Parallel Numerical Methods for Option Pricing: Monte Carlo and Finite Difference Approaches

by

Mark BECKMANN,
Gabriel CANAPLE,
Clément GILLIER,
Elie TARASSOV

Department of Computer Science
INSA Lyon
Villeurbanne, France

November 2024

Abstract

This report explores numerical techniques for solving the Black-Scholes partial differential equation used to value European options. We focus on Monte Carlo simulations and finite difference methods, optimizing their computational performance with parallel processing techniques, leveraging multithreaded CPUs and CUDA-enabled GPUs. Benchmarking results highlight the efficiency gains in terms of execution time. In short, we make Option Pricing go BRMMMM!

Contents

Chapter 1

Introduction

1.1 Background

Financial derivatives are fundamental instruments in modern finance, allowing market participants to hedge risk, speculate on future price movements, or gain exposure to specific market conditions. Among the most well-known derivatives are **options**, which are contracts providing the buyer the right, but not the obligation, to buy or sell an underlying asset at a predetermined price (strike price) before or at a specified expiration date.

Options are classified into two primary types based on their exercise conditions:

- **European Options**: Can only be exercised at the expiration date.
- **American Options**: Can be exercised at any time up to and including the expiration date.

Options can further be divided based on their position and payoff structure:

- **Call Options**: Provide the right to purchase the underlying asset. The payoff for a call option is given by:

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

  where S_T is the underlying asset price at maturity, and K is the strike price.

- **Put Options**: Provide the right to sell the underlying asset. The payoff for a put option is:

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

Participants in the options market can take either a **long position** (buying the option) or a **short position** (selling the option), further expanding the strategies available to traders.

1.2 The Black-Scholes Model

The valuation of European options is often conducted using the **Black-Scholes model**, a seminal framework in financial mathematics. This model assumes that the price of the

underlying asset follows a geometric Brownian motion, characterized by:

$$dS = \mu S dt + \sigma S dW$$

where S is the asset price, μ is the drift rate, σ is the volatility, and W is a Wiener process. Under certain conditions, the model reduces the pricing problem to a partial differential equation (PDE), known as the **Black-Scholes equation**:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

Here, $V(S, t)$ represents the option price as a function of the underlying asset price S and time t, and r is the risk-free interest rate.

1.3 Problem Statement and Objectives

Accurate and efficient computation of option prices is critical for financial institutions, particularly when dealing with portfolios containing large numbers of derivatives or requiring real-time valuation. The Black-Scholes model, while mathematically elegant, can become computationally expensive when extended to simulate large-scale scenarios or multidimensional problems.

In this project, we aim to **parallelize the computation of option pricing** using modern numerical techniques. Specifically, we focus on:

- **Monte Carlo simulation**: For stochastic modeling of asset price paths.
- **Finite difference methods**: For solving the Black-Scholes PDE.

Our goal is to optimize these numerical methods through parallel computation, leveraging both **multithreaded CPUs** and **general-purpose GPUs (GPGPUs)** via OpenMP and CUDA programming. This approach seeks to significantly reduce computation time, enabling fast and scalable pricing solutions.

Chapter 2

Numerical Procedures for Solving the Problem

2.1 Overview

The valuation of options based on the Black-Scholes equation can be approached using various numerical methods. We focus on two widely used techniques: **Monte Carlo Simulation** and the **Finite Difference Method**. Each method has its own characteristics and is suited for specific types of problems in financial modeling.

2.1.1 Monte Carlo Simulation

Monte Carlo simulation is a stochastic method that leverages random sampling to model the behavior of complex systems. In the context of option pricing, it involves simulating multiple paths of the underlying asset price using the geometric Brownian motion model:

$$S_{t+\Delta t} = S_t \exp\left(\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\, Z\right)$$

where Z is a random variable sampled from a standard normal distribution. The simulated paths are then used to compute the option payoff, which is averaged and discounted to determine the option price.

2.1.2 Finite Difference Method

The finite difference method is a deterministic approach that discretizes the Black-Scholes PDE over a computational grid in both time and asset price dimensions. The PDE is then solved iteratively, using techniques such as explicit, implicit, or Crank-Nicolson schemes, to approximate the option price at each grid point.

2.2 Monte Carlo Simulation in C++, with OpenMP and CUDA

2.2.1 Starting Point and Performance Profiling

...

2.2.2 Possible Optimizations

...

Listing 2.1: Optimized Monte Carlo Simulation

...

