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

Li Cao

December 8, 2025

Abstract

This project investigates parallel algorithmic strategies to accelerate the Binomial Options Pricing Model (BOPM), a financial algorithm constrained by strict sequential dependencies. All experiments were conducted on the CMU GHC machines (8-core Intel CPU, NVIDIA RTX 2080 GPU). Three parallel paradigms were systematically explored to address specific bottlenecks: Shared Memory (OpenMP), Distributed Memory (MPI), and SIMT (CUDA). For multi-core execution, scheduling overheads were analyzed, determining that static scheduling outperforms dynamic assignment for this uniform workload. On the GPU, 10 distinct CUDA kernels were developed, utilizing techniques such as deep temporal tiling, register-level warp shuffles, cooperative groups, and persistent global barriers. This characterization revealed two distinct performance regimes: a latency-bound regime for small N and a throughput-bound regime for large N. The final solution is a Hybrid CPU-GPU Adaptive Pipeline that dynamically switches between a "Warp Per Block" kernel (for low latency), a "Shared Memory Tiling" kernel (for high throughput), and CPU execution (for zero-latency tails). This hybrid approach achieved a 384x speedup over serial execution and 9.35x over optimized OpenMP. Additionally, the distributed MPI Master-Worker implementation demonstrated near-linear strong scaling (7.6x on 8 ranks), demonstrating its suitability for pricing large portfolios of options for risk management purposes.

Contents

1 INTRODUCTION AND 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. While European options can only be exercised on the 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 Black-Scholes formula) exists for American options due to the optimal early-exercise boundary, 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 models the asset price evolution as a discrete-time lattice. The algorithm consists of two distinct computational phases:

1.1 Phase 1: Forward Lattice Construction

First, a recombinant lattice of all possible future asset prices is constructed. The time from now ($t = 0$) to expiration ($t = T$) is divided into N discrete time steps. At each step, the asset price S moves to either S_u (up) or S_d (down). Because the tree is recombinant ($Sud = Sdu$), the number of unique nodes at time step n is $n + 1$, and the total number of nodes is $(N + 1)(N + 2)/2$, or $O(N^2)$.

1.2 Phase 2: Backward Induction

The option's value is calculated using backward induction, starting from the expiration date ($t = N$) where the value is known (e.g., $\max(K - S_T, 0)$ for a put). The algorithm works backward to $t = 0$. The value at any node (t, i) is determined by the specific recurrence for American options:

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

where V_{hold} is the discounted expected value of the future nodes:

$$V_{\text{hold}} = e^{-r\Delta t}[p \cdot V_{t+1,i+1} + (1 - p) \cdot V_{t+1,i}]$$

and V_{exercise} is the intrinsic value of exercising immediately (e.g., $K - S_{t,i}$). This non-linear $\max()$ operation is the core computational challenge.

1.3 Inputs, Outputs, and Data Structures

To satisfy the strict latency requirements of financial computing, the data layout is optimized for cache locality.

- **Inputs:** The model takes 6 parameters: Spot Price (S_0), Strike Price (K), Time to Expiration (T), Risk-free Rate (r), Volatility (σ), and Time Steps (N).
- **Output:** A single floating-point number representing the fair option value V_0.
- **Data Structures:**
  - **Lattice Storage:** Instead of a 2D matrix ($O(N^2)$), a 1D flat array ($O(N)$) of double precision floats is utilized. This fits comfortably in the L1/L2 cache for most practical N.
  - **Access Pattern:** The array is accessed sequentially. In the parallel GPU implementation, this 1D structure is partitioned across threads or loaded into Shared Memory tiles.

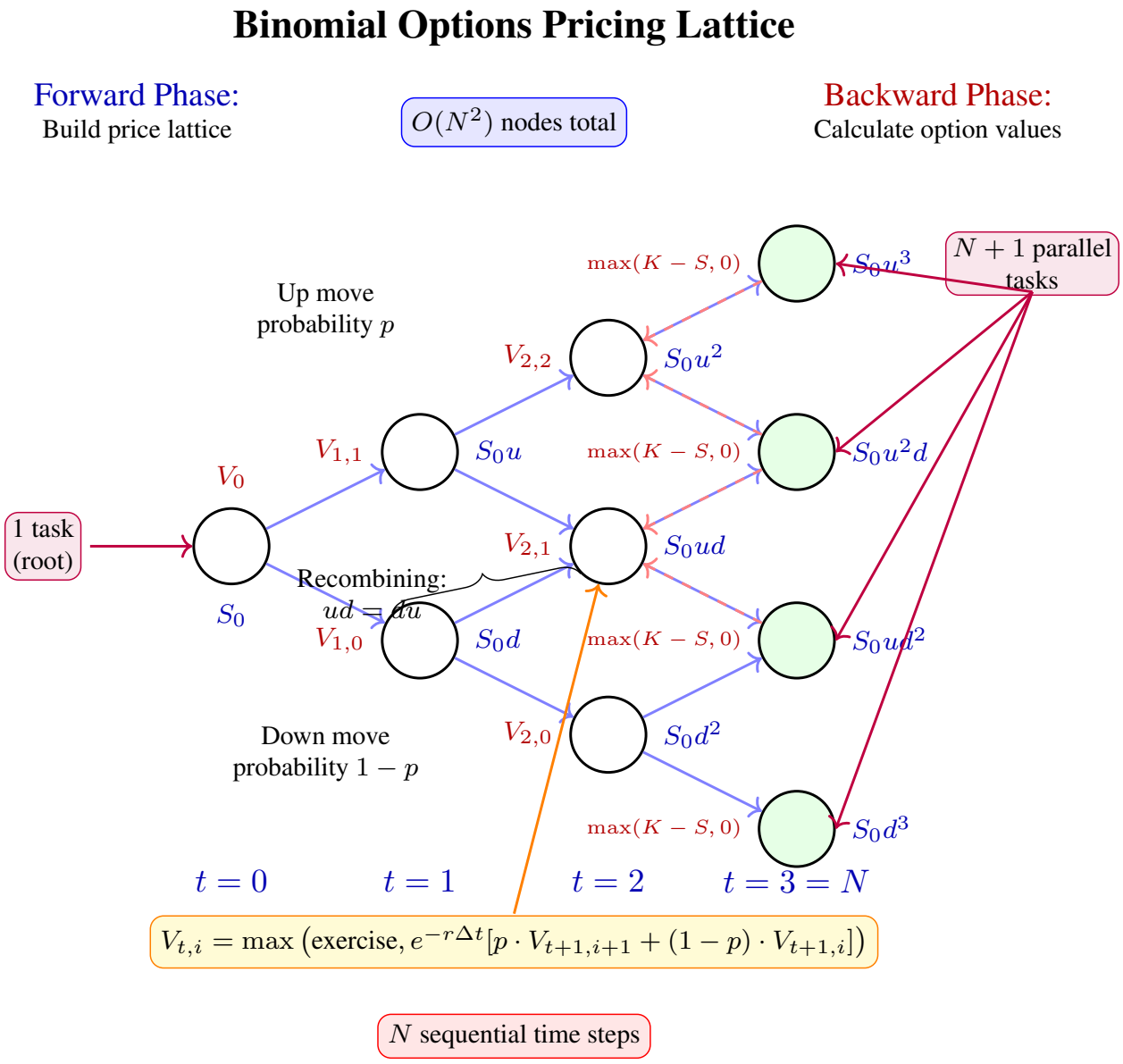


Figure 1: Illustration of the binomial options pricing lattice for $N = 3$ time steps, showing the forward phase (blue solid arrows) for price lattice construction and the backward phase (red dashed arrows) for option value calculation via backward induction.

1.4 The Computational Challenge

The BOPM is a challenging parallel programming problem because it is a dynamic programming task with strict dependencies, not an embarrassingly parallel one.

- **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 parallelization across time steps.
- **High-Bandwidth, Low-Compute:** Each node requires only 10 FLOPs (multiplies, add, max) but requires reading 2 doubles and writing 1 double (24 bytes). This results in a very low arithmetic intensity (< 0.5 FLOP/byte), making the problem strictly memory-bandwidth bound.

- **GPU Architectural Mapping:**
  - **Warp Divergence:** The $\max(\text{exercise}, \text{hold})$ check introduces conditional branching. If threads in a warp disagree on the outcome, execution is serialized.
  - **Memory Coalescing:** Accessing the lattice efficiently requires careful mapping of threads to memory addresses to ensure coalesced transactions.
- **Declining Parallelism (The "Cliff"):** The wavefront of $N + 1$ nodes at $t = N$ shrinks linearly to 1 node at $t = 0$. For a large simulation with $N = 100,000$, the GPU is fully saturated at the leaves. However, as the computation proceeds towards the root, the available parallelism drops below the hardware capacity of the GPU. At $t < 5,000$, the GPU becomes latency-bound rather than throughput-bound, leading to severe underutilization. This "parallelism cliff" necessitates the hybrid approach explored in this report.

