Accelerating Binomial Tree Methods for American Option Pricing: Parallel Algorithm Analysis on CPU and GPU Architectures

Team Member: Li Cao (licao)

November 17, 2025

SUMMARY:

This project involves the design, implementation, and deep performance analysis of parallel algorithms for the binomial option pricing model, a computationally intensive problem in finance. I will implement and compare several parallelization strategies, including a multi-threaded CPU implementation (OpenMP) and multiple CUDA-based GPU implementations using different algorithmic approaches. The goal is to analyze the trade-offs between different parallel programming models and memory-hierarchy-aware optimizations for solving a dynamic programming problem with inherent data dependencies. Key parallel programming challenges include: managing fine-grained barrier synchronization, optimizing SIMT execution in the presence of control-flow divergence, exploiting memory hierarchy through shared memory and coalescing, and load-balancing declining parallelism across heterogeneous CPU-GPU resources.

BACKGROUND:

An option is a financial contract that gives the buyer the right, but not the obligation, to buy (a call option) or sell (a put option) an underlying asset at a fixed strike price by a certain expiration date. A European option can only be exercised on its expiration date. This project focuses on the more complex American option, which can be exercised at any time up to and including expiration. Because no closed-form analytic solution (like the famous Black-Scholes formula) exists for American options, they must be priced using numerical methods.

The binomial options pricing model (BOPM), specifically the Cox-Ross-Rubinstein (CRR) method, is a fundamental tool for this task. It is a lattice method that models the problem in discrete time steps, making it ideal for handling the early-exercise decisions of American options. The algorithm consists of two distinct computational phases:

1. **Forward Phase (Lattice Construction):** First, a recombinant lattice (a directed acyclic graph) of all possible future asset prices is constructed. The time from now ($t = 0$) to the option's expiration ($t = T$) is divided into N discrete time steps. At each step, the asset price S is assumed to move to one of two new values: an up move (S_u) or a down move (S_d). Because an up move followed by a down move results in the same price as a down move followed by an up move, the tree is recombinant. This is a critical property: the number of unique nodes at time step n is $n + 1$, and the total number of nodes in the entire lattice is $(N + 1)(N + 2)/2$, resulting in $O(N^2)$ total nodes, not an exponential $O(2^N)$. This $O(N^2)$ graph structure is the data set we must process.
2. **Backward Phase (Backward Induction):** The option's value is then calculated using backward

