Monte Carlo Pricing Optimization (CPU vs GPU)

> Parallel Programming Final Project — Group 23
> Members: David Lu (T14902116), 蔡琦皇 (P13922006), 楊翊廷 (R14944018), 詹淯翔 (B13201026)

Introduction

Monte Carlo (MC) simulation is one of the most flexible approaches for option pricing, especially for products without closed-form solutions or with complex payoffs (e.g., Asian options) and higher-dimensional problems (e.g., basket/multi-asset options). However, MC is computationally expensive: achieving low statistical error typically requires **millions to billions of paths** and (for path-dependent products) **hundreds of time steps**, which can be too slow on CPUs for latency-sensitive use cases.

This project implements MC option pricing with **three levels of parallelism**:

- **CPU OpenMP** (multi-core parallel paths)
- **Single-GPU CUDA** (massively parallel path simulation)
- **Multi-GPU CUDA** (distribute paths across multiple GPUs for basket pricing)

In addition, we calibrate simple GBM parameters from real historical data (`data/`) and benchmark performance/scalability under different workloads.

Method

1) Stochastic model (GBM)

We simulate the underlying asset price using Geometric Brownian Motion (GBM):

$$S_{t+\Delta t} = S_t \cdot \exp\big(\left(r - \tfrac{1}{2}\sigma^2\right)\Delta t + \sigma\sqrt{\Delta t}, Z\big), \quad Z \sim \mathcal{N}(0,1)$$

For multi-asset baskets, we generate **correlated Gaussian vectors** Z using an equicorrelation covariance structure and **Cholesky decomposition** $igma = LL^{\top}, thenapplyy = Lz.$

2) Payoffs implemented

This repo implements three core payoffs (plus different experiment groupings):

- **European Call (single asset)**: payoff depends only on terminal price S_T.
  - Validation: compare MC estimate vs **Black–Scholes closed-form**.
- **Asian Arithmetic Call (single asset)**: payoff depends on arithmetic average along the path.
- **Basket European Call (multi-asset)**: payoff depends on arithmetic mean of terminal prices across assets; correlation handled via Cholesky.

CPU implementation: `src/cpu/mc_pricer.cpp`
GPU implementation: `src/gpu/gpu_pricer.cu` + core kernel/workspace in `src/gpu/mc_pricer.cu`

3) Parallelization strategy

- **CPU (OpenMP)**: each thread simulates independent paths (per-thread RNG seeded by thread id).

  - Implementation: OpenMP parallel loop in `src/cpu/mc_pricer.cpp`.

- **GPU (CUDA)**:

  - RNG: per-thread cuRAND Philox states (`curandStatePhilox4_32_10_t`).
  - Kernel design: grid-stride loop where each CUDA thread processes many paths.
  - Accumulation: atomic accumulation of sum and sum of squares to compute mean and standard error.
  - Workspace reuse: allocate device buffers once (states/sum/sum2) and reuse across calls.
  - Implementation: `src/gpu/mc_pricer.cu` (kernel `mc_kernel_general`, workspace `GpuWorkspace`).

- **Multi-GPU (basket)**:

  - Split total paths across devices; each GPU runs the basket kernel on its portion.
  - Combine partial means/variances to produce final mean and standard error.
  - Implementation: `src/gpu/gpu_pricer.cu` (multi-GPU path splitting + merge).

4) Parameter calibration from real data

We calibrate basic GBM parameters from historical CSVs:

- S_0: last close price
- \sigma: annualized volatility from daily log returns
- \rho: average off-diagonal correlation for the multi-asset set

Implementation: `src/tools/calibrate_from_data.py`
Outputs a shell script `params.sh` used by `src/cpu/run_all.sh` and `src/gpu/run_all.sh`.

Experiment

All experiments are scripted and write results as CSV rows with timing and standard error, then generate plots.

1) How experiments are run

- **GPU experiments**: `src/gpu/run_experiments.sh`
- **CPU experiments**: `src/cpu/run_experiments.sh`
- **Run everything + plot**: `src/run_all_experiments.sh`
- **Plotting**: `src/plot_experiments.py` (reads `src/experiments/results/*.csv`, writes `src/experiments/plots/*.png`)

2) Experiment dimensions (what we vary)

From the proposal and the actual scripts, the repo benchmarks:

- **Paths scaling**: vary number of Monte Carlo paths N (GPU goes up to 10^9 paths in script; CPU uses smaller N).
- **Steps scaling**: vary time steps (e.g., 64, 128, 252, 512) for Asian/basket discretization.
- **CPU threads scaling** (OpenMP): vary thread count for basket.
- **Multi-GPU scaling** (basket): vary number of GPUs (1/2/4) and measure speedup.

- **Assets scaling** (basket): vary number of assets (2/4/8/16/32) to show dimensionality cost.
- **GPU configuration tuning**: block size and blocks-per-SM (occupancy) tuning.
- **Option type comparison**: compare European vs Asian vs Basket under same path counts.
- **Direct GPU vs CPU comparison**: for multiple option types under aligned path counts.

3) Outputs

Results are written to:

- `src/experiments/results/*.csv`

Plots are written to:

- `src/experiments/plots/*.png`

Each CSV row includes (examples): engine, type, workers (threads or GPUs), paths, steps, assets, rho, S0/K/r/sigma/T, price, std_error, time_ms (and GPU tuning fields for block size).

Conclusion

Based on the project slides and the implemented experiments:

- **GPU acceleration is substantial** for MC pricing because paths are embarrassingly parallel.
- **Complexity increases significantly** as we move from single-asset to multi-asset basket pricing due to correlation handling (Cholesky and matrix-vector operations) and higher dimensional simulation.
- **Speedup characteristics differ by product**:
    - Asian option shows a relatively consistent GPU speedup in the reported results.
    - European option speedup depends more on path count (GPU becomes more favorable for larger workloads).
- **Best GPU configuration (from block tuning in slides)**:
    - Block size ≈ **256**
    - Occupancy setting ≈ **2 blocks per SM**
- Overall, the repo demonstrates that CUDA (and multi-GPU for basket) can reduce runtime dramatically while keeping pricing accuracy consistent (standard error decreases with more paths, and European can be validated against Black–Scholes).

Work Distribution

- **David Lu (T14902116)**: Presentation
- **蔡琦皇 (P13922006)**: CPU baseline implementation (OpenMP Monte Carlo pricer)、single-asset implementation、CPU experiment scripts.
- **楊翊廷 (R14944018)**: Experiment design/automation and results visualization (CSV schema, plotting scripts, figure generation),multi-asset implementation、Data calibration tooling from real market CSVs and integration into run pipelines
- **詹淯翔 (B13201026)**: report generation.

ref.

- 1.An introduction to Monte Carlo methods [https://arxiv.org/pdf/1404.0209]
- 2.GitHub [https://github.com/Yiting1022/pp-final.git]