2 TESTING AND VALIDATION METHODOLOGY

To ensure the robustness and correctness of the optimizations, a comprehensive testing suite was developed. This suite serves two distinct purposes: algorithmic verification of mathematical properties and numerical validation against high-precision benchmarks.

2.1 Validation of Mathematical Invariants

A series of unit tests was implemented to verify that the option pricer adheres to fundamental financial theory. These tests ensure that the model behaves consistent with the partial derivatives of the option pricing formula (the "Greeks") and arbitrage constraints.

The specific properties verified are detailed in Table 1.

Property Tested	**Mathematical Assertion**
Positivity	$V_{put} \geq 0$
Upper Bound	$V_{put} \leq K$ (Strike Price)
Intrinsic Lower Bound	$V_{put} \geq \max(K - S, 0)$
American Premium	$V_{American} \geq V_{European}$ (Early exercise value ≥ 0)
Spot Monotonicity	$\frac{\partial V_{put}}{\partial S} < 0$ (Price decreases as asset price increases)
Strike Monotonicity	$\frac{\partial V_{put}}{\partial K} > 0$ (Price increases as strike price increases)
Volatility Monotonicity	$\frac{\partial V}{\partial \sigma} > 0$ (Price increases with volatility, Vega > 0)
Time Monotonicity	$\frac{\partial V}{\partial T} > 0$ (Price increases with time to maturity, Theta < 0)
Convergence	$\lim_{N \to \infty} \lvert V_N - V_{2N} \rvert = 0$ (Cauchy convergence)

Table 1: Mathematical properties of American Put Options verified by the correctness suite.

2.2 High-Precision Benchmark Generation

To validate the numerical accuracy of the implementations, a "ground truth" dataset was generated using the reference serial implementation.

- **Reference Resolution:** All benchmarks were generated using $N = 20,000$ time steps. This provides sufficient resolution to serve as a ground truth for the lower-resolution performance tests ($N = 1,000 \dots 5,000$).
- **Test Case Diversity:** The benchmark suite covers 65 distinct market scenarios to prevent overfitting to a single regime. These include:
  - **Moneyness:** At-the-money ($S = K$), In-the-money ($S < K$), and Out-of-the-money ($S > K$).
  - **Volatility Regimes:** Low volatility ($\sigma = 0.1$) to High volatility ($\sigma = 0.6$).
  - **Maturity:** Short-term ($T = 0.25$) to Long-term ($T = 5.0$) options.
  - **Interest Rates:** Includes cases with $r = 0\%$, $r = 5\%$, and $r = 10\%$.

During development, any optimized kernel (CPU or GPU) was required to pass regression tests against these values within a relative error tolerance of 10^{-4}.

3 SERIAL IMPLEMENTATION

A reference CPU version was implemented to serve as the baseline for correctness and performance. This implementation calculates the price of an American option using the standard BOPM logic detailed in Section 1.

3.1 Algorithmic Structure

The core of the serial pricing engine is the backward induction phase. The algorithm proceeds as follows:

1. **Initialization:** A single vector V of size $N + 1$ is allocated. It is initialized with the terminal option values at maturity ($t = N$) using the payoff function $\max(K - S_N, 0)$ for a put option.
2. **Backward Iteration:** The main loop iterates backwards in time from $t = N - 1$ down to 0.
3. **Node Calculation:** At each time step t, the inner loop iterates through nodes $i = 0 \dots t$. For each node, it:
   - Calculates the underlying asset price $S_{t,i}$.
   - Computes the **continuation value** (holding the option) as the discounted expectation of the next step's values: $V_{hold} = e^{-r\Delta t}[p \cdot V_{t+1,i+1} + (1 - p) \cdot V_{t+1,i}]$.
   - Computes the **exercise value** (exercising now): $K - S_{t,i}$.
   - Updates the node value: $V_{t,i} = \max(V_{hold}, V_{exercise})$.

3.2 Memory Optimization and Implementation

A naive implementation might store the full $O(N^2)$ lattice. However, since the value at time t depends only on values at time $t + 1$, space complexity is optimized to $O(N)$ using a single `std::vector<double>`.

Crucially, in the serial implementation, this vector can be updated in-place safely. Because the iteration proceeds for i from 0 to t, and the value $V_{t,i}$ depends on $V_{t+1,i}$ (which corresponds to index i in the array before update) and $V_{t+1,i+1}$ (which corresponds to index $i+1$), the values being read have not yet been overwritten by the current step's loop.

The C++ core logic is shown below:

```cpp
// Backward induction from t = N-1 down to t = 0
for (int t = opt.N - 1; t >= 0; --t) {
    for (int i = 0; i <= t; ++i) {
        // Calculate stock price at node (t, i)
        double S = opt.S0 * std::pow(params.u, i)
                          * std::pow(params.d, t - i);

        // Continuation value
        double V_hold = params.discount *
            (params.p * V[i + 1] + (1.0 - params.p) * V[i]);

        // Early exercise check
        double V_exercise = (opt.isCall) ?
            callPayoff(S, opt.K) : putPayoff(S, opt.K);

        // Update in-place
        V[i] = std::max(V_hold, V_exercise);
    }
}
```

Listing 1: Core Serial Backward Induction Kernel

This implementation performs exactly $N(N+1)/2$ node updates, resulting in $O(N^2)$ computational complexity, but runs efficiently in L1 cache due to the small memory footprint.

4 OPENMP IMPLEMENTATION

To establish a multi-threaded CPU baseline, a parallel version was implemented using OpenMP. This implementation focuses on exploiting loop-level parallelism across the spatial domain (nodes) at each time step.

4.1 Parallelization Strategy

1. **Geometric Decomposition:** At any time step t, there are $t+1$ independent node values to compute. This 1D array of nodes is partitioned into contiguous chunks, assigning each chunk to a thread. This optimizes for spatial locality, as adjacent threads work on adjacent data, minimizing false sharing and maximizing cache line utilization.

2. **Double Buffering:** Unlike the serial version which updates a single vector in-place, the parallel version requires two vectors, `V_in` (reading valid data from $t+1$) and `V_out` (writing new data for t). This eliminates race conditions where a thread might overwrite a value $V[i+1]$ that its neighbor still needs to read.

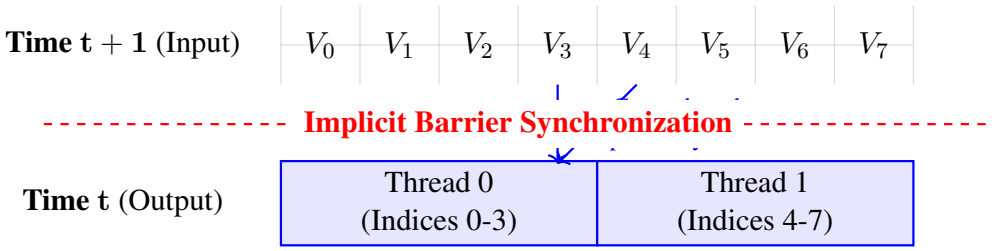


Figure 2: Visualization of the 1D Geometric Decomposition strategy. Threads process contiguous chunks of the lattice. The blue arrows indicate data dependencies ($V_{t,i}$ depends on $V_{t+1,i}$ and $V_{t+1,i+1}$). Note that Thread 0 needs a value (V_4) that belongs to Thread 1's chunk, necessitating the global barrier (red dashed line) to ensure all inputs are valid before processing.

4.2 Optimization: Hoisting Parallel Region

Initial benchmarks revealed a catastrophic performance regression when scaling to 8 threads. This performance degradation resulted in execution times orders of magnitude slower than the serial implementation (e.g., at $N = 10,000$, 8-thread execution took 56 seconds while 4-thread execution took 60 milliseconds).

This was traced to the overhead of creating and destroying a parallel thread team at every time step (N times total). The cost of thread pool management for 8 threads was significant accumulating to a massive penalty over thousands of time steps.

