CudaQuants

Mohammad Hussain, Yidong Li, Yushan Lu

December 2, 2024

Introduction: What and Why

Parallelization of Monte Carlo Simulation for Barrier Option Pricing

- What is an Option?
- What is a *Barrier* Option?
- How to Price an Option?
- How does Monte Carlo Simulation help with it?

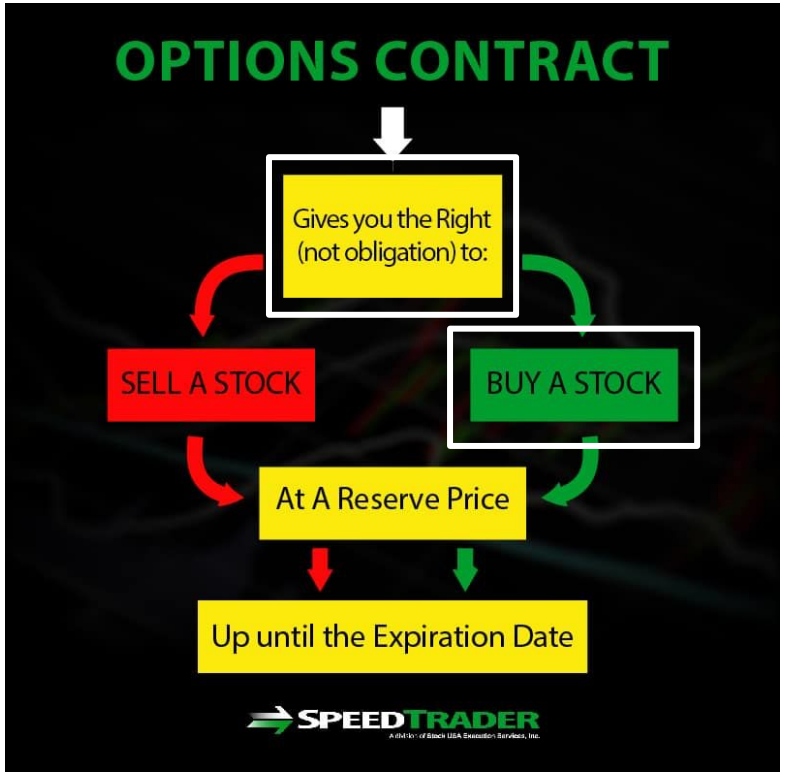
Introduction: Parallelization of Monte Carlo Simulation for Barrier Option Pricing

What is an Option?

- **Definition:**
  An option is a financial derivative that gives the holder the right, but not the obligation, to buy or sell an underlying asset at a specified price (strike price) before or on a specific date.

- **Key Terms:**
  - **Call Option:** Right to buy.
  - **Put Option:** Right to sell.

- **Example:**
  Imagine you have the a **call option** of a Tesla share for **$90** that matures on Dec 31st.
  - If the stock's price stays at **$100 on Dec 31st**, you can exercise your option and gain **$10 profit** per share.
  - If the stock's price drops below **$90**, you can let the option expire and lose only the cost of the option (premium).



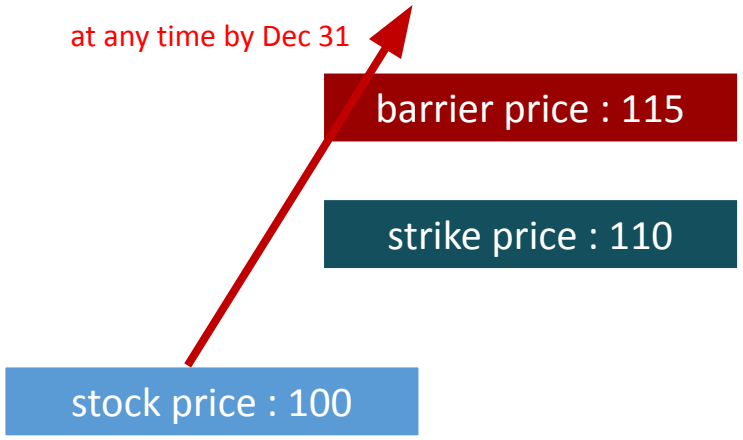
source: https://speedtrader.com/introduction-to-options-trading/

Introduction: Parallelization of Monte Carlo Simulation for Barrier Option Pricing

What is a *Barrier* Option?

