 * count_inside / num_samples
```

Listing 1: Basic Monte Carlo π Estimation Algorithm

1.4 Workload Characteristics

- **Per-Iteration Cost**: 2 random number generations, 3 floating-point operations, 1 comparison
- **Memory Requirements**: Minimal: only final count requires storage (also if seeds are used(for reproducibility) then input may require the seed (which maybe float or int type))

1.5 Parallelism Opportunities

This represents an ideal **parallel** workload:

- **Sample-level independence**: Each random trial is completely autonomous and hence can be done independent of one another
- **No data dependencies**: No communication needed between processing elements
- **Uniform workload**: All iterations perform identical operations

2 Compute vs Memory Analysis

2.1 Compute-Bound Nature

Monte Carlo π estimation is **fundamentally compute-bound** due to:

- High ratio of arithmetic operations to memory accesses
- Minimal data movement requirements (only input: random seeds; output: single counter)
- Dominant cost being random number generation and arithmetic
- Memory bandwidth utilization being negligible compared to computational demand

2.2 Arithmetic Intensity Profile

- **High arithmetic to memory ratio**: Each sample requires significantly more floating-point operations than memory accesses
- **Minimal memory footprint**: Only aggregated results require storage, with negligible per-sample memory overhead

- **Favorable intensity characteristic**: The workload exhibits high arithmetic intensity due to minimal data movement requirements

2.3 Data Reuse Characteristics

- **No temporal locality**: Each sample uses unique random numbers
- **No spatial locality**: Samples are independent with no shared data
- **Memory hierarchy impact**: Cache provides minimal benefit; registers suffice for per-thread state
- **Optimization focus**: Efficient RNG and reduction, not data reuse

2.4 Dependencies and Sequential Constraints

- **Parallel region**: Entire sampling loop is parallelizable
- **Sequential component**: Final reduction of counts across all threads
- **Synchronization**: Only required at reduction phase
- **Control flow**: Minimal branching (only one condition check)

3 Expected Behavior on a GPU

3.1 CUDA Execution Model Mapping

- **Thread-to-sample mapping**: Each thread processes multiple independent samples, typically assigned via contiguous blocks of samples per thread
- **Grid organization**: Grid dimension = $\lceil N/(\text{threads\_per\_block} \times \text{samples\_per\_thread}) \rceil$
- **Block granularity**: Blocks process contiguous chunks of the sample space
- **Warp behavior**: All threads execute identical control flow, minimizing divergence

3.2 Scalability Prospects

- **Theoretical scaling**: Linear speedup with additional compute units
- **Practical limits**: Only bounded by GPU thread capacity and memory for reduction
- **Utilization efficiency**: Can achieve near-perfect utilization with sufficient samples
- **Strong scaling**: Excellent due to minimal inter-thread communication

3.3 Potential Implementation Challenges

1. **Parallel Random Number Generation**
   - Requirement for statistically independent streams across thousands of threads
   - Need for high-quality, high-performance parallel RNG (random number generation)
   - Memory overhead for storing RNG states
2. **Efficient Reduction**
   - Combining partial counts from all threads with minimal overhead
   - Avoiding atomic operation bottlenecks
   - Hierarchical reduction strategies
3. **Precision Considerations**
   - Single-precision accumulation sufficient for moderate N
   - Double-precision needed for very large N to avoid rounding errors
   - Trade-off between precision and performance
4. **Load Balancing**
   - Inherently perfect due to uniform work distribution
   - Only potential imbalance from thread divergence (minimal)

3.4 Optimization Opportunities

- **Block-level aggregation**: Partial reductions within blocks before global reduction
- **RNG optimization**: Using on-chip PRNGs or optimized library functions
- **Instruction-level parallelism**: Loop unrolling and ILP exploitation
- **Memory access patterns**: Coalesced writes for partial results

Conclusion

The Monte Carlo π estimation workload represents an ideal candidate for GPU acceleration due to its highly parallel nature, high arithmetic intensity, and minimal memory requirements. Its computational pattern maps naturally to the CUDA execution model, with each thread independently processing random samples and only requiring synchronization during the final reduction phase. While challenges exist in parallel random number generation and efficient reduction, these are well-studied problems with established solutions. The workload's simplicity combined with its computational intensity makes it an excellent pedagogical example for demonstrating GPU programming principles while achieving substantial performance gains over CPU implementations. This CPU baseline implementation establishes a clear performance reference for evaluating the effectiveness of GPU acceleration in subsequent tasks.

4 CPU Baseline Implementation

4.1 Reference Implementation