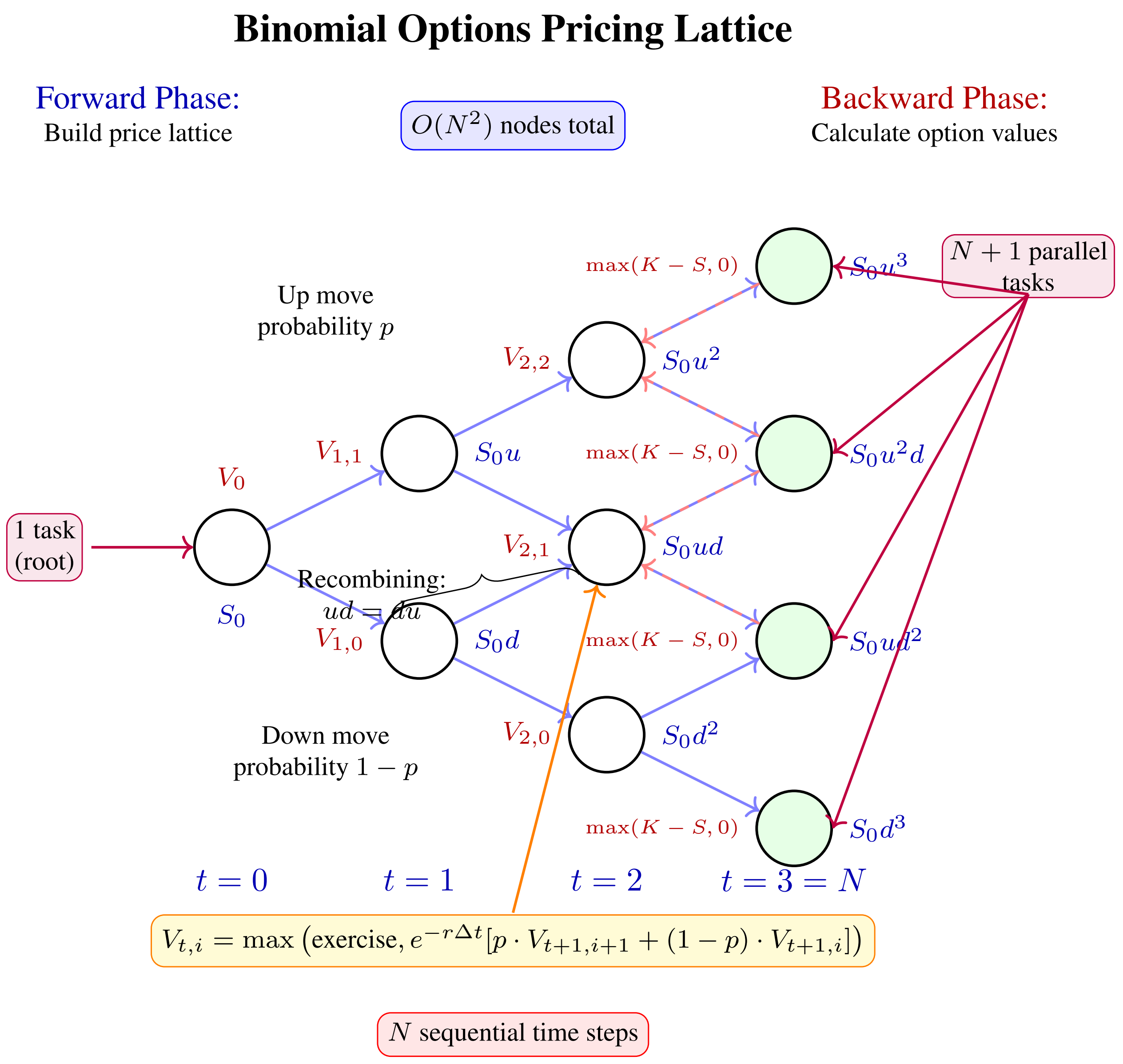

Figure 1: Illustration of the binomial options pricing lattice for $N = 3$ time steps, showing the forward phase (blue solid arrows) for price lattice construction and the backward phase (red dashed arrows) for option value calculation via backward induction.

induction. The algorithm starts at the expiration date ($t = N$, the leaves of the graph), where the option's value at all $N+1$ final nodes is known (e.g., for a put option, $V = \max(K - S_T, 0)$). It then works backward, one time step at a time, toward the present ($t = 0$, the root). The value of any non-leaf node is calculated as the discounted expected value of its two child nodes from the next time step. For a European option, this is a simple linear recurrence: $V_{\text{node}} = e^{-r\Delta t}(p \cdot V_{\text{up}} + (1-p) \cdot V_{\text{down}})$.

However, for an American option, the calculation at each node is a non-linear recurrence defined by the early-exercise check:

$$V_{\text{node}} = \max(V_{\text{exercise}}, V_{\text{hold}})$$

where V_{hold} is the discounted expected value of the two subsequent child nodes. Here, V_{exercise} is the intrinsic value of exercising immediately (e.g., $\max(K - S_{\text{node}}, 0)$ for a put). This $\max()$ operation is the core reason the problem is computationally expensive and cannot be solved analytically. A serial implementation has a time complexity of $O(N^2)$ (as it must visit all $O(N^2)$ nodes), making it computationally expensive for the high-accuracy (large N) calculations required by financial institutions.

THE CHALLENGE:

The BOPM is a challenging parallel programming problem because it is a dynamic programming task, not an embarrassingly parallel one. The challenge lies in mapping this $O(N^2)$ algorithm with its inherent sequential dependencies and non-linear logic onto modern parallel architectures.

To quantify the problem: for $N = 10,000$ time steps (typical for production-quality pricing requiring fine time discretization), the algorithm must compute approximately 50 million nodes. Each node requires 2 floating-point reads, 1 write, 1 exponential, 2 multiplications, 1 addition, and 1 max operation—approximately 10-15 FLOPs of compute paired with 12 bytes of memory traffic, resulting in a very low arithmetic intensity of only $\sim$1 FLOP/byte.