- **Definition:**
  A barrier option is a type of option where its activation (or deactivation) depends on the underlying asset reaching a specific price level (the barrier).

- **Types of barrier option:**
  - **Knock-In Option:** Becomes active only if the asset price hits the barrier.
  - **Knock-Out Option:** Becomes void if the asset price hits the barrier.
- **Example:**
  - **Knock-in call option** with **1-month maturity** (expires on Dec 31) to buy Tesla stock at **$110** (strike price): it becomes active only if the stock price **reaches $115** (the barrier).
  - If Tesla's price rises to **$115 anytime** before Dec 31, the option is activated. If the stock price rises to **$120**, you can exercise it for a **$10 profit per share**.
  - If Tesla never reaches **$115**, the option is inactive and worthless.



Introduction: Parallelization of Monte Carlo Simulation for Barrier Option Pricing

How to Price an Option, and How does Monte Carlo Simulation help with it?

Monte Carlo methods, or **Monte Carlo experiments**, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be

Stochastic Differential Equation

$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

Euler Discretization

$$Y^{n+1} - Y^n = \mu Y^n \Delta t + \sigma Y^n \Delta W_t$$

- μ is the expected return per year
- σ is the expected volatility per year
- T is the time to maturity
- dt is the amount of time elapsing at each step
- S_t and S_{t-1} are the current and the previous prices
- dW is a random number distributed according to a normal distribution brownian motion component)
- Y is the price at the time step n, where $Y_0 = S_0$.

```
double sum = 0.0;
double monte_carlo_start = double(clock()) / CLOCKS_PER_SEC;

for (size_t i = 0; i < N_PATHS; i++)
{
    int n_idx = i * N_STEPS;

    float s_curr = S0;
    int n = 0;

    do
    {
        s_curr = s_curr + mu * s_curr * dt + sigma * s_curr * normals[n_idx];
        n_idx++;
        n++;
    } while (n < N_STEPS && s_curr > B);

    double payoff = (s_curr > K ? s_curr - K : 0.0);
    sum += exp(-r * T) * payoff;
}

sum /= N_PATHS;
double monte_carlo_end = double(clock()) / CLOCKS_PER_SEC;
```

Challenges

Consider a barrier option with 1 year maturity…

- To run an accurate Monte Carlo Simulation, **millions of paths** (iterations) are needed
- To use a reasonable proxy for price changes, simulating **daily changes** is required
- 5 million paths of daily changes translates to 5,000,000 * 365 random floats
- How to generate the random floats?
- How to parallelize the loop (sequential v.s. openMP v.s. GPU)?
- How to leverage different techniques (e.g. shared memory, tiling, coalescing)?

Implementation Details: Random Generation CPU v.s. GPU

Task:

- Generate 5,000,000 * 365 (1.825*10^8) floats
- Follows normal distribution

CPU: Intel(R) Xeon(R) CPU E5-2650 v3
Using std::normal_distribution for CPU
Time taken: **218,891 ms** (~219 seconds)

```cpp
// CPU: Generate random numbers
double cpu_start = double(clock()) / CLOCKS_PER_SEC;

vector<float> cpu_normals(N_NORMALS);
std::random_device rd;
std::mt19937 gen(rd());
std::normal_distribution<float> dist(0.0f, sqrdt);

for (size_t i = 0; i < N_NORMALS; i++)
{
    cpu_normals[i] = dist(gen);
}

double cpu_end = double(clock()) / CLOCKS_PER_SEC;
```