```python
import random
import time

N = 5_000_000
count = 0

start = time.perf_counter()

for _ in range(N):
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)
    if x*x + y*y <= 1:
        count += 1

pi_est = 4 * count / N

end = time.perf_counter()

print("Estimated Pi:", pi_est)
print("CPU runtime:", end - start)
```

Listing 2: CPU Baseline Implementation for Monte Carlo π Estimation

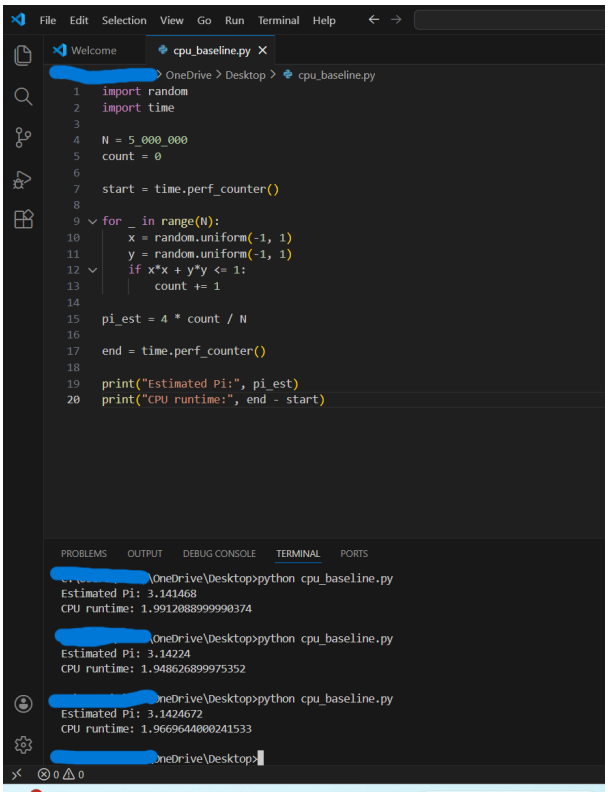

Figure 1: Example for CPU Baseline Run

5 Task 2: CUDA Execution Model Mapping

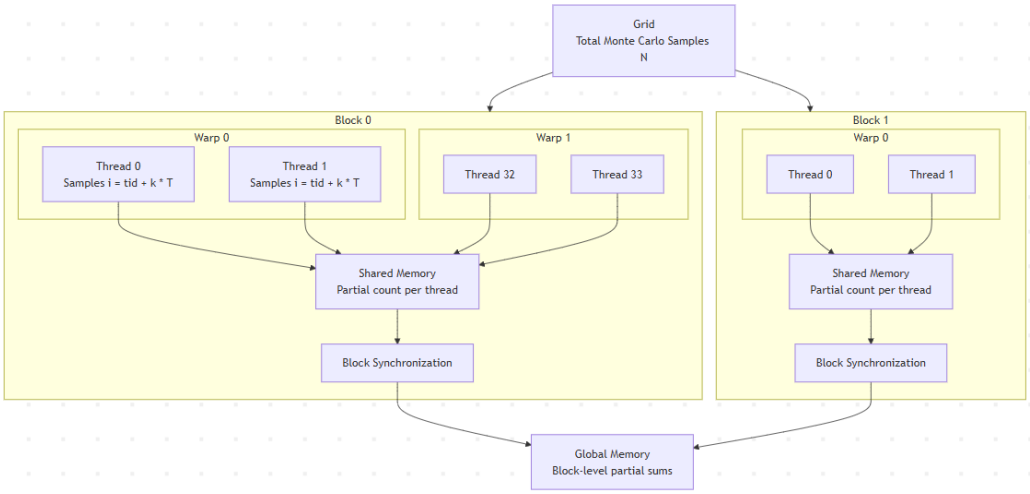

Figure 2: CUDA execution hierarchy for Monte Carlo π estimation showing the mapping of the iteration space to grid, blocks, warps, and threads. Each thread independently processes a subset of Monte Carlo samples. Shared memory is used for efficient block-level reduction, and synchronization is required only during the reduction phase.

Iteration Space Mapping

The Monte Carlo sampling loop is mapped directly onto the CUDA execution hierarchy. Each GPU thread processes multiple Monte Carlo samples using a strided access pattern. If T denotes the total number of active threads, then thread t processes samples of the form

$$i = t + k \cdot T, \quad k = 0, 1, 2, \ldots$$

This mapping ensures uniform workload distribution across threads and allows the iteration space to scale efficiently with increasing GPU parallelism.

Synchronization

Synchronization is not required during the Monte Carlo sampling phase, as each thread independently generates random points and performs point-in-circle tests without any data dependencies. Synchronization is required only during the reduction phase. After completing their assigned samples, threads within a block write their local counts to shared memory. A block-level synchronization barrier using `__syncthreads()` ensures that all threads have completed their shared-memory writes before participating in the block-level reduction. No global synchronization is required within the kernel.

Shared Memory Usage

Shared memory is used to store per-thread partial counts within each block. Each thread writes its local count to a shared memory location indexed by its thread ID. A parallel reduction is then performed within the block to compute a single block-level partial sum. This approach significantly reduces the number of global memory accesses and avoids the use of expensive atomic operations on global memory.

Memory Bottleneck Analysis

The Monte Carlo π estimation workload exhibits minimal memory bottlenecks during the sampling phase, as computations are performed entirely using registers with no global memory accesses. The primary memory bottleneck arises during the reduction phase, where partial results from multiple threads must be aggregated. A naive implementation using global atomic updates would result in high contention and serialized execution. This bottleneck is mitigated by performing intra-block reductions using shared memory, such that only one partial sum per block is written to global memory. As a result, global memory traffic is minimized and the kernel remains predominantly compute-bound.