To resolve this, the implementation was refactored to hoist the parallel region. The thread team is created once, and the work sharing construct (`#pragma omp for`) is used repeatedly inside the time loop.

```
// Create thread team once
#pragma omp parallel shared(V_in, V_out, ...)
{
    // Step 1: Initialize (Parallelized)
    #pragma omp for schedule(static)
    for (int i = 0; i <= opt.N; ++i) { ... }

    // Step 2: Time Stepping Loop
    for (int t = opt.N - 1; t >= 0; --t) {
        // Work sharing construct (implicit barrier at end)
        #pragma omp for schedule(static)
        for (int i = 0; i <= t; ++i) {
            // Safe to read V_in, write to V_out
            double val = ...;
            V_out[i] = val;
        }

        // Pointer swap by single thread
        // 'single' construct has implicit barrier
        #pragma omp single
        { std::swap(V_in, V_out); }
    }
}
```

Listing 2: Optimized OpenMP Implementation with Parallel Region Hoisting

4.3 Results and Analysis

The optimized OpenMP implementation demonstrates significant speedup over the serial baseline and scales effectively with thread count up to the hardware limit.

N (time steps)	Serial (ms)	OMP 1-Th (ms)	OMP 4-Th (ms)	OMP 8-Th (ms)	Speedup (8-Th vs Serial)
1,000	14.02	1.72	0.95	1.44	**9.7x**
5,000	346.35	41.59	15.29	12.87	**26.9x**
10,000	1,385.79	167.64	56.86	47.90	**28.9x**
50,000	34,593.34	4,027.89	1,232.18	796.17	**43.4x**
100,000	137,801.96	15,836.30	4,838.87	3,353.96	**41.1x**

Table 2: Performance comparison of Serial vs. OpenMP implementation across varying number of time steps N and thread counts.

As shown in Table 2, the parallel implementation achieves over 40x speedup for large lattices. The scaling behavior indicates that even for small N ($N = 1,000$), the hoisted implementation avoids the "parallelism cliff" that typically plagues fine-grained tasks, maintaining a speedup of nearly 10x. The slight regression at 8 threads for $N = 1,000$ (1.44ms vs 0.95ms) is attributed to the remaining minimal synchronization cost relative to the extremely small workload.

4.4 Scheduling Strategy Analysis

To justify the use of static scheduling, an alternative implementation using dynamic scheduling (chunk size 1024) was benchmarked.

N	Static (8-Th) [ms]	Dynamic (8-Th) [ms]	Difference
1,000	1.44	2.56	**+77.8% (Slower)**
5,000	12.87	21.52	**+67.2% (Slower)**
10,000	47.90	57.61	**+20.3% (Slower)**
50,000	796.17	773.99	**-2.8% (Faster)**
100,000	3,353.96	3,197.74	**-4.7% (Faster)**

Table 3: Performance comparison of Static vs. Dynamic scheduling (8 threads).

Table 3 confirms that Static scheduling is superior for small to medium problem sizes. Because the binomial tree workload is perfectly predictable and uniform within each time step (every node performs identical operations), the overhead of the dynamic scheduler (queue management) outweighs any load balancing benefits. For small $N = 1,000$, dynamic scheduling is nearly 2x slower. While dynamic scheduling exhibits a marginal performance edge (4%) when N is large ($N > 50,000$)—potentially by mitigating system-induced latency spikes—its significant overhead at smaller scales confirms that static scheduling remains the superior, robust choice for this workload.

5 GPU IMPLEMENTATION

The primary challenge in accelerating the Binomial Option Pricing Model on the GPU is its inherent sequential dependency structure. Unlike Monte Carlo simulations which are "embarrassingly parallel," the

binomial lattice requires the values at time step t to be fully computed before step $t-1$ can begin. This creates a synchronization bottleneck that strictly limits parallelism.

To conquer this challenge, a suite of 10 distinct CUDA kernels was developed and characterized. This exhaustive exploration was necessary because the optimal strategy changes radically depending on the problem size (N):

- **Small N (Latency-Bound):** The overhead of launching kernels and synchronizing threads dominates. Strategies must focus on fusing steps and minimizing barrier latency.
- **Large N (Throughput-Bound):** The sheer volume of data dominates. Strategies must focus on maximizing memory bandwidth and instruction throughput via tiling and caching.

The following sections detail the evolution of the optimization strategy, effectively telling the story of moving data closer to the compute units: from **Global Memory** (Phase 1), to **Shared Memory** (Phase 2), to **Registers** (Phase 3), and finally to a **Hybrid Architecture** (Phase 4).

5.1 Challenges of Binomial Tree Parallelization

The primary challenge is the tight data dependency pattern ($V_{t,i}$ depends on $V_{t+1,i}$ and $V_{t+1,i+1}$). This restricts parallelism in two ways:

1. **Forward Dependency:** A global synchronization barrier is strictly required between time steps $t+1$ and t.
2. **Declining Parallelism:** The number of active nodes decreases linearly from $N+1$ to 1. For the "tail" of the computation ($t < 1000$), the GPU is severely underutilized (occupancy $< 1\%$) while paying the full fixed cost of kernel launches.

5.2 Phase 1: The Baseline

This phase established the baseline performance and identified the memory bandwidth bottleneck.

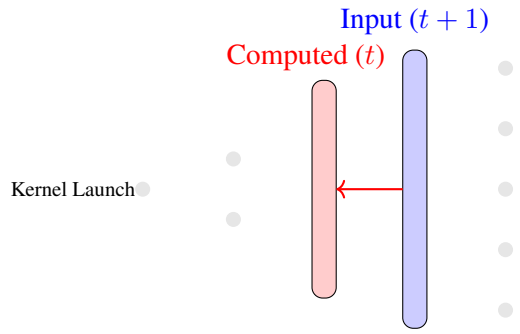
5.2.1 Naive Wavefront (Baseline Implementation)

File: `src/cuda/cuda_wavefront.cu`

Algorithm Description The Naive Wavefront implementation represents the most direct translation of the iterative dynamic programming algorithm to the GPU. Since the value of a node $V(t,i)$ depends on values at time $t+1$ ($V(t+1,i)$ and $V(t+1,i+1)$), there is a strict sequential dependency between time steps. This implementation enforces this dependency by launching a separate CUDA kernel for each time step t, iterating backwards from $N-1$ to 0.

Parallel Strategy

- **Decomposition:** At each time step t, all $t+1$ nodes are independent and can be computed in parallel.
- **Mapping:** One GPU thread is assigned to each node i.
- **Memory Model:** The algorithm relies exclusively on Global Memory. To prevent race conditions (reading a value that is being overwritten), **Double Buffering** (ping-pong) is employed. Two arrays, `V_in` and `V_out`, are allocated. In step t, the kernel reads from `V_in` (holding data for $t+1$) and writes to `V_out` (for time t). The pointers are swapped on the host after every kernel launch.



Wavefront Execution:
A separate kernel is launched for each vertical column. Global memory barriers (kernel termination) enforce synchronization between columns.

Figure 3: Wavefront Strategy: Strict time-step synchronization via kernel launches.

```
__global__ void wavefrontStepKernel(const double *V_in, double *V_out,
                                    const double *u_pow, const double *d_pow,
                                    int t, ...) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i <= t) {
    // 1. Read Global Memory (High Latency)
    double v_up = V_in[i + 1];
    double v_down = V_in[i];

    // 2. Compute
    double V_hold = discount * (p * v_up + (1.0 - p) * v_down);
    V_out[i] = fmax(V_hold, V_exercise);
  }
}
```

Listing 3: Wavefront Kernel (One thread per node)

Performance Analysis This implementation serves as the performance floor.

1. **Kernel Launch Overhead:** For $N = 10,000$, 10,000 separate kernel launches are issued. With a minimum driver latency of $\approx 5\mu s$ per launch, the theoretical minimum time is $50ms$. For small $N = 1,000$, this overhead constitutes over 90% of the runtime.
2. **Memory Wall:** Each thread performs 2 Global Loads (16 bytes) and 1 Global Store (8 bytes) for just a handful of arithmetic operations (Arithmetic Intensity ≈ 0.3 FLOPs/Byte). The kernel is strictly memory bandwidth bound.

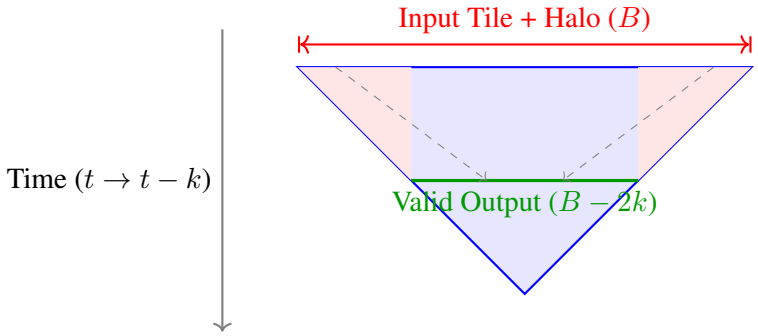
Result: $2.86ms$ ($N = 1k$), $42.4s$ ($N = 1M$). Slowest implementation.

