David Lu T14902116
蔡琦皇 P13922006
楊翊廷 R14944018
詹淯翔 B13201026

Parallel Programming Final Project Proposal:

Topic: High performance computing / supercomputing (parallel Monte Carlo simulation on GPUs). Elements of "parallelism in other topics" (computational finance).

Motivation: Amongst the various option pricing techniques, Monte Carlo is one of the most flexible and uses simulation to generate prices of an underlying asset. It is often used to model Asian and European options.

However, Monte Carlo methods are computationally expensive. Simulating millions of price paths, each with hundreds of steps, requires lots of computational power. On a CPU, simulations can take seconds to minutes, which is too slow for the real world.

GPUs are well suited for this type of programming. Each simulation path is independent, which means it's pretty natural to parallelize them. The goal of this project is to maximize simulation throughout and explore the use of CUDA in finance.

Background: To understand multi assets, let's first understand single assets.

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

S_t : price of the security at time t,

$S_{t+\Delta t}$: price of the security after one time step Δt,

r : risk-free interest rate,

σ : volatility of the security (annualized standard deviation).

$$\Delta t = \frac{\text{time period}}{\text{number of desired ticks}}$$

Each simulation of the equation above is one path. We simulate the paths millions of times and take the discounted average as our final expected value. Let K be the strike price of our European option. Let C be the fair value of the option price.

$$C = e^{-rT} \cdot \frac{1}{N} \sum_{i=1}^{N} \text{payoff}_i$$

$$\text{payoff}_i = \max(S_T - K,\, 0)$$

However, what we've talked about so far is just for single assets. In the real world, we have many assets that are correlated together in some way. So we extend our idea.

$$S_{t+\Delta t}^{(j)} = S_t^{(j)} \cdot \exp\left(\left(r - \frac{1}{2}\sigma_j^2\right)\Delta t + \sigma_j \sqrt{\Delta t}\, Z_t^{(j)}\right), \quad j = 1, \ldots, M$$

The only major difference here is Z. Z is no longer a single-variate normal distribution, but instead, multi-variate:

$$Z = \left(Z^{(1)}, \ldots, Z^{(M)}\right). \; Z \sim \mathcal{N}(0, \Sigma)$$

And as before, the covariance matrix Sigma is based on historical data. Sigma_{i}{j} measures how correlated the ith and jth security are. Now, since cudaRAND does not provide a multi-variate distribution natively, we will have to implement it ourselves. We start with independent gaussians:

$$u_1, u_2, \ldots, u_M \sim N(0, 1)$$

Find a L such that (Cholesky decomposition):

$$LL^T = \Sigma$$

Then, to get the actual distributions, get

$$Z = Lu$$

Objectives:

1. Correctly implement single/Multi Asset Monte Carlo options pricing model.
    a. Validate correctness using the Black–Scholes closed-form solution.
    b. Establish a baseline for later GPU speedups.
2. Extend to multi-asset simulation on CPU
    a. Load correlation matrices from files.
    b. Implement Cholesky decomposition.
    c. Generate correlated Gaussian vectors
    d. Validate correctness with known covariance structures.
3. Design a GPU-accelerated Monte Carlo engine using CUDA
    a. RNG setup using cuRAND device states.
    b. Kernel where each thread simulates a full price paths
    c. Support both single-asset and multi-asset cases.
4. Optimize performance
    a. Grid-stride loops (if needed).
    b. Minimize global memory accesses.
    c. Shared memory for Cholesky matrix.
    d. Fast math intrinsics.
    e. Consider pre-simulated normals vs on-the-fly RNG.
5. Benchmark and analyze scalability
    a. GPU vs CPU performance for different parameters (N, M, T)
    b. Determine how GPU performance changes with workload size.
    c. Identify bottlenecks and propose improvements.

Implementation/Schedule:

Phase 1: CPU Single Asset

- Implement Monte Carlo using C++ STL <random>.
- Test varying path counts and step sizes.
- Compare results to Black–Scholes closed-form values to ensure correctness.

Phase 2: CPU Multi Asset

- Load or generate covariance matrices.
- Implement Cholesky decomposition from scratch.
- Validate by checking that the reconstructed covariance matrix approximates the input.
- Generate correlated normals and verify empirical correlations match expectations.

Phase 3:

- Allocate device memory for:
  - initial prices
  - drift, volatility
  - Cholesky matrix
  - RNG states
  - payoff results
- Write kernel:
  - Each thread performs one entire path simulation.
  - Use cuRAND to generate independent normals.
  - Multiply by Cholesky matrix to create correlated normals.
  - Update prices and compute payoff.

Phase 4 — Optimization and Tuning:

We will identify bottlenecks using:

- Nsight Compute
- CUDA Profiling Tools

Phase 5:

We will measure:

- Time to simulate N paths for various N (e.g., 10^6, 5×10^6, 10^7).
- Time scaling with number of assets M.
- GPU speedup vs CPU baseline.

Final deliverables:

- Graphs (log-scaled if needed)
- Tables comparing performance
- Presentation
- Runtime analysis & discussion