- **Data Dependency:** The calculation at time t is fully dependent on the results from time $t + 1$. This creates a sequential bottleneck along the time axis, preventing a simple parallel-for over all $O(N^2)$ nodes. Unlike embarrassingly parallel Monte Carlo methods, backward induction fundamentally requires processing the lattice in strict reverse-chronological order.
- **Data Structure & Memory Access:** The natural $O(N^2)$ lattice structure is inefficient. A key optimization is to collapse the lattice into an $O(N)$ 1D array representing the current time-step's wavefront of nodes. Parallelizing this 1D array update presents new challenges:
  - **CPU:** Requires careful synchronization (e.g., barriers) between each time step. For $N = 10,000$, this means 10,000 synchronization points, each potentially adding 1-10 microseconds of overhead.
  - **GPU:** A naive wavefront implementation (one kernel per time step) will be severely memory-bandwidth-bound. For $N = 10,000$, each of 10,000 kernel launches must read $\sim 8N$ bytes (two float arrays) and write $\sim 4N$ bytes back to global memory, totaling $\sim$1.2 GB of memory traffic across all launches for minimal computation. With kernel launch overhead of $\sim$5-20 microseconds each, this adds up to 50-200 milliseconds of pure overhead.
  - **Coalescing:** Accessing the 1D array $V[i]$ and $V[i + 1]$ to compute the new $V[i]$ must be done carefully to ensure coalesced memory access from global memory. Uncoalesced access can reduce effective bandwidth by 75% or more.
- **GPU Architectural Mapping (SIMT):**
  - **Warp Divergence:** The $\max(\text{exercise}, \text{hold})$ check is a conditional branch. If threads within a single warp are processing different nodes (some exercising, some holding), the warp will diverge, serializing execution and hurting performance. For options near the money, divergence could be as high as 50%, effectively halving warp throughput.
  - **Shared Memory:** A high-performance tiled or block-based algorithm aims to overcome the global memory bottleneck by loading a tile of the lattice into fast on-chip shared memory. This requires explicit data management, double-buffering within shared memory, and `__syncthreads()` barriers—a pattern similar to tiled matrix multiplication. Optimal tile size must balance shared memory capacity (48 KB per SM) against occupancy.

- **Declining Parallelism:** The wavefront of $N + 1$ nodes at time $t = N$ shrinks to a single node at $t = 0$. For $N = 10,000$, parallelism declines from 10,001 parallel tasks to just 2 tasks—a 5000× reduction. As the computation approaches the root, the amount of available parallelism decreases, leading to severe underutilization of the GPU (potentially $< 1\%$ occupancy in the final steps). This ”parallelism cliff” is the fundamental motivation for the hybrid CPU-GPU approach.

RESOURCES:

- **Computers:** I will use the GHC cluster machines for the OpenMP implementation and the CUDA implementations. For the stretch goals, I will use the PSC machines to explore MPI.
- **Starter Code:** I will write the program from scratch in C++ to have full control over the algorithm and data structures.
- **References:** I will base my implementations on algorithms described in academic papers and NVIDIA's SDK documentation. See the References section below for specific sources.

GOALS AND DELIVERABLES:

The project's goal is to produce a highly-optimized parallel pricer and, more importantly, a detailed analysis report comparing the strategies. The deliverable will be a set of well-commented implementations and a final report (with graphs) analyzing speedup, bottlenecks (memory, compute, divergence), and scalability.

- **50% Goal (Baseline):**
  - Implement a correct, serial single-threaded CPU version of the BOPM for American options.
  - Implement a baseline parallel CPU version using OpenMP (e.g., `omp parallel for` at each time step, with implicit barriers).
  - Validate both implementations for correctness against known financial examples.
- **75% Goal (Naive GPU):**
  - Implement the wavefront parallel algorithm in CUDA. This will use N sequential kernel launches, one for each time step.
  - Each kernel will parallelize the computation across the nodes of a single time step, reading from and writing to global memory.
  - Profile this implementation to establish a GPU baseline and demonstrate/quantify the global memory bandwidth bottleneck.
- **100% Goal (Optimized Tiled GPU):**
  - Implement an optimized tiled or block-based algorithm in a single CUDA kernel.
  - This kernel will use thread blocks to load tiles of the 1D array from global memory into on-chip shared memory in a coalesced-friendly way.
  - It will perform the backward induction for multiple steps within shared memory, using double-buffering and `__syncthreads()`, to fuse many steps of the Wavefront algorithm and minimize global memory access.

  - **Deliverable:** A detailed performance comparison (Speedup, Memory Bandwidth, Occupancy) of the Serial, OpenMP, Wavefront, and Tiled implementations as a function of N (number of time steps).

- **125% Goal (Hybrid Algorithm & Deeper Analysis):**
  - Address the declining parallelism problem. Implement a hybrid CPU-GPU algorithm that uses the fast CUDA Tiled kernel for the wide part of the tree (large t) and hands off the narrow part (small t) to the optimized OpenMP implementation, avoiding GPU underutilization and kernel launch overhead.
  - Use `nvprof` / Nsight Compute to deeply analyze the 100% kernel, measuring warp divergence from the $\max()$ check and tuning block/tile size.