5.2.2 Tiled Global Memory

File: (`cuda_tiled.cu`)

- **Algorithm:** To reduce Global Memory traffic, this kernel fuses K time steps (e.g., $K = 16$) into a single kernel launch. A thread block loads a "tile" of width B (e.g., 256) into Shared Memory. It then computes K steps entirely in on-chip memory. Due to the dependency cone, the valid output width shrinks by 1 at each step, resulting in a valid output tile of width $B - K$.
- **Parallel Strategy:** Block-iterative tiling. Reduced global synchronization frequency by a factor of K.

Performance Analysis Actual execution results demonstrate a performance improvement inversely proportional to problem size. At $N = 1,000$, the tiled implementation achieves 1.07ms, a 2.67x speedup over the baseline Wavefront kernel ($2.86ms$). This gain stems primarily from the K-fold reduction in global synchronization points (kernel launches). However, at $N = 1,000,000$, the speedup diminishes to just 1.12x ($37.9s$ vs $42.4s$). This limited scalability at large N confirms that while tiling successfully reduces global memory transactions, the **"halo overhead"**—the redundant computation of invalid boundary zones required to satisfy dependencies—becomes a new bottleneck that neutralizes the bandwidth savings.



Tiled Execution Model:
1. **Load:** Thread Block loads a wide tile B into Shared Mem.
2. **Compute:** Iterates k steps on-chip.
3. **Halo Loss:** Dependencies consume boundary values.
4. **Result:** A smaller valid output tile is written back.

Figure 4: Visualizing the Tiled Execution Strategy: The Red regions represent the "Halo" overhead—computations performed effectively just to satisfy dependencies for the inner Green valid region.

5.3 Phase 2: Throughput Optimization

This phase focused on maximizing arithmetic intensity and occupancy for large problem sizes ($N > 10,000$). The goal was to beat the memory bandwidth limit by keeping data on-chip for longer.

5.3.1 Time Parallel / Deep Temporal Tiling

File: `src/cuda/cuda_time_parallel.cu`

Algorithm Description The "Time Parallel" strategy takes the concept of Tiling to its logical extreme. If fusing $K = 16$ steps reduces bandwidth, why not fuse $K = 256$ steps? This kernel attempts **Deep Temporal Tiling**. It defines a very large "super-tile" (e.g., Width = 2048) in Shared Memory and attempts to compute 256 time steps in a single kernel launch. To facilitate this massive computation per block, it also employs Thread Coarsening ($VT = 4$), assigning multiple elements to each thread to hide instruction latency.

Parallel Strategy

- **Massive Shared Memory Usage:** A tile of width 2048 with double buffering requires $2048 \times 8\text{bytes} \times 2 = 32\text{KB}$ of Shared Memory per block.
- **Deep Fusion:** The kernel loops for $STEPS\_L1 = 256$ iterations internally.
- **Thread Coarsening:** A block of 256 threads processes $256 \times 4 = 1024$ elements (active region shifts).

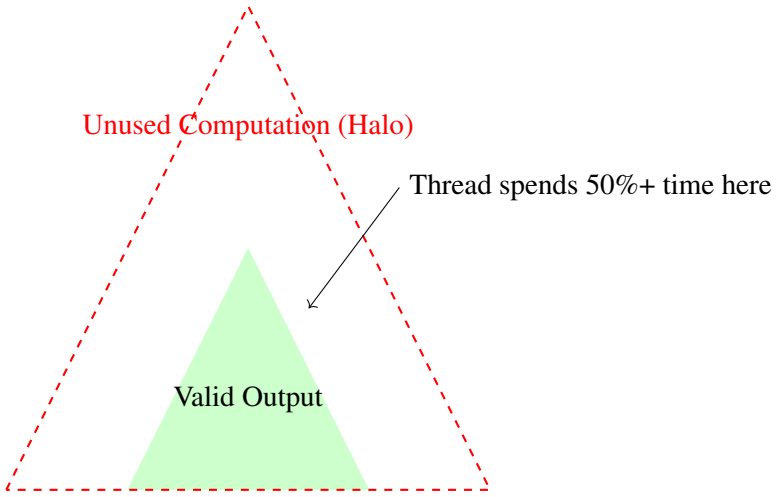


Figure 5: The "Halo Explosion" in Time Parallel Tiling.

```
// Configuration: BLOCK_SIZE=256, VT=4, STEPS=256
__global__ void tiledKernelTimeParallel(...) {
    // 1. Load Huge Tile (Width=2048)
    #pragma unroll
    for(int i=0; i<VT; ++i) V_s_in[tid*VT + i] = ...;

    // 2. Deep Temporal Loop (256 Steps)
    for (int k = 0; k < steps; ++k) {
        #pragma unroll
        for (int i = 0; i < VT; ++i) {
            // Dependency Check & Compute
            if (valid) {
                V_s_out[...] = ...; // Compute using neighbors
            }
        }
        __syncthreads(); // Barrier every step
        std::swap(V_s_in, V_s_out);
    }
}
```

Listing 4: Deep Temporal Tiling Kernel Structure

Performance Analysis Despite the aggressive design, this kernel did not outperform simpler tiling strategies.

1. **Halo Explosion:** To output a valid region after 256 steps, a halo of 256 elements on each side must be loaded. For small block sizes, the ratio of (Halo / Valid Data) becomes immense, meaning specific threads spend their entire execution time computing data that is discarded.
2. **Occupancy Collapse:** The high Shared Memory usage (32KB/block) and Register Pressure (from the complex loop body and unrolling) strictly bounded the number of active warps per SM. The GPU could not hide memory latency.

Result: $2.52ms$ at $N = 1k$. While faster than the baseline ($2.86ms$), it was significantly more complex than standard Tiling ($1.07ms$) and suffered from diminishing returns.

5.3.2 Cooperative Multi-Warp

File: `src/cuda/cuda_cooperative_multi_warp.cu`

Algorithm Description The Cooperative Multi-Warp kernel attempts to solve the "Halo" problem by making an entire Thread Block (256 threads, 8 warps) behave as a single coherent unit processing a massive tile (Width = 2048). Instead of discarding halo data at warp boundaries, warps "publish" their boundary data to Shared Memory so their neighbors can read it. This allows for communication across the entire block without redundant Global Memory loads.

Parallel Strategy

- **Intra-Warp Communication:** Threads within a warp use `__shfl_down_sync` to pass data from $tid + 1$ to tid. This is register-fast.
- **Inter-Warp Communication (The Bottleneck):** Thread 31 of Warp W needs data from Thread 0 of Warp $W + 1$. Since warps cannot shuffle between each other, Warp $W + 1$ writes its first element to a shared memory buffer `warp_first_elem[8]`.
- **Barrier Synchronization:** To ensure Warp W reads the correct value written by Warp $W + 1$ for the current step, a global `__syncthreads()` is required at every single time step.

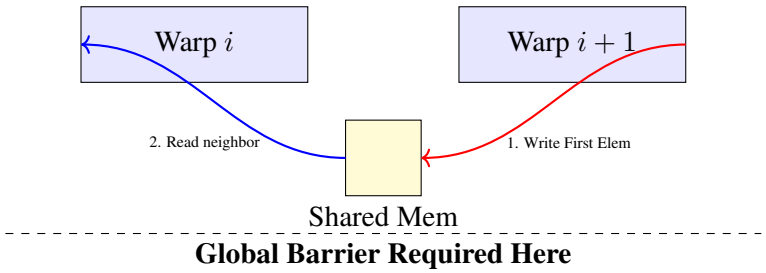


Figure 6: Cooperative Inter-Warp Communication Cost.

```
// 1. Publish Boundary Data to Shared Mem
if (lane_id == 0) {
    warp_first_elem[warp_id] = r_val[0];
}
__syncthreads(); // Barrier 1: Wait for publication

// 2. Read Neighbor Data
double next_val;
if (lane_id == 31) {
    // Read from next warp's published value
    next_val = (warp_id < 7) ? warp_first_elem[warp_id + 1] : 0.0;
} else {
    // Read from neighbor lane via shuffle
    next_val = __shfl_down_sync(0xffffffff, r_val[0], 1);
}

// 3. Compute
r_next[i] = fmax(discount * (p * next_val + ...), V_exercise);
__syncthreads(); // Barrier 2: Wait for computation before next step
```

Listing 5: Complex Synchronization Dance in Cooperative Kernel

Performance Analysis This kernel illustrates the cost of synchronization.