2.2.3 Cuda Optimization

To try to optimize the algorithm as much as possible, we are now going to push the code onto the graphics card using Cuda. Cuda (Compute Unified Device Architecture) is a programming model developed by NVIDIA for parallel computing on GPUs (Graphics Processing Units). The idea is to move the intensive calculations that take place on the CPU to the GPU. As the GPU is equipped with several SMs (Scalar Multiprocessors), themselves capable of executing several hundred threads in parallel, the use of Cuda enables intensive parallelization of calculations. We are going to use this computing power to parallelize the execution of all the simulations.

To do this, we are going to create two Cuda kernels. The first is used to initialize our random number generator. The second assigns one or more simulations to be calculated for each thread. The result of each simulation is then stored in global memory and sent back to the CPU where the final mean can be calculated.

Cuda random number generator

```
__global__ void setup_kernel(
    curandStatePhilox4_32_10_t *state,
    long int random_thing
) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    curand_init((1234 +
        random_thing * threadIdx.x *
        blockIdx.x * blockDim.x)%14569,
        id, 0, &state[id]
    );
}
```

Cuda compute simulation

```
__global__ void generate_monte_carlo_bs(
    curandStatePhilox4_32_10_t *state,
    long int nbSim,
    int lengthSim,
    double* result,
    double K,
    double S0 = 100.0,  // Initial stock price
    double T = 1.0,     // Time to maturity (1 year)
    double r = 0.05,    // Risk-free rate (5%)
    double sigma = 0.2  // Volatility (20%)
) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    curandStatePhilox4_32_10_t localState = state[id];

    double dt = T / lengthSim;
    double drift = (r - 0.5 * sigma * sigma) * dt;
    double diffusion = sigma * sqrt(dt);

    for (int i=id; i<nbSim; i+=blockDim.x*gridDim.x) {
        double ST = S0;
        for (int j=0; j<lengthSim; ++j) {
            double r = curand_normal_double(&localState);
            ST *= exp(drift + diffusion * r);
            state[id] = localState;
        }
        if (ST > K) {
            result[i] = ST - K;
        } else {
            result[i] = 0;
        }
    }

}
```

Benchmarking

...

2.3 Python Implementation of Monte Carlo Simulation

We evaluate the performance of several optimization approaches for computing the Black-Scholes equation via Monte Carlo simulations:

- **Classical Python** (baseline)

- **Vectorization with Numpy**
- **Multiprocessing and Threading**
- **Numba for Just-In-Time Compilation**
- **Cython**

Our aim is to analyze and compare the performance (execution time) of these methods and understand how different techniques can address the computational inefficiencies of the classical Python approach.

2.3.1 Methodology

The primary task is to simulate the evolution of a stock price using the Black-Scholes model over multiple paths, then compute the payoff for each path (for a European call option) and finally estimate the option price as the discounted average of these payoffs.

Classical Python

The classical Python implementation uses nested loops to simulate each path and calculate the corresponding payoff. Each path generates random samples from the normal distribution via `np.random.normal(0, 1)`. While this approach works correctly, it is inefficient, especially for a large number of paths and simulations due to the repeated use of random number generation and the for-loop structure.

Hypothesis for Improvement: The execution time can be reduced by leveraging more efficient tools for numerical computation and parallelism.

Optimized Approaches

Vectorization with Numpy The vectorized approach eliminates the nested loop by using NumPy's vectorized operations to generate all random variables at once and compute the price paths in a single step. The use of `np.cumsum` calculates the cumulative sum of the logarithmic returns across all paths, improving efficiency.

Hypothesis for Improvement: Vectorization should significantly reduce execution time by removing explicit loops and leveraging optimized C-based implementations of NumPy operations.

Multiprocessing Multiprocessing leverages multiple CPU cores to parallelize path simulation across multiple processes. Each process runs the `simulate_path` function independently for a subset of the total paths.

Hypothesis for Improvement: Parallelization should improve performance by distributing the workload across multiple cores, potentially reducing execution time for large simulations.

Numba for Just-In-Time Compilation Numba is used to compile the function just-in-time (JIT), optimizing loops and array operations for performance. The `@njit(parallel=True)` decorator enables parallel execution within the loop, improving execution speed on multicore machines.

Hypothesis for Improvement: JIT compilation should significantly speed up the computation by converting Python code into machine code, removing the overhead of interpreted execution. Parallelization within Numba will also distribute the workload efficiently.

