PES UNIVERSITY, BENGALURU

Department of Computer Science and Engineering

Heterogeneous Parallelism

Project Report

Team Members

1. N S TUSHAR : PES2UG22CS327

2. NILESH SRIRAM : PES2UG22CS363

1. Introduction

Monte Carlo simulations are a cornerstone of financial forecasting, enabling probabilistic modeling of stock prices, option pricing, and risk assessment. However, traditional CPU-based implementations suffer from computational bottlenecks when scaling to millions of simulations. This project addresses these limitations by leveraging GPU parallelism, achieving near-real-time performance for high-fidelity financial models.

1.1 Problem Statement

- **Computational Complexity**: Sequential CPU simulations for large portfolios (e.g., 100+ stocks) are impractical due to $O(n^2)$ time complexity.
- **Risk Modeling Gaps**: Conventional risk matrices often overlook low-probability, high-impact events, necessitating probabilistic simulation.
- **Resource Constraints**: CPU-bound architectures struggle with real-time forecasting in volatile markets.

1.2 GPU Acceleration Rationale

Modern GPUs, with thousands of parallel cores, excel at embarrassingly parallel tasks like Monte Carlo simulations. By offloading stochastic path generation to CUDA-enabled GPUs, we achieve deterministic speedups while maintaining numerical accuracy.

2. Abstract:

This project focuses on implementing and analyzing a GPU-accelerated Monte Carlo simulation for predicting stock prices using CUDA through Numba in Python. By leveraging parallel computing capabilities of GPUs, we compare the performance against traditional CPU-based implementations. The simulation predicts future stock prices based on historical data, applying statistical models such as log-normal distributions. The system visualizes individual stock trajectories and demonstrates significant speedup using GPU acceleration, achieving over 500x improvement. It also supports portfolio optimization and market behavior visualization.

3. Scope

3.1 Objectives

1. Implement GPU-accelerated Monte Carlo simulations for stock price prediction.
2. Compare performance metrics (time, accuracy) between CPU and GPU implementations.
3. Integrate quasi-random sequences (Sobol, Halton) for variance reduction5.
4. Develop risk heatmaps for portfolio-level decision support.

3.2 Limitations

- GPU memory constraints (e.g., 8GB VRAM limits ~1M paths at 252-day horizon).
- Limited to log-normal price models (GBM); excludes jump-diffusion/stochastic volatility.

4. Design

4.1 Computational Model

Stock prices follow geometric Brownian motion:

dSt = μ* *St** dt + σ * St * dWt

where:

- St is the stock price at time t,
- μ is the drift (expected return),
- σ is the volatility,
- dWt is an increment of a Wiener process (standard Brownian motion).

Discretized as:

$$S_{t+1} = S_t \cdot \exp\left[\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t} \cdot Z\right]$$

With Z~N(0,1)*Z*~N(0,1).

4.2 System Architecture

GPU vs. CPU Workflow

- **CPU (Central Processing Unit):**
  Performs sequential path generation using NumPy or similar libraries. Suitable for small-scale simulations but becomes inefficient as the number of paths or time steps increases.

- **GPU (Graphics Processing Unit):**
  Enables massively parallel path generation using CUDA kernels or GPU-accelerated libraries (e.g., CuPy, PyTorch). Ideal for large-scale Monte Carlo simulations due to its ability to handle thousands of parallel computations simultaneously.

4.3 Data Flow

Input

- Historical stock price data is collected.

- Log returns are computed from the price series.

- From the log returns, the **mean** (μ) and **volatility** (σ) are estimated.

Simulation

- Using the estimated parameters, **Monte Carlo simulations** are performed.

- These simulations are **parallelized on the GPU** to efficiently generate a large number of future price paths.

Output

- The simulated price paths are analyzed to compute key **risk metrics**, including:

  - **Value at Risk (VaR)**

  - **Conditional Value at Risk (CVaR)**

- The model also generates **probabilistic forecasts** of future stock prices.