1. **Barrier Overhead:** The algorithm requires $2 \times$ `__syncthreads()` per time step. For a block of 256 threads, this stalls the entire pipeline.
2. **Complexity vs Payoff:** The cost of coordinating 8 warps exceeded the savings from reducing halo loads.

Result: $5.18ms$ at $N = 1k$. This is significantly slower than even the baseline Wavefront approach ($2.86ms$) for small N, and non-competitive at large N. It demonstrates that intra-block synchronization is expensive.

5.3.3 Warp Shuffle Tiling (Register-Based Optimization)

File: `src/cuda/cuda_warp_shuffle_tiling.cu`

Algorithm Description To eliminate the latency of Shared Memory (approx. 20-30 cycles), this implementation keeps the active tile data entirely in Registers (approx. 1 cycle). Communication between threads is handled via **Warp Shuffle Intrinsics** (`__shfl_down_sync`), which allow threads within the same warp to read each other's registers directly. Shared Memory is only used for the "Inter-Warp Halo" (passing data between Warp i and Warp $i + 1$).

Parallel Strategy

- **Register Tiling:** Each thread maintains a local array `double r_val[VT]` in registers.
- **Shuffle Exchange:** At each step, a thread needs value $i + 1$. If $i + 1$ is in another thread (neighbor lane), it pulls it using `__shfl_down_sync`.
- **Minimal Shared Memory:** Only 1 double per warp is written to Shared Memory (to bridge the gap between Lane 31 and Lane 0 of the next warp).

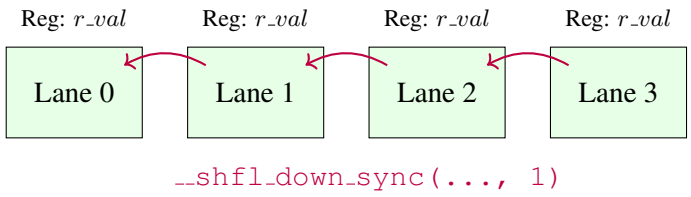


Figure 7: Zero-Latency Register Shuffle Exchange.

```
// 1. Publish first value to Shared Mem if Lane is 0
if (lane_id == 0) warp_halo[warp_id] = r_val[0];
__syncthreads();

// 2. Get neighbor value
double neighbor_val;
if (lane_id == 31) {
    // Read from next warp's halo in Shared Mem
    neighbor_val = (warp_id < WARPS - 1) ? warp_halo[warp_id + 1] : 0.0;
} else {
    // Direct register-to-register transfer from next lane
    neighbor_val = __shfl_down_sync(0xffffffff, r_val[0], 1);
```

```
}
// 3. Compute next step using neighbor_val
...
```

Listing 6: Warp Shuffle Logic for Halo Exchange

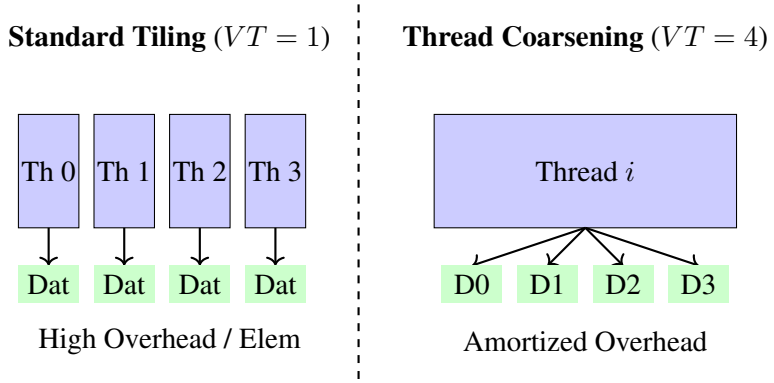
Performance Analysis

- **Benefit:** Eliminates Shared Memory bank conflicts and latency for 96% of the data exchanges.
- **Limitation (Register Pressure):** Holding the tile in registers limits the tile depth K and thread coarsening factor VT. Compilers will spill to Local Memory (L1/L2 cache) if register usage per thread is too high, negating the benefit.
- **Result:** $2.58ms$ at $N = 1k$. Comparable to the Wavefront baseline for small N, but scales better for medium sizes. However, it cannot beat Shared Memory Tiling for very large N due to lower total occupancy.

5.3.4 Shared Memory Tiling + Thread Coarsening (The Phase Winner)

File: `src/cuda/cuda_shared_mem_tiling.cu`

Algorithm Description This kernel represents the "Sweet Spot" in the Throughput optimization phase. It combines the benefits of Tiling (locality) with Thread Coarsening. Instead of assigning one thread to one node ($VT = 1$), each thread is assigned to process a "strip" of $VT = 4$ nodes. This technique is also known as "Register Blocking" or "ILP Optimization" because it allows the compiler to unroll loops and keep data in registers for longer, amortizing the cost of instruction fetch and index calculation.



Left: Each thread processes 1 element. Overhead (scheduling, branch) is paid per element.
Right: One thread processes 4 elements. Overhead is paid once for 4 ops.

Figure 8: Visualizing Thread Coarsening: Increasing Arithmetic Intensity by assigning multiple data elements to a single thread.

Parallel Strategy

1. **Balanced Coarsened Tile:** A block of 256 threads loads a tile of width $256 \times 4 = 1024$ into Shared Memory.

2. **Decoupled from Deep Fusion:** Crucially, unlike the "Time Parallel" kernel (which suffered from halo overhead), this strategy does not attempt extreme temporal fusion ($K = 16$ steps vs $K = 256$). This prevents the "Halo Explosion".

3. **The Advantage of Coarsening:** By keeping the temporal depth shallow but increasing the workload per thread ($VT = 4$), the benefits of high Arithmetic Intensity (amortized instruction overhead) are realized without suffering the penalty of massive halo redundancy or low occupancy.

```
// Configuration: VT=4. Each thread computes 4 values.
// Outer loop: Iterate through time steps 'k'
for (int k = 0; k < steps; ++k) {
    #pragma unroll
    for (int i = 0; i < VT; ++i) {
        // Dependency: V_s[local_idx + 1] and V_s[local_idx]
        int loc = tid * VT + i;
        if (loc < valid_range) {
           double V_hold = disc * (p * V_s_in[loc+1] + (1.0-p) * V_s_in[loc]);
           V_s_out[loc] = fmax(V_hold, V_exercise);
        }
    }
    __syncthreads(); // Barrier after computing one full step for the tile
    std::swap(V_s_in, V_s_out);
}
```

Listing 7: Thread Coarsened Kernel Core Loop

5.4 Phase 3: Latency Optimization

This phase targeted the "Kernel Launch Overhead" bottleneck critical for small N ($N < 5000$).

5.4.1 Persistent Global Barrier (Latency Optimization)

File: `src/cuda/cuda_persistent_global_barrier.cu`

1. **Algorithm:** Standard CUDA kernels cannot synchronize across blocks. This implementation launches a single "Persistent" kernel that loops through all time steps. It uses a Global Software Barrier to enforce synchronization: blocks atomically increment a global counter, and the last block to arrive resets it and flips a "sense" flag, releasing all other blocks.

2. **Goal:** To eliminate the $\approx 5\mu s$ overhead of launching a new kernel for every time step.

3. **Outcome: Counter-productive.** The latency of the global atomic round-trip proved higher than the hardware kernel launch it replaced. $2.59ms$ at $N = 1k$ (vs $0.89ms$ for the eventual winner).

