High-Performance Computing in Finance

C++, CUDA, and Julia Implementations

Andre Oviedo, CFA

Overview

Project Summary

- **Goal**: Demonstrate HPC techniques for computational finance
- **Technologies**: C++, CUDA, Julia, OpenMP
- **Focus Areas**:
  - Option pricing algorithms
  - GPU acceleration
  - Parallel computing patterns
  - Performance optimization

Mathematical Foundations

Monte Carlo for Pi Estimation

Simple example of MC method

$$\pi \approx 4 \cdot \frac{\text{points inside circle}}{\text{total points}}$$

```
// Circle: x^2 + y^2 <= 1
int inside = 0;
for (int i = 0; i < N; i++) {
    float x = random(-1, 1);
    float y = random(-1, 1);
    if (x*x + y*y <= 1.0) inside++;
}
float pi_estimate = 4.0 * inside / N;
```

Demonstrates: Monte Carlo fundamentals

Matrix Multiplication

Optimized implementations for linear algebra

Applications in finance:

- Covariance matrix calculations
- Portfolio optimization
- Risk factor models
- PCA for dimensionality reduction

Techniques:

- Cache-aware blocking
- Loop unrolling
- SIMD vectorization
- GPU acceleration potential

Option Pricing: Analytical Methods

Black-Scholes-Merton Formula

Classical closed-form solution for European call options

Key formula:

$$C = S \cdot N(d_1) - K \cdot e^{-rT} \cdot N(d_2)$$

where:

$$d_1 = \frac{\ln(S/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}$$

Black-Scholes: C++ Implementation

```cpp
float norm_cdf(float x) {
    return 0.5f * (1.0f + erf(x / sqrt(2.0f)));
}

float european_call_option(float S, float K, float T,
                          float r, float sigma) {
    float d1 = (log(S/K) + (r + 0.5*sigma*sigma)*T)
               / (sigma*sqrt(T));
    float d2 = d1 - sigma * sqrt(T);
    return S * norm_cdf(d1) - K * exp(-r*T) * norm_cdf(d2);
}
```

Performance: ~1M evaluations in milliseconds

Monte Carlo Methods

Monte Carlo with Brownian Motion

Stochastic simulation approach for option pricing

Geometric Brownian Motion:

$$S_T = S_0 \cdot \exp\left((r - \frac{\sigma^2}{2})T + \sigma\sqrt{T} \cdot Z\right)$$

where $Z \sim N(0, 1)$

Option price:

$$C = e^{-rT} \cdot \mathbb{E}[\max(S_T - K, 0)]$$

Monte Carlo: Implementation

```
float value_option_with_mc(int M) {
    float S = 100.0f, K = 100.0f;
    float T = 1.0f, r = 0.05f, sigma = 0.2f;
    float c = 0.0f;

    for(int i = 0; i < M; i++) {
        float z = random_normal();  // N(0,1)
        float ST = S * exp((r - 0.5*sigma*sigma)*T
                           + sigma*sqrt(T)*z);
        float payoff = max(0.0f, ST - K);
        c += exp(-r*T) * payoff;
    }
    return c / M;
}
```

Convergence: Error $\propto 1/\sqrt{M}$

Discrete-Time Methods

Binomial Tree Model

Discrete-time approach with backward induction

- Build price tree: $S_{i,j} = S_0 \cdot u^j \cdot d^{i-j}$
- Terminal payoff: $\max(S_T - K, 0)$ for calls
- Backward induction: $V_i = e^{-r\Delta t}(p \cdot V_{i+1,up} + q \cdot V_{i+1,down})$

Advantages:

- Handles American options
- Intuitive visualization
- Flexible for exotic payoffs

Binomial Tree: Setup

```cpp
float priceEuropeanOption(float S0, float K, float T,
                          int N, float sigma, float r) {
    float dt = T / N;

    // Calculate up and down factors
    float u = exp((r - 0.5*sigma*sigma)*dt
              + sigma*sqrt(dt));
    float d = exp((r - 0.5*sigma*sigma)*dt
              - sigma*sqrt(dt));

    vector<float> optionValues(N + 1);

    // Terminal payoffs (parallelized)
    #pragma omp parallel for
    for (int j = 0; j <= N; ++j) {
        float ST = S0 * pow(u, j) * pow(d, N - j);
        optionValues[j] = max(0.0f, ST - K);
    }
```

Note: Terminal values computed in parallel

Binomial Tree: Backward Induction

```
    // Backward induction through the tree
    for (int i = N - 1; i >= 0; --i) {
        for (int j = 0; j <= i; ++j) {
            optionValues[j] = exp(-r*dt) *
                (0.5*optionValues[j+1] +
                 0.5*optionValues[j]);
        }
    }
    return optionValues[0];
}
```

Key insight: Backward induction cannot be parallelized due to data dependencies

CPU Parallel Computing

Multi-threaded Fork-Join

CPU parallelization using std::thread

Patterns demonstrated:

1. **Fork-Join**: Spawn threads, distribute work, synchronize
2. **Map-Reduce**: Parallel computation + aggregation

```
auto worker = [&](int stock_index) {
    const Stock& stock = stocks[stock_index];
    for (int i = stock_index; i < num_options;
         i += num_stocks) {
        results[i] = options[i].value(stock);
    }
};

// Fork
for (int i = 0; i < num_threads; ++i) {
    threads.emplace_back(worker, i);
}
```

Fork-Join: Reduction

```cpp
// Join
for (std::thread& t : threads) {
    t.join();
}

// Reduction: Calculate mean per stock
for (int s = 0; s < num_stocks; ++s) {
    double sum = 0.0;
    for (int i = s; i < num_options; i += num_stocks) {
        sum += results[i];
    }
    mean_per_stock[s] = sum / count;
}
```