~9x Performance Improvement

GPU: NVIDIA RTX 4000 Ada Generation
Using **CURAND** for GPU
Time taken: **24,591 ms** (~24.6 seconds)

```cpp
// GPU: Generate random numbers and measure time (including transfer)
double gpu_start = double(clock()) / CLOCKS_PER_SEC;

dev_array<float> d_normals(N_NORMALS); // Array on GPU

curandGenerator_t curandGenerator;
curandCreateGenerator(&curandGenerator, CURAND_RNG_PSEUDO_MTGP32); // Mersenne Twis
curandSetPseudoRandomGeneratorSeed(curandGenerator, 1234ULL);
curandGenerateNormal(curandGenerator, d_normals.getData(), N_NORMALS, 0.0f, sqrdt);

// Copy the random numbers from GPU to CPU
vector<float> normals(N_NORMALS);
d_normals.get(&normals[0], N_NORMALS);

double gpu_end = double(clock()) / CLOCKS_PER_SEC;
```

Implementation Details: Sequential v.s. openMP v.s. GPU Parallelization

- N_PATHS: 5,000,000 (simulations)
- N_STEPS: 365 (simulates daily changes)
- normals[n_idx]: array that stores 1.825*10^8 floats

```cpp
// CPU Monte Carlo Simulation
double sum = 0.0;
double monte_carlo_start = double(clock()) / CLOCKS_PER_SEC;

for (size_t i = 0; i < N_PATHS; i++)
{
    int n_idx = i * N_STEPS;

    float s_curr = S0;
    int n = 0;

    do
    {
        s_curr = s_curr + mu * s_curr * dt + sigma * s_curr * normals[n_idx];
        n_idx++;
        n++;
    } while (n < N_STEPS && s_curr > B);

    double payoff = (s_curr > K ? s_curr - K : 0.0);
    sum += exp(-r * T) * payoff;
}

sum /= N_PATHS;
double monte_carlo_end = double(clock()) / CLOCKS_PER_SEC;
```

```cpp
// OpenMP CPU Monte Carlo Simulation
double sum_openmp = 0.0;
double start = double(clock()) / CLOCKS_PER_SEC;

#pragma omp parallel for reduction(+ : sum_openmp)
for (int i = 0; i < N_PATHS; i++)
{
    int n_idx = i * N_STEPS;

    float s_curr = S0;
    int n = 0;

    do
    {
        s_curr = s_curr + mu * s_curr * dt + sigma * s_curr * normals[n_idx];
        n_idx++;
        n++;
    } while (n < N_STEPS && s_curr > B);

    double payoff = (s_curr > K ? s_curr - K : 0.0);
    sum_openmp += exp(-r * T) * payoff;
}

sum_openmp /= N_PATHS;
double end = double(clock()) / CLOCKS_PER_SEC;
```

Implementation Details: Sequential v.s. openMP v.s. GPU Parallelization

	Time	Step Speedup	Cumulative Speedup
Sequential CPU: Intel(R) Xeon(R) CPU E5-2650 v3	**113,392 ms**		
OpenMP CPUs per Task: 2	**69,873 ms**	**1.62x**	**1.62x**

Implementation Details: Sequential v.s. openMP v.s. GPU Parallelization

```
// start the clock
cudaEventRecord(start);

// call the kernel
mc_dao_call_shared(d_s.getData(), T, K, B, S0, sigma, mu, r, dt,
                   d_normals.getData(), N_STEPS, N_PATHS);

// End the clock
cudaEventRecord(stop);
cudaEventSynchronize(stop);
```

```
void mc_dao_call(
    float *d_s,
    float T,
    float K, // strike price
    float B, // barrier price
    float S0, // market price
    float sigma, // expected volatility per year
    float mu, // expected return per year
    float r, // risk-free rate
    float dt, // amount of time elapsing at each step
    float *d_normals,
    unsigned N_STEPS,
    unsigned N_PATHS)
{
    const unsigned BLOCK_SIZE = 16;
    const unsigned GRID_SIZE = ceil(float(N_PATHS) / float(BLOCK_SIZE));
    mc_kernel<<<GRID_SIZE, BLOCK_SIZE>>>(
        d_s, T, K, B, S0, sigma, mu, r, dt, d_normals, N_STEPS, N_PATHS);
}
```