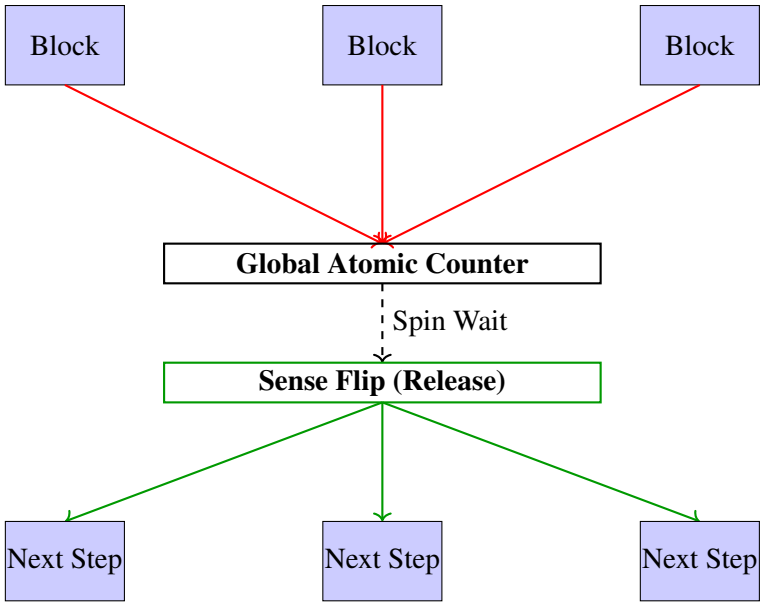


Figure 9: Persistent Kernel Synchronization: Blocks spin-wait on a global flag. The contention on the single atomic counter creates a new bottleneck.

```
__device__ void global_sync(int *count, volatile int *sense) {
  __syncthreads(); // 1. Wait for all threads in block
  if (threadIdx.x == 0) {
    bool my_sense = (*sense != 0);
    // 2. Atomic Increment
    if (atomicAdd(count, 1) == num_blocks - 1) {
      *count = 0;          // Reset
      *sense = !my_sense; // 3. Release: Flip Sense
    } else {
      // 4. Spin Wait
      while ((*sense != 0) == my_sense) { /* busy wait */ }
    }
  }
  __syncthreads(); // 5. Sync before proceeding
}
```

Listing 8: Global Sense-Reversing Barrier

5.4.2 Independent Multi-Warp (Occupancy Optimization)

File: `src/cuda/cuda_independent_multi_warp.cu`

Algorithm Description In standard CUDA programming, a Thread Block is the smallest unit of dispatch, and synchronization barriers (`__syncthreads()`) halt all threads in the block. This kernel introduces the concept of **Virtual Blocks**:

1. **Warp Packing:** Each Warp (32 threads) is treated as a completely independent execution unit.
2. **Physical Block:** A standard CUDA block of 256 threads acts as a container for $256/32 = 8$ independent "Virtual Blocks".

3. **Implicit Sync:** Since threads within a warp are execution-synchronous, all `__syncthreads()` are removed. Synchronization is implicit, and communication happens via register shuffles.

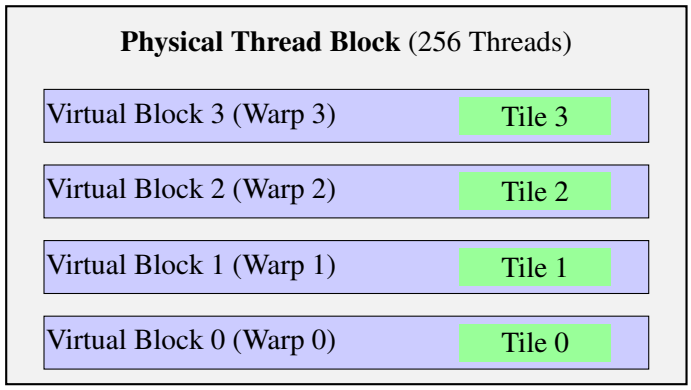


Figure 10: Warp Packing: 8 independent tasks are packed into a single hardware block to maximize Occupancy (Active Warps / SM).

```
// Each Warp acts as a "Virtual Block"
int tid = threadIdx.x;      // 0..255
int warp_id = tid / 32;     // 0..7
int lane_id = tid % 32;     // 0..31

// Calculate Virtual Block ID for data indexing
// A single physical block launches 8 independent tasks
int virtual_block_id = blockIdx.x * WARPS_PER_BLOCK + warp_id;

// No __syncthreads() needed!
// Warps execute independently.
```

Listing 9: Virtual Block ID Calculation

Performance Analysis

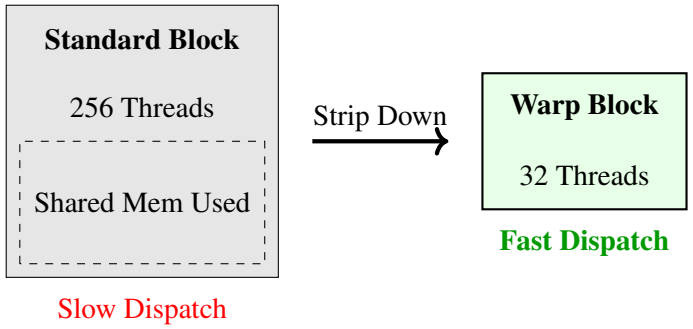
- **Outcome:** $2.77ms$. This strategy did not improve performance over the baseline latency-optimized kernels.
- **Why?** For small N ($N < 1000$), the GPU is Latency Bound, not Throughput Bound. The problem is that there isn't enough total work to hide the pipeline latency. Increasing Occupancy (the number of active warps) helps hide memory latency, but performance is limited by the serial dependency chain of the time steps.

5.4.3 Warp Per Block (The Phase Winner)

File: `src/cuda/cuda_warp_per_block.cu`

Algorithm Description This kernel represents the "Lean and Mean" philosophy. Recognizing that for small N, the hardware scheduler's overhead to dispatch large thread blocks (allocating shared memory, barriers, etc.) dominates the runtime, the kernel is stripped down to the bare metal.

1. **Block Size = 32:** The block size is matched exactly to the hardware warp size.
2. **No Shared Memory:** All data stays in registers.
3. **No Synchronization:** Execution within a warp is lock-step synchronous by definition. All `__syncthreads()` are removed.



By reducing the block to a single warp and removing shared memory, the GPU's Gigathread Engine is allowed to dispatch thousands of these tiny blocks with minimal latency.

Figure 11: Warp Per Block Strategy: Eliminating overhead (barriers, shared memory allocation) to minimize dispatch latency.

```
// 1. Launch Bounds Hint: Tell compiler max threads is 32
__global__ void __launch_bounds__(32) tiledKernelV4(...) {
    // 2. No Shared Memory Allocation
    // 3. Register Tiling
    double r_val[VT];

    // 4. Implicit Sync Loop
    for (int k = 0; k < steps; ++k) {
        // Exchange data via Shuffle (No __syncthreads)
        double neighbor = __shfl_down_sync(0xffffffff, r_val[0], 1);

        // Compute...
        // Update Registers...
    }
}
```

Listing 10: Warp Per Block "Lean" Kernel

Performance Analysis

- **Result:** 0.89ms at $N = 1k$. This is >3x faster than the Baseline Wavefront ($2.86ms$) and beats the "Persistent" approach ($2.59ms$).
- **Why?** For small problem sizes, the runtime is dominated by the fixed cost of "waking up" the GPU and scheduling blocks. This kernel minimizes that cost.

5.5 Phase 4: Hybrid Architecture

File: `src/cuda/cuda_hybrid_cpu_gpu.cu`

The final Hybrid implementation is not merely a choice between CPU and GPU; it is an Adaptive Pipeline that dynamically switches between the best kernels described in previous phases as the problem size N shrinks during execution. The binomial tree structure means the effective N (number of time steps remaining) decreases in every iteration. A kernel that is optimal for $N = 1,000,000$ (Shared Mem Tiling) is suboptimal for $N = 1,000$ (Warp Per Block).

5.5.1 Adaptive Kernel Switching

The solver monitors the current time step t and transitions through three execution regimes. The specific thresholds below were determined through careful tuning on the target GHC machines to find the optimal crossover points where one kernel overtakes another:

1. **High Throughput Regime ($N > 80,000$):** When parallelism is abundant, the solver utilizes the **Shared Memory Tiling** kernel (Phase 2 Winner). This maximizes memory bandwidth utilization.
2. **Low Latency Regime ($500 < N \leq 80,000$):** As the tree narrows, the overhead of large thread blocks becomes prohibitive. The solver switches to the Warp Per Block kernel (Phase 3 Winner), which minimizes dispatch overhead.
3. **Serial Regime ($N \leq 500$):** Finally, when kernel launch overhead ($\approx 5\mu s$) dominates the actual computation time, the solver copies the intermediate curve V to the Host (CPU) via 'cudaMemcpy' and finishes the remaining steps using the serial CPU algorithm. Even with the PCI-E transfer cost, the CPU's lower latency wins at these small sizes.

```
// Phase 1: Throughput Optimized (Shared Mem Tiling)
while (current_t > LARGE_N_THRESHOLD) {
    run_SharedMemTiling_Kernel(...);
    current_t -= STEPS;
}

// Phase 2: Latency Optimized (Warp Per Block)
while (current_t > CPU_THRESHOLD) { // Threshold = 500
    run_WarpPerBlock_Kernel(...);
    current_t -= STEPS;
}

// Phase 3: Zero Latency (CPU)
cudaMemcpy(h_V, d_V_in, ...);
for (int t = current_t - 1; t >= 0; --t) {
    // Finish simply on Host CPU
    serial_computation(...);
}
```

Listing 11: Adaptive Hybrid Dispatch Loop

Performance Conclusion ("The Ultimate Winner") This hybrid approach yields the monotonically best performance across the entire range of N.