Cython Cython is used to compile the function into C, offering faster execution by leveraging C-based performance. Mathematical operations are explicitly optimized using the C math library for functions like `exp` and `sqrt`.

Hypothesis for Improvement: Cython should provide a substantial speedup by compiling Python code to C, reducing overhead and increasing efficiency for numerical operations.

2.3.2 Results

Method	Mean Option Price	Execution Time (seconds)
Classic	10.4933	5.4152
Vectorized	10.4690	0.0419
Multithreading	10.5132	6.4855
Multiprocessing	10.4507	4.4074
Numba	10.4272	0.2789
Cython	21.0213	0.0090

Table 2.1: Performance Comparison of Optimization Techniques

2.3.3 Discussion and Interpretation

- **Classical Python**: The classical method has high execution times due to inefficient looping and repeated random number generation for each path.
- **Vectorized Approach**: The vectorized approach significantly improves execution time by removing explicit loops and utilizing optimized NumPy functions. However, the mean option price is slightly different, likely due to how the random normal samples are handled across paths.
- **Multithreading and Multiprocessing**: While both approaches allow parallel computation of paths, the overhead from managing threads/processes limits their effectiveness in this case. **Multithreading** resulted in a slightly higher execution time compared to multiprocessing, likely due to the Global Interpreter Lock (GIL) in CPython.
- **Numba**: Numba significantly reduced execution time and showed reasonable accuracy, although the mean option price differed slightly from the others. The benefit of JIT compilation and parallel execution was evident, particularly for larger numbers of paths.
- **Cython**: Cython yielded the fastest execution time and the highest option price, which suggests that the computation is highly sensitive to the optimization at the C level. The accuracy of the price might be affected by the precision of the floating-point calculations in C, which could differ slightly from the Python-based methods.

2.3.4 Conclusion

Optimizing Monte Carlo simulations for Black-Scholes option pricing can yield significant improvements in execution time. Vectorization with NumPy, Numba's JIT compilation, and Cython offer the best performance, with Cython providing the fastest results at the cost of slight deviations in the option price. The choice of optimization depends on the trade-off between execution time and result accuracy, with Cython being the optimal choice for speed and Numba offering a good balance between speed and flexibility.

2.4 Finite Difference Method

2.4.1 Basic Implementation

Below, we present the implementation of an explicit finite difference scheme for a European call option.

Explicit Finite Difference Scheme

```
import numpy as np

# Parameters
S_max = 200  # Maximum stock price
T = 1        # Maturity in years
sigma = 0.2  # Volatility
r = 0.05     # Risk-free rate
K = 100      # Strike price
M, N = 100, 100  # Time and price grid size
dt, dS = T / M, S_max / N

# Initialize grid and boundary conditions
V = np.zeros((M+1, N+1))
S = np.linspace(0, S_max, N+1)
V[-1, :] = np.maximum(S - K, 0)  # Terminal payoff

# Finite difference iteration
for i in reversed(range(M)):
    for j in range(1, N):
        delta = (V[i+1, j+1] - V[i+1, j-1]) / (2 * dS)
        gamma = (V[i+1, j+1] - 2 * V[i+1, j] + V[i+1, j-1]) / (dS**2)
        V[i, j] = V[i+1, j] + dt * (0.5 * sigma**2 * S[j]**2 * gamma
                                    + r * S[j] * delta - r * V[i+1, j])

# Extract option value at t=0
option_price = V[0, int(N/2)]
print("Option Price:", option_price)
```

Limitations

The explicit scheme requires a fine grid resolution to maintain stability and accuracy. Boundary conditions also need careful treatment, as poorly chosen boundaries can lead to errors in the solution.

2.4.2 Performance Profiling

We profile the above implementation to identify inefficiencies. Common challenges include:

- **Memory usage:** The grid can become large for fine resolutions.
- **Computation time:** Explicit schemes require a small time step for stability, increasing computation time.

2.4.3 Optimized Code

We optimize the finite difference method by:

- Using sparse matrix representations to reduce memory overhead.
- Leveraging linear solvers for implicit and Crank-Nicolson schemes.