Kernel:

```
{
    const unsigned tid = threadIdx.x;    // Thread index within block
    const unsigned bid = blockIdx.x;    // Block index
    const unsigned bsz = blockDim.x;    // Threads per block
    const unsigned s_idx = tid + bid * bsz; // Global thread ID

    if (s_idx < N_PATHS)
    {
        // Starting index for this thread's random numbers
        int base_idx = s_idx * N_STEPS;

        // Initialize current stock price
        float s_curr = S0;

        int n = 0;
        do
        {
            // Use random numbers for Euler discretisation
            s_curr = s_curr + mu * s_curr * dt + sigma * s_curr * d_normals[base_idx + n];
            n++;
        } while (n < N_STEPS && s_curr > B);

        // Compute payoff
        float payoff = (s_curr > K ? s_curr - K : 0.0);
        d_s[s_idx] = exp(-r * T) * payoff;
    }
}
```

Experimental Results

	Time	Step Speedup	Cumulative Speedup
Sequential CPU: Intel(R) Xeon(R) CPU E5-2650 v3	**113,392 ms**		
OpenMP CPUs per Task: 2	**69,873 ms**	**~1.6 x**	**~1.6 x**
GPU NVIDIA RTX 4000 Ada Block size: 16	**93 ms**	**~751 x**	**~1,219 x**

Shared Memory Optimization

- **Goal**: Accelerate using shared memory to reduce global memory latency.
- **Challenge**: Each thread requires 365 random floats, and for 1024 threads per block:

$$1024 \times 365 \times 4 \text{ bytes} = \mathbf{1.4MB}$$

However, we only have **48 KB** memory per block.

- Reduced block size to **16** threads. Shared memory fits, but the performance is even better without using shared memory.

```
    for (size_t i = 0; i < N_STEPS; ++i) {
        shared_mem[tid * N_STEPS + i] = d_normals[base_idx + i];
    }
    __syncthreads();
    if (s_idx < N_PATHS)
    {
        float s_curr = S0;
        int s = 0;
        do
        {
            s_curr = s_curr + mu * s_curr * dt + sigma * s_curr * shared_mem[tid * N_STEPS + s];
            s++;
        } while (s < N_STEPS && s_curr > B);
        float payoff = (s_curr > K ? s_curr - K : 0.0);
        d_s[s_idx] = exp(-r * T) * payoff;
    }
}
```

Time with shared memory

```
****************** PRICE *****************
Option Price (GPU): 8.49924
****************** TIME ******************
GPU Monte Carlo Computation: 53.5255 ms
****************** END *******************
```

Time without shared memory

```
****************** PRICE *****************
Option Price (GPU): 8.49924
****************** TIME ******************
GPU Monte Carlo Computation: 19.7755 ms
****************** END *******************
```

Shared Memory Optimization

- **Solution** : Coalescing
  - Optimizes global memory access by **aligning thread memory requests**.
  - Ensures threads in a warp **access consecutive memory addresses**.
- **How?**
  - Rearranged random numbers from path-first to **time-step-first order**.

```
__global__ void rearrange_random_numbers(
    float *d_normals_src,
    float *d_normals_dst,
    unsigned N_STEPS,
    unsigned N_PATHS)
{
    unsigned idx = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned total_elements = N_STEPS * N_PATHS;

    if (idx < total_elements)
    {
        unsigned path = idx / N_STEPS;
        unsigned step = idx % N_STEPS;
        unsigned new_idx = step * N_PATHS + path;
        d_normals_dst[new_idx] = d_normals_src[idx];
    }
}
```

Result: Enable us to use 1024 threads per block and speed up the computation 1.6x

```
********** PRICE COMPARISON **********
Option Price (GPU - Original Kernel): 8.50369
Option Price (GPU - Optimized Kernel): 8.49924
********** TIME COMPARISON **********
GPU Computation Time (Original Kernel): 18.6952 ms
GPU Computation Time (Optimized Kernel): 11.1092 ms
********** END **********
```

Takeaway

- Things you learn from the project in terms of “technical implementation”
  - Explored CUDA-specific optimizations like shared memory and global memory coalescing
- Things you learn from the project in terms of “teamwork and collaboration”
  - GitHub is really helpful in teamwork for version control and branch management, allowing each of us to have our own branch to optimize the project code based on our strengths
- Things you plan to complete by the final report due on 12/13
  - Allow customized inputs for the parameters, and add calculations for different barrier options
  - Visualization