- **150% Goal (Multi-Node / Multi-GPU Batching):**
  - Explore scalability beyond a single machine. Using MPI (and/or NCCL), I will implement a high-throughput batch processing system.
  - This embarrassingly parallel problem (e.g., pricing a portfolio of 1,000,000 independent options for risk analysis) will distribute the work across multiple nodes. Each node will use the optimized 125% goal implementation to price its subset of options.
  - The goal is to demonstrate scalability with the number of nodes for a high-throughput risk-calculation workload.

Fallback Goals (If Progress is Slower Than Expected):

In the event that implementation challenges or time constraints prevent achieving the planned goals, the following fallback positions will still constitute valuable project outcomes:

- **60% Fallback Goal:**
  - Working CUDA Wavefront implementation with basic profiling analysis demonstrating memory bandwidth bottlenecks.
  - Detailed documentation of the bottleneck characterization using `nvprof` metrics (e.g., memory throughput, kernel launch overhead).
  - Comparative analysis of serial vs. OpenMP vs. naive GPU performance with speedup graphs.

- **80% Fallback Goal:**
  - Partially optimized Tiled CUDA kernel, even if not fully optimized.
  - Thorough documentation of what optimization strategies were attempted (e.g., different tile sizes, shared memory configurations, coalescing patterns).
  - Analysis of what worked, what didn't work, and why, with specific performance measurements.
  - This ”negative results” documentation would still demonstrate mastery of CUDA optimization principles and GPU architecture understanding.

- **90% Fallback Goal:**
  - Fully working and optimized Tiled CUDA kernel achieving significant speedup over Wavefront implementation.

  - Comprehensive performance analysis including memory bandwidth utilization, occupancy, and speedup graphs.
  - If the hybrid CPU-GPU algorithm (125% goal) proves too complex, instead provide a detailed theoretical analysis and design proposal for the hybrid approach, including performance projections based on measured CPU and GPU performance characteristics.

PLATFORM CHOICE:

- **CPU / OpenMP:** The GHC machines provide multi-core CPUs, which are a natural fit for a baseline shared-memory parallel implementation using OpenMP. This allows for a direct comparison with the GPU.
- **GPU / CUDA:** The latedays cluster GPUs are the ideal platform. The BOPM challenge maps directly to the CUDA programming model's concepts: managing memory hierarchies (global vs. shared), thread block synchronization (`__syncthreads()`), and SIMT execution (warp divergence).
- **Cluster / MPI:** PSC / cluster machines are necessary for the 150% goal, which explores distributed-memory parallelism (MPI) for a high-throughput problem, a common pattern in large-scale scientific and financial computing.

SCHEDULE:

- **Week 1 (11/17 - 11/23):** 50% Goal. Finalize serial C++ implementation and validate correctness. Implement and debug the OpenMP parallel-for version.
- **Week 2 (11/24 - 11/30):** 75% Goal & Milestone. Implement the ”Wavefront” (naive) CUDA algorithm. Begin performance profiling. **(11/27: Submit Milestone Report)** detailing the correct serial/OpenMP versions and the working Wavefront-GPU version, with preliminary analysis of its bottlenecks.
- **Week 3 (12/1 - 12/7):** 100% Goal. This is the main implementation week. Focus entirely on designing, implementing, and debugging the ”Tiled” shared-memory CUDA kernel. This will be the most complex part of the project.
- **Week 4 (12/8 - 12/15):** 125%/150% Goals & Final Report.
  - (12/8 - 12/10): Implement the 125% (Hybrid CPU-GPU) goal and/or the 150% (MPI Batching) goal.
  - (12/11 - 12/13): Final performance runs, generate all graphs for the report, create poster.
  - (12/14 - 12/15): Write final report. **(12/15: Final Report Due)**.

REFERENCES:

1. Cox, J. C., Ross, S. A., & Rubinstein, M. (1979). Option pricing: A simplified approach. *Journal of Financial Economics*, 7(3), 229-263. [Original CRR binomial model]

2. Broadie, M., & Detemple, J. (1996). American option valuation: New bounds, approximations, and a comparison of existing methods. *Review of Financial Studies*, 9(4), 1211-1250. [Benchmark values for validation]

3. NVIDIA Corporation. (2025). *CUDA C++ Programming Guide*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/` [Shared memory optimization patterns, coalescing guidelines]