- **Small N ($N = 1k$):** 0.63ms. By switching to CPU for the tail, it outperforms the best pure-GPU strategy (Warp Per Block: $0.89ms$) by another 30%.

- **Large N ($N = 100k$):** 358ms. It matches the massive throughput of the Shared Memory Tiling kernel.
- **Speedup:** Up to 250x vs Serial CPU for large N.

5.6 Performance Analysis: The Suite of 10 Kernels

This project explored 10 distinct parallel strategies. To organize the findings, the kernels and their key characteristics are summarized in Table 4, followed by a quantitative performance comparison in Table 5.

Table 4: Summary of CUDA Kernel Strategies

Kernel Name	Strategy / Key Characteristic
1. Wavefront (Baseline)	Naive diagonal parallelism. High global memory traffic (uncoalesced).
2. Tiled (Global Mem)	Logical block mapping. Unoptimized memory access.
3. **Shared Mem Tiling**	Phase 2 Winner. Uses Thread Coarsening ($VT = 4$) and Shared Memory to reduce global bandwidth pressure.
4. Warp Shuffle Tiling	Register-level communication via `__shfl_down_sync`. Limited by register pressure.
5. Time Parallel	Deep Temporal Tiling. Limited by "Halo Explosion" (excessive redundant computation).
6. Cooperative Multi-Warp	Inter-warp synchronization using shared memory. Performance bottlenecked by barrier overhead.
7. Persistent Barrier	Global software barrier with atomic counters. Limited by atomic contention ($2.59ms$).
8. Independent Multi-Warp	Packing multiple independent wavefronts into one block. Limited by Latency-bound constraints at small N.
9. **Warp Per Block**	Phase 3 Winner. "Lean and Mean" strategy. Block Size = 32. Zero synchronization overhead. Best pure-GPU latency ($0.89ms$).
10. **Hybrid CPU-GPU**	The Ultimate Winner. Adaptive pipeline. Uses Shared Mem for Large N, Warp Per Block for Medium N, and CPU for Small N.

5.7 Profiling Insights: Mechanisms of Acceleration

Detailed profiling using Nsight Compute (NCU) revealed the precise micro-architectural reasons for the performance differences observed in the Throughput Optimization and Latency Optimization phases.

5.7.1 Kernel Launch Overhead Reduction

For the baseline Wavefront algorithm, the sheer number of kernel launches proved to be the primary bottleneck. Profiling data confirms that the optimized strategies succeeded by drastically reducing CPU-GPU interaction:

- **Wavefront (Baseline):** Required $\approx 20,000$ kernel launches for $N = 10,000$ steps. With a hardware overhead of $\approx 5\mu s$ per launch, the GPU spent a significant portion of execution time idle, waiting for commands.
- **Optimized Kernels (Shared Mem Tiling / Warp Per Block):** Achieved a 16x reduction in launch count ($\approx 1,250$ launches) via temporal tiling (fusing multiple time steps into a single kernel call).

This reduction was the key factor in shifting the workload from a Latency-bound to a Compute-bound regime.

- **Persistent Barrier:** Achieved the theoretical minimum of ≈ 4 launches. However, despite eliminating launch overhead, the high cost of global software synchronization (spin-waiting on atomic counters) outweighed the benefits, resulting in net performance degradation.

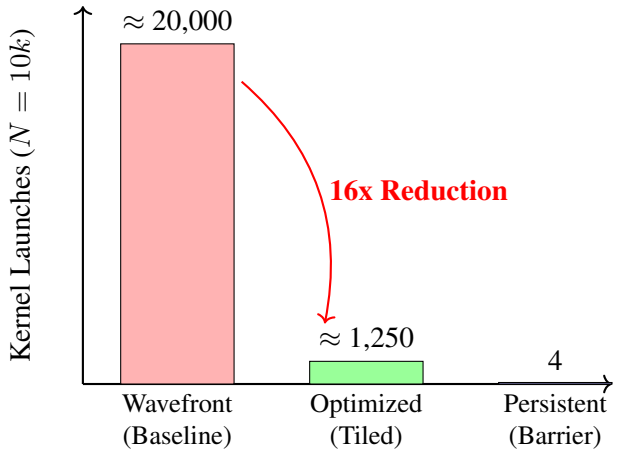


Figure 12: Impact of Algorithm Strategy on Kernel Launch Count. The dramatic reduction in launches explains the performance shift from Latency-bound to Compute-bound.

5.7.2 Occupancy and Utilization

- **Shared Memory Tiling (Throughput Winner):** Maintained an average SM Occupancy of $\approx 25\%$. While lower than the theoretical peak, this represents a "Sweet Spot": it allows for maximal Instruction-Level Parallelism (ILP) per thread via Thread Coarsening ($VT = 4$) without causing register spilling (which would occur if higher occupancy were attempted). The performance gain comes from higher Arithmetic Intensity, not just raw parallelism.
- **Warp Per Block (Latency Winner):** Exhibited a very low average occupancy of $\approx 6\%$. This confirms that its performance dominance for small N creates a paradox: it is the fastest kernel despite being the least utilized. It sacrifices throughput (Occupancy) to minimize dispatch latency, "racing to idle" faster than any other strategy.

5.8 Results

Table 5 presents the execution time (in milliseconds) for selected representative kernels across a wide range of N.

Table 5: CUDA Kernels Performance Results (Time in ms)

Kernel Strategy	**N=1k**	**N=5k**	**N=10k**	**N=50k**	**N=100k**	**N=500k**	**N=1M**
1. Wavefront (Baseline)	2.86	14.36	28.93	187.88	551.44	11,156	42,425
2. Tiled (Global Mem)	1.07	5.10	10.15	122.13	428.51	9,621	37,877
3. **Shared Mem Tiling (Phase 2)**	2.63	13.32	26.72	144.04	405.99	**7,281**	**28,332**
4. Warp Shuffle Tiling	2.58	14.05	28.34	152.35	440.37	7,938	30,082
5. Time Parallel (Deep Tiling)	2.52	12.40	24.71	155.96	465.39	9,327	36,146
6. Cooperative Multi Warp	5.18	28.51	58.24	295.50	633.17	11,737	50,216
7. Persistent Barrier	2.59	22.27	47.41	248.91	522.64	7,956	29,958
8. Independent Multi Warp	2.78	14.91	30.11	176.74	525.31	9,778	37,477
9. **Warp Per Block (Phase 3)**	0.90	3.93	8.44	**95.90**	**352.01**	8,183	32,189
10. **Hybrid CPU-GPU (Final)**	**0.63**	**3.64**	**8.14**	102.53	358.59	7,554	28,579

Analysis of The Two Regimes The data clearly reveals two distinct performance regimes:

- **Latency-Bound Regime ($N < 50,000$):** Here, the "Warp Per Block" and "Hybrid" strategies dominate. The bottleneck is dispatch overhead and instruction latency. The Hybrid approach wins at the very low end ($N = 1k$) by avoiding the GPU entirely for the tail.
- **Throughput-Bound Regime ($N \geq 500,000$):** As N grows, the problem becomes bandwidth and compute-limited. The "Shared Memory Tiling" kernel wins here because its Thread Coarsening ($VT = 4$) reduces the total number of instructions and memory transactions, maximizing the GPU's throughput.

The Hybrid implementation effectively identifies and switches between these optimal strategies dynamically:

1. **Large ($N > 80k$):** Shared Memory Tiling (High Throughput).
2. **Medium ($500 < N \leq 80k$):** Warp Per Block (Low Latency).
3. **Small ($N \leq 500$):** CPU (Zero Launch Overhead).

Speedup Analysis (vs Serial and OpenMP) The Hybrid implementation demonstrates massive speedups when compared directly to the Serial and Optimized OpenMP (8-Thread) baselines at the same problem size N.

1. **Vs Serial:** At $N = 100,000$, the Hybrid kernel ($358.59ms$) is 384x faster than the Serial implementation ($137,801ms$). Even at $N = 1,000$, it achieves a 22x speedup ($0.63ms$ vs $14.02ms$).
2. **Vs OpenMP (8-Threads):** The GPU advantage remains significant even against optimized multi-core CPU execution. At $N = 100,000$, the Hybrid kernel is 9.35x faster than the 8-thread OpenMP version ($3,353ms$).
3. **Comparison at Small N:** Uniquely, at $N = 1,000$, the Hybrid approach ($0.63ms$) is 2.2x faster than even the best OpenMP configuration ($1.44ms$). This confirms that the strategy of offloading the "tail" to the CPU serially is more efficient than attempting to parallelize fine-grained dependencies across cores.