Speedup: Near-linear with number of cores (4-8$\times$)

GPU Computing with CUDA

Monte Carlo CUDA

Massive parallelization on GPU

Key optimizations:

- Each thread computes one option price
- Common Random Numbers (CRN) for variance reduction
- cuRAND for parallel random number generation
- 1M+ paths per option

Monte Carlo CUDA: Kernel Signature

```
__global__ void monteCarloCallPrice_kernel_CRN(
    float* d_optionPrices,
    const float* d_Z,  // Precomputed random numbers
    float S0,
    const float* d_K_batch,
    const float* d_T_batch,
    const float* d_r_batch,
    const float* d_v_batch,
    int numOptions,
    int numPaths)
```

Each thread prices one option using all Monte Carlo paths

Monte Carlo CUDA: Kernel Body

```
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < numOptions) {
        float K = d_K_batch[idx];
        float T = d_T_batch[idx];
        float r = d_r_batch[idx];
        float v = d_v_batch[idx];

        float sumPayoffs = 0.0f;
        for (int i = 0; i < numPaths; ++i) {
            float ST = S0 * expf((r - 0.5f*v*v)*T
                                   + v*sqrtf(T)*d_Z[i]);
            sumPayoffs += fmaxf(ST - K, 0.0f);
        }
        d_optionPrices[idx] =
            expf(-r*T) * sumPayoffs / numPaths;
    }
}
```

Greeks Computation with CUDA

Sensitivity analysis for risk management

Greeks measure option price sensitivity:

- **Delta** (Δ): $\frac{\partial C}{\partial S}$ - price sensitivity to spot
- **Gamma** (Γ): $\frac{\partial^2 C}{\partial S^2}$ - delta sensitivity
- **Vega**: $\frac{\partial C}{\partial \sigma}$ - volatility sensitivity
- **Rho**: $\frac{\partial C}{\partial r}$ - rate sensitivity
- **Theta** (Θ): $\frac{\partial C}{\partial t}$ - time decay

Greeks: Analytical Formulas

```
__host__ __device__ void compute_all_greeks(
    float S0, float K, float T, float v, float r,
    float& call_price, float& delta_call,
    float& gamma, float& vega,
    float& rho_call, float& theta_call) {

    float d1 = (log(S0/K) + (r + 0.5*v*v)*T)
               / (v*sqrt(T));
    float d2 = d1 - v * sqrt(T);
    float nd1 = cdf_normal(d1);
    float nd2 = cdf_normal(d2);
    float pd1 = pdf_normal(d1);

    call_price = S0 * nd1 - K * exp(-r*T) * nd2;
    delta_call = nd1;
    gamma = pd1 / (S0 * v * sqrt(T));
```

Greeks: CUDA Kernel

```
    vega = S0 * sqrt(T) * pd1 * 0.01;
    rho_call = K * T * exp(-r*T) * nd2 * 0.01;
    theta_call = (-(S0*v*pd1)/(2*sqrt(T))
                 - r*K*exp(-r*T)*nd2) / 365.0;
}

__global__ void greeks_kernel(
    float* S0, float* K, float* T,
    float* sigma, float* r,
    float* prices, float* deltas, float* gammas,
    float* vegas, float* rhos, float* thetas,
    int num_options) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_options) {
        compute_all_greeks(S0[idx], K[idx], T[idx],
            sigma[idx], r[idx], prices[idx],
            deltas[idx], gammas[idx], vegas[idx],
            rhos[idx], thetas[idx]);
    }
}
```

Julia Implementations

Derivatives Visualization

```
# Call and Put options
Call(S, K) = max(S - K, 0)
Put(S, K) = max(K - S, 0)

# 3D surface plot
surface(S_range, K_range, (S, K) -> Call(S, K),
        xlabel="Stock Price (S)",
        ylabel="Strike Price (K)",
        zlabel="Call Option Value")
```

Benefits:

- High-level syntax
- Built-in mathematical operations
- Easy visualization with Plots.jl
- Comparable performance to C++

Fixed Income Instruments

Bond pricing fundamentals:

```
# Bond cash flows
t = 1:0.5:15
FV = 100
c = 0.05
C = c/2 * FV  # Semi-annual coupon

# Present value
r = 0.05  # Yield
PV = sum(C * exp(-r*t_i) for t_i in t) +
     FV * exp(-r*T)
```

Julia advantages: Financial math notation translates directly to code

Performance Comparison

Technology Stack Summary

Approach	**Best For**	**Speedup**
Sequential C++	Baseline	$1\times$
OpenMP	Shared memory parallel	4–8$\times$
std::thread	Custom parallelism	4–8$\times$
CUDA	Embarrassingly parallel	100–1000$\times$
Julia	Rapid prototyping	$\sim 1\times$ (with JIT)

Key insight: Choose tool based on problem structure and hardware

Key Takeaways

Lessons Learned

1. **Algorithm matters**: Black-Scholes (ms) vs Monte Carlo (seconds)
2. **GPU excel at**: Independent simulations, no branching
3. **Memory is critical**: Cache-aware algorithms, data transfer costs
4. **Parallel patterns**: Fork-join, map-reduce are fundamental
5. **Variance reduction**: CRN dramatically improves MC convergence

Best Practices

- Profile before optimizing
- Start with correct sequential code
- Understand memory hierarchy
- Use appropriate precision (float vs double)
- Leverage existing libraries (cuRAND, MKL)
- Validate against analytical solutions

Conclusion

Repository Overview

Complete HPC toolkit for computational finance

- Multiple pricing models (analytical, numerical, stochastic)
- GPU acceleration with CUDA
- Multi-threading patterns
- Cross-language comparison (C++, Julia)
- Production-ready code structure

Next steps: Extend to American options, stochastic volatility, multi-asset derivatives