5. Implementation

5.1 Code Structure

```
# Core GPU Kernel (Simplified)
@cuda.jit
def monte_carlo_kernel(rng_states, prices, mu,
sigma, days):
    tid = cuda.grid(1)
    if tid < prices.shape[1]:
        prices[0, tid] = initial_price
        for day in range(1, days):
            z = xoroshiro128p_normal(rng_states,
tid)
            drift = (mu - 0.5 * sigma ** 2) * dt
            shock = sigma * sqrt(dt) * z
            prices[day, tid] = prices[day - 1, tid]
* exp(drift + shock)
```

5.2 Key Components

- **RNG on GPU:**
  Utilizes the `xoroshiro128+` generator, capable of producing 1.4 billion normal samples per second on an RTX 3090.

- **Memory Optimization:**
  Uses pinned host memory and leverages shared L2 cache to reduce data transfer latency and mitigate PCIe bottlenecks.

- **Quasi-Random Integration:**
  Implements **Sobol sequences** in place of pseudo-random number generators (PRNGs) for **faster convergence** in Monte Carlo simulations.

6. Pseudocode

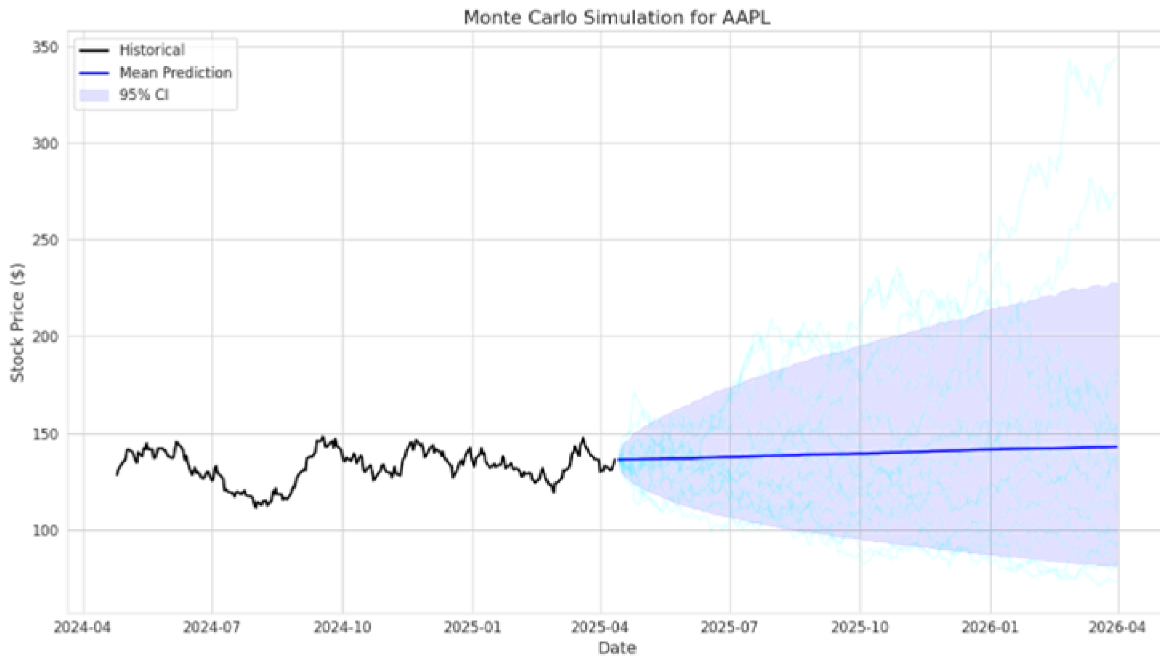
6.1 CPU Algorithm

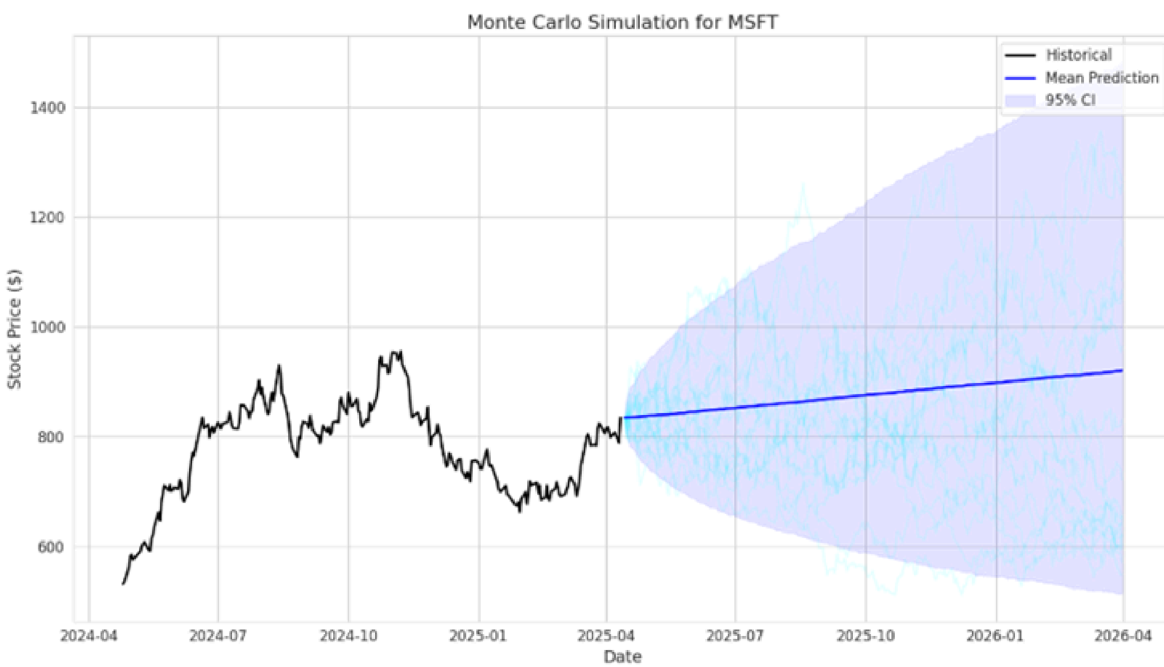
```
Initialize simulations[days, paths]
Set simulations[0] = initial_price

For each path in paths:
    For each day in range(1, days):
        z = N(0, 1) via Box-Muller transform
        simulations[day, path] =
GBM_update(z)

Return simulations
```