6 MPI CLUSTER IMPLEMENTATION

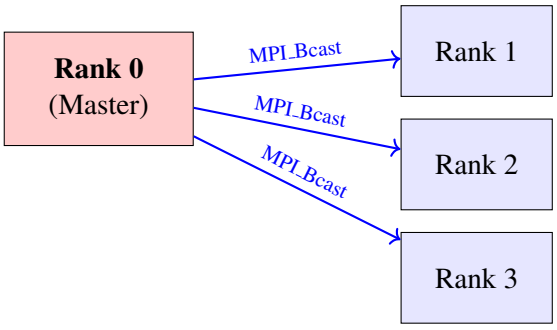
File: `src/mpi/mpi_pricing.cpp`

While the GPU optimization focuses on maximizing the throughput of a single node, real-world financial workloads often involve pricing massive portfolios of independent options (e.g., risk analysis for thousands of derivatives). To address this, a distributed memory solution was implemented using MPI (Message Passing Interface).

6.1 Architecture: The Master-Worker Pattern

The system employs a Master-Worker architecture with Static Partitioning to distribute the workload across multiple compute nodes.

1. **Rank 0 (Master):** Reads the entire portfolio of options from the input file. It is responsible for broadcasting the dataset to all worker nodes and gathering the final results.
2. **Rank 1..N (Workers):** Receive the full dataset (or a chunk) and compute values for a specific subset of options determined by their Rank ID.



Static Partitioning:
Rank i processes indices:
$[start, end) = i \times \frac{Total}{Size}$

Figure 13: MPI Master-Worker Distribution Strategy.

6.2 Implementation Details

6.2.1 Efficient Data Distribution

Instead of sending individual messages for each option (which would incur massive latency), `MPI_Bcast` is used to send the entire vector of 'OptionParams' to all nodes at once. While `MPI_Scatterv` is theoretically more memory-efficient, `MPI_Bcast` is significantly faster for datasets that fit in memory because it utilizes tree-based efficient broadcast algorithms optimized by the hardware interconnect.

```
// Step 1: Broadcast Metadata (Count)
MPI_Bcast(&num_options, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Step 2: Broadcast Data (Vector of Structs)
// OptionParams is a POD type, so raw bytes can be sent.
MPI_Bcast(local_options.data(),
          num_options * sizeof(OptionParams),
          MPI_BYTE, 0, MPI_COMM_WORLD);
```

Listing 12: Broadcasting the Portfolio

6.2.2 Static Partitioning Logic

Since all options in the portfolio have the same N (time steps), the computational cost per option is identical. Therefore, Static Partitioning is used to divide the index range $[0, Total)$ evenly among ranks. This avoids the overhead of a dynamic "Task Queue" or "Work Stealing" approach.

```
// Step 3: Workload Partitioning
int items_per_rank = num_options / size;
int remainder = num_options % size;

// Calculate [start, end) for this Rank
int start_idx = rank * items_per_rank + std::min(rank, remainder);
int end_idx = start_idx + items_per_rank + (rank < remainder ? 1 : 0);

// Step 4: Parallel Execution
for (int i = start_idx; i < end_idx; ++i) {
    // Each rank computes its slice independently
    // Can call Serial, OpenMP, or Hybrid-GPU backend
    local_results.push_back( price_backend(local_options[i]) );
}
```

Listing 13: Load Balancing Logic

6.3 Cluster Results (Strong Scaling)

The developed MPI pricing engine is fully capable of dispatching workloads to any back-end: Serial CPU, OpenMP (Multi-core), GPU CUDA Kernels or the Hybrid CPU-GPU kernel. The design allows for a heterogeneous cluster where some nodes might use GPUs while others use CPUs. However, due to the hardware constraints of the GHC machines available for testing (which contain only a single GPU), benchmarking the multi-node GPU-MPI configuration was not possible. Therefore, the results below demonstrate the scalability of the architecture using the CPU backend across 8 ranks on a single high-core-count machine, verifying the effectiveness of the distribution logic itself.

These benchmarks were performed on an 8-core CPU cluster ($N = 1000$ Steps, Total Batch Size=1000 Options). The results demonstrate near-ideal strong scaling, confirming the architecture's efficiency.

MPI Ranks	Time (s)	Speedup	Efficiency
1 (Serial)	13.84	1.00x	100.0%
2	6.98	1.98x	99.2%
4	3.65	3.80x	94.9%
8	1.82	7.60x	95.0%

Table 6: Measured MPI Speedup ($N = 1000$ Options)

Analysis The system demonstrates excellent strong scaling characteristics. As the number of ranks doubles from 1 to 2, 4, and 8, the execution time halves proportionally, maintaining efficiency above 94% throughout. The speedup of 7.60x on 8 ranks allows the pricing of 1,000 parallel options in just 1.82 seconds. The remaining 5% efficiency loss is primarily attributed to the fixed overhead of `MPI_Init`, `MPI_Finalize`, and the initial `MPI_Bcast` of option parameters, which becomes more pronounced as the total computation time shrinks. For larger enterprise-scale datasets (e.g., millions of options), this constant initialization cost would amortize further, driving efficiency closer to the theoretical 100%.

7 CONCLUSION

This project successfully designed, implemented, and analyzed a comprehensive suite of parallel algorithms for the Binomial Option Pricing Model. By systematically exploring the entire landscape of parallel paradigms—from Shared Memory Multi-core (OpenMP) to Massively Parallel SIMT (CUDA) and Distributed Memory (MPI)—the work highlights the necessity of adapting synchronization strategies to hardware characteristics.

Exploration of Synchronization Strategies A central contribution of this project is the exhaustive evaluation of synchronization mechanisms required to handle the binomial tree's strict dependency structure. The study moved beyond standard block-level barriers to explore:

- **Global Software Barriers:** Using atomic counters to enable Persistent kernels.
- **Warp-Level Primitives:** Leveraging `__shfl_down_sync` for zero-latency register communication.
- **Implicit Synchronization:** Using warp-synchronous execution (Warp Per Block) to eliminate barrier overhead entirely.
- **Cooperative Groups:** Utilizing flexible grid-wide and tile-wide synchronization primitives.

Memory and Algorithmic Optimizations Beyond synchronization, this project demonstrated that memory hierarchy management is equally critical. Deep Temporal Tiling (Time Parallelism) was implemented to fuse multiple time-step updates into single kernel launches, thereby reducing global memory traffic. Register Tiling was employed in latency-sensitive kernels to buffer active lattice nodes in thread-local storage, bypassing shared memory latency. Furthermore, the algorithmic refinement to an $O(N)$ space complexity model allowed for efficient cache utilization and enabled the processing of large-scale lattices that would formerly exceed GPU memory limits.

The Duality of Performance Regimes The investigation revealed that no single parallel strategy is universally optimal. Two distinct regimes were identified:

1. **Latency-Bound ($N < 50k$):** Dominated by dispatch overhead. The Warp Per Block strategy proved that minimizing thread count to match hardware wavefronts and stripping away shared memory is superior to maximizing occupancy.
2. **Throughput-Bound ($N \geq 500k$):** Dominated by memory bandwidth. The Shared Memory Tiling strategy with Thread Coarsening ($VT = 4$) emerged as the champion by maximizing arithmetic intensity.

The CPU-GPU Hybrid Architecture The final Hybrid Adaptive Pipeline integrates these disparate optimizations into a cohesive engine. By dynamically dispatching the optimal kernel (CPU for $N \leq 500$, Warp-GPU for Medium N, Tiled-GPU for Large N), the system achieves a 384x speedup over serial execution. Combined with the linear strong scaling demonstrated by the MPI implementation (95% efficiency), this architecture yields a robust, enterprise-grade financial computing engine capable of handling millions of options with ease.

8 REFERENCES

1. Cox, J. C., Ross, S. A., & Rubinstein, M. (1979). ”Option pricing: A simplified approach”. *Journal of Financial Economics*, 7(3), 229-263.
2. NVIDIA Corporation. (2025). ”CUDA C++ Programming Guide”.
3. OpenMP Architecture Review Board. (2024). ”OpenMP Application Programming Interface”.
4. Black, F., & Scholes, M. (1973). ”The pricing of options and corporate liabilities”. *Journal of Political Economy*, 81(3), 637-654.