Below is an example of an optimized implementation using sparse matrices for the Crank-Nicolson scheme:

Crank-Nicolson with Sparse Matrices

```
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve

# Build tridiagonal matrix for Crank-Nicolson
alpha = 0.5 * dt * (sigma**2 * S[1:-1]**2 / dS**2 - r * S[1:-1] / dS)
beta = 1 + dt * (sigma**2 * S[1:-1]**2 / dS**2 + r)
gamma = -0.5 * dt * (sigma**2 * S[1:-1]**2 / dS**2 + r * S[1:-1] / dS)

diagonals = [alpha, beta, gamma]
A = diags(diagonals, offsets=[-1, 0, 1]).tocsc()

# Time-stepping with Crank-Nicolson
for i in reversed(range(M)):
    rhs = V[i+1, 1:-1]
    V[i, 1:-1] = spsolve(A, rhs)

print("Optimized Option Price:", V[0, int(N/2)])
```

Benchmarking

...

Chapter 3

Conclusions and Future Work

Summarize key findings and insights. Propose improvements or extensions for future work.

Bibliography

[1] J. Hull, *Options, Futures, and Other Derivatives*, 10th ed. Pearson Education, 2017.

[2] E. Panova, V. Volokitin, A. Gorshkov, and I. Meyerov, "Black-Scholes Option Pricing on Intel CPUs and GPUs: Implementation on SYCL and Optimization Techniques," in *Supercomputing*, Springer International Publishing, 2022, pp. 48–62. DOI: 10.1007/978-3-031-22941-1_4.

[3] E. László, Z. Nagy, M. B. Giles, I. Z. Reguly, J. Appleyard, and P. Szolgay, "Analysis of Parallel Processor Architectures for the Solution of the Black-Scholes PDE," in *Proceedings of the 2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2015, pp. 1977-1980. DOI: 10.1109/ISCAS.2015.7169062.

[4] V. Cvetanoska and T. Stojanovski, "Using High Performance Computing and Monte Carlo Simulation for Pricing American Options," arXiv:1205.0106 [cs.DC], 2012. Available at: https://arxiv.org/abs/1205.0106.

Appendix A

Appendix: Mathematics and Algorithms

A.1 Monte Carlo Simulation

Monte Carlo simulation is a stochastic approach that employs random sampling to model complex systems. For option pricing, it relies on the risk-neutral valuation principle, which states that the value of a derivative can be determined by the discounted expected payoff in a risk-neutral world.

The underlying asset price S is assumed to follow a geometric Brownian motion:

$$dS = \mu S\, dt + \sigma S\, dz$$

where μ is the drift, σ is the volatility, and dz is a Wiener process.

In a risk-neutral world, the drift μ is replaced by r, the risk-free interest rate, and the discrete-time approximation of the process becomes:

$$S_{t+\Delta t} = S_t \exp\left(\left(r - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\, Z\right)$$

where $Z \sim \mathcal{N}(0,1)$ is a standard normal random variable.

The Monte Carlo simulation proceeds as follows:

1. Simulate N paths for S over the life of the derivative.
2. Calculate the payoff $\phi(S_T)$ for each path, where S_T is the terminal stock price.
3. Average the payoffs to estimate the expected value in a risk-neutral world:

$$\mathbb{E}[\phi(S_T)] = \frac{1}{N}\sum_{i=1}^{N} \phi(S_T^{(i)})$$

4. Discount this expected payoff at the risk-free rate to determine the option price:

$$V = e^{-rT}\mathbb{E}[\phi(S_T)]$$

A.2 Finite Difference Method

The finite difference method is a numerical approach to solve partial differential equations such as the Black-Scholes equation. We begin by considering the Black-Scholes PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0,$$

where V is the option value, S is the underlying asset price, σ is volatility, and r is the risk-free rate.

We discretize the time domain into M intervals of size Δt and the asset price domain into N intervals of size ΔS. Using finite difference approximations, we derive three schemes:

- **Explicit Scheme:** Uses a forward difference for time and central difference for space. This scheme is conditionally stable.
- **Implicit Scheme:** Uses a backward difference for time, requiring solving a tridiagonal system at each time step. This scheme is unconditionally stable but computationally expensive.
- **Crank-Nicolson Scheme:** Combines the explicit and implicit methods for second-order accuracy. It is unconditionally stable and often preferred in practice.

A.2.1 Stability and Convergence Conditions

The stability of the explicit scheme depends on the size of Δt and ΔS. Specifically, the Courant-Friedrichs-Lewy (CFL) condition must be satisfied to ensure convergence:

$$\Delta t \leq \frac{\Delta S^2}{2\sigma^2 S^2}.$$

Implicit and Crank-Nicolson methods, being unconditionally stable, do not require such constraints.