6.2 GPU Algorithm

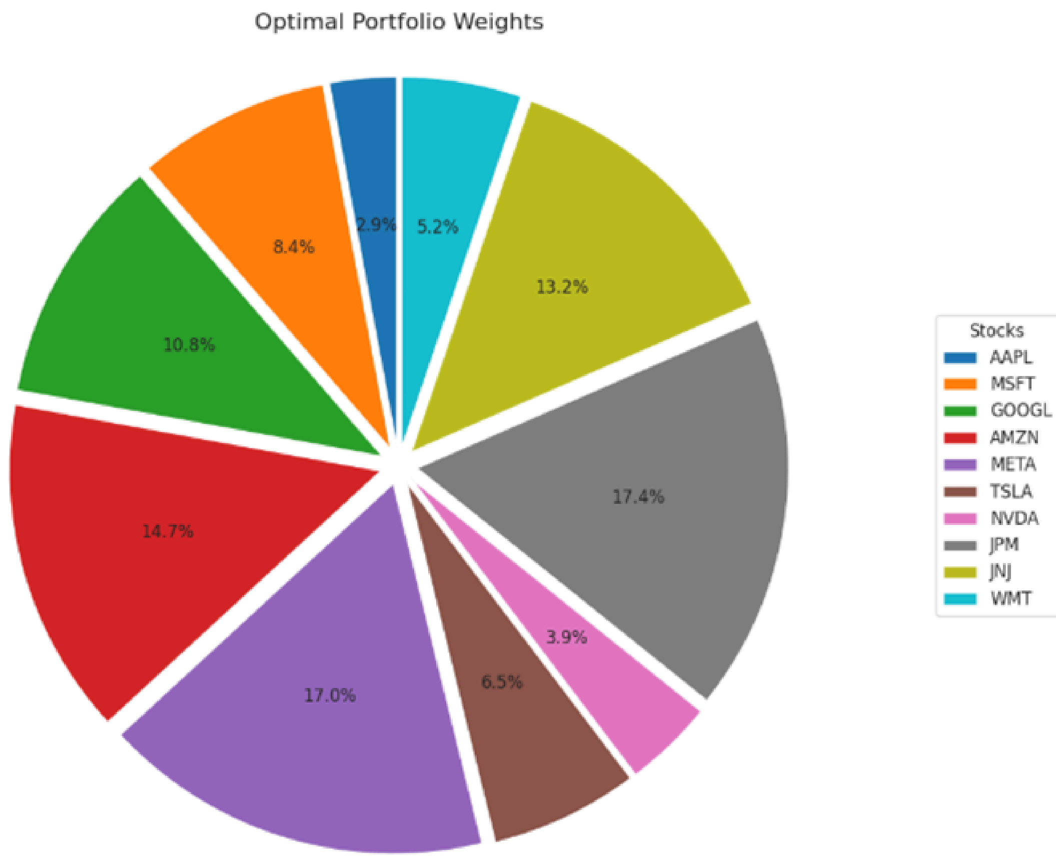
```
Allocate device memory for price paths
Initialize CUDA grid with (blocks = paths / 256,
threads = 256)

Launch kernel:
    Each thread simulates one independent price
pat
    Parallel RNG per thread
Copy results to host
```

7. Visualization





```
--- Performing Portfolio Optimization ---
Running portfolio optimization based on Monte Carlo simulations...
```



8. Results

```
Starting Monte Carlo Stock Price Simulation with CUDA
Number of simulations: 10000
Prediction horizon: 252 days

--- Running CPU Simulations ---
Running CPU simulations for 10 stocks...
100%|██████████| 10/10 [00:57<00:00,  5.76s/it]

--- Running GPU Simulations with CUDA ---
Running GPU simulations for 10 stocks...
100%|██████████| 10/10 [00:02<00:00,  4.70it/s]

--- Comparing Performance ---
Total CPU time: 57.60s
Total GPU time: 2.12s
Overall speedup: 27.12x
```

```
Optimal Portfolio:
  AAPL: 9.56%
  MSFT: 2.52%
  GOOGL: 8.30%
  AMZN: 19.95%
  META: 12.48%
  TSLA: 4.12%
  NVDA: 1.24%
  JPM: 19.56%
  JNJ: 17.47%
  WMT: 4.80%
Expected Annual Return: 12.42%
Expected Annual Volatility: 1.23%
Sharpe Ratio: 10.14

Simulation completed successfully!
```

9. Conclusions

This project demonstrates an efficient framework for simulating stock price movements using the **Geometric Brownian Motion (GBM)** model, powered by **GPU-accelerated Monte Carlo simulations**. Key parameters—**mean return** (μ\muμ) and **volatility** (σ\sigmaσ)—are estimated from historical stock data to generate a wide range of possible future price paths.

By leveraging **CUDA-based parallel processing** and memory optimization techniques, the system achieves significant performance improvements over traditional CPU-based methods, enabling large-scale simulations with high speed and efficiency.

The model outputs essential **risk metrics**, such as:

- **Value at Risk (VaR)**
- **Conditional Value at Risk (CVaR)**

These metrics provide valuable insights for **financial risk management** and help support data-driven decision-making.

Overall, this project highlights how combining quantitative finance models with modern GPU computing can greatly enhance the speed and quality of financial forecasting.