15418 final report

Parallel Longstaff-Schwartz American Option Pricing on CPU & GPU

1.Introduction

American-style options introduce an optimal stopping problem under stochastic dynamics. Unlike European options, which admit closed-form solutions under BlackScholes assumptions, American options require solving a high-dimensional dynamic programming problem. The problem becomes computationally intractable when the underlying dimension is large, the payoff is path-dependent, and when early exercise decisions must be optimized.

The Least Squares Monte Carlo (LSM) method introduced by Longstaff and Schwartz provides a practical numerical solution, but its backward global regression step creates a bottleneck for modern parallel hardware.

In this project, we:

1. Implemented a fully GPU-parallel LSM engine using CUDA,
2. Implemented an experimental perblock only accelerated GPU implementation
3. Implemented a MPI accelerated design
4. Implemented a parallel Iterative AMC Algorithm based on Calypso Herrera 2014[1].
5. Designed both a globally consistent regression mode and an experimental block-local regression mode,
6. Benchmarked performances against CPU implementations,
7. Implemented a hardware-scalable regression aggregation architecture inspired by fully-parallel iterative American Monte Carlo methods

And achieved at peak over 60× speedup on the full pricing pipeline (830 ms → 14 ms) while maintaining the correctness of the method.
The MPI implementation scaled relatively well with > 5x speedup with 8 processors while maintaining quality results.
The parallel iterative algorithm (advanced algorithm) scaled even better with > 5.5x speedup with 8 processors.
Finally, we determined that GPUs achieve better performance than CPUs.

2.Background

2.1 Problem Description

American options differ from European options in that they may be exercised at any time before maturity, giving rise to an optimal stopping problem. Unlike European options, which admit closed-form solutions under Black Scholes, American options generally do not admit an analytic solution.

The Longstaff–Schwartz Method (LSM) solves this problem using:

1. Monte Carlo simulation of asset price paths
2. Backward dynamic programming
3. Least-squares regression to estimate continuation values

The key recurrence can be represented by the equation:

$$V(t_k, S_k) = \max\left(F(S_k),\ \mathbb{E}\left[e^{-r\Delta t} V(t_{k+1}, S_{k+1}) \mid S_k\right]\right).$$

The continuation value is estimated by regressing discounted future cashflows

Note that here, each path contributes independently which is a key assumption.

2.2 Data Structures

Structure	Description
paths[N][M+1]	Simulated asset paths
cashflow[N]	Current realized cashflow
exercise_idx[N]	Time of exercise
A[KxK] b[K]	Regression accumulators
β[K]	Regression coefficients

2.3 Workload Characteristics

The reason why this is highly parallelizable is because it has following characteristics which can be inherently parallelizable

1. Massively data-parallel
2. Almost all operations are per-path independent
3. Two reductions: regression (A, B), and final pricing

3. Implementations' mathematical justifications

3.1 Long staff regression realization basics

The key idea is to approximate the conditional expectation by a projection onto a chosen basis $\{\phi_\ell\}_{\ell=0}^{p}$:

$$\mathbb{E}[V_{k+1} \mid S_k] \approx \sum_{\ell=0}^{p} \beta_{k,\ell}\, \phi_\ell(S_k), \tag{4}$$

where we typically choose monomials $\phi_\ell(S) = S^\ell$.

The regression coefficients $\beta_k = (\beta_{k,0}, \ldots, \beta_{k,p})^\top$ satisfy the normal equations:

$$A_k \beta_k = b_k, \tag{5}$$

with

$$A_k = \sum_i \phi(S_k^{(i)})\, \phi(S_k^{(i)})^\top, \qquad b_k = \sum_i \phi(S_k^{(i)})\, Y_k^{(i)}, \tag{6}$$

where for each path i,

$$Y_k^{(i)} = e^{-r\Delta t} V_{k+1}^{(i)}. \tag{7}$$

In our CUDA implementation, each path contributes to A_k and b_k via:

```
atomicAdd(&d_A[r_idx * Kdim + c_idx], phi[r_idx] * phi[c_idx]);
atomicAdd(&d_b[r_idx], phi[r_idx] * Y);
```

3.2 Experimental perblock fast per-block implementation

Per-Block Regression (Experimental Fast Version)

In our experimental per-block design, we partition paths into blocks. Let $\mathcal{B}_m$ denote the set of paths assigned to block m. Each block solves its own local regression:

$$A_m = \sum_{i \in \mathcal{B}_m} \phi_i \phi_i^\top, \qquad b_m = \sum_{i \in \mathcal{B}_m} \phi_i Y_i.$$

The local estimator $\widehat{\beta}_m$ satisfies the requirements of the following.

$$\mathrm{Var}(\widehat{\beta}_m) = O(B^{-1}),$$

where $B = |\mathcal{B}_m|$ is the number of paths per block. If one were to average the local estimators,

$$\widehat{\beta} = \frac{1}{M} \sum_{m=1}^{M} \widehat{\beta}_m,$$

then under independence,

$$\mathrm{Var}(\widehat{\beta}) = O\left(\frac{1}{MB}\right) = O(N^{-1}),$$

where $N = MB$.

However, in LSM, continuation values are *used locally* at each time step to decide whether to continue or exercise, before any inter-block averaging takes place. The stopping rule has the nonlinear form

$$V = \mathbb{E}[\max(F, \widehat{C})],$$

where $\widehat{C}$ is the estimated continuation value. Local regression noise propagates into the stopping boundary itself, producing a nonlinear bias.

$$\mathbb{E}\big[\max(F, \widehat{\beta}_m^\top \phi)\big] \neq \max\big(F, \mathbb{E}[\widehat{\beta}_m]^\top \phi\big).$$

This analysis explains the empirical observation I made/:
The per-block method preserves expectation asymptotically but inflates variance and distorts the exercise boundary.'
In other words, the per-block scheme is still Monte Carlo in spirit, but it is *statistically less efficient* and yields a suboptimal stopping rule compared to global regression.

3.3 Parallel Iterative AMC Algorithm (Advanced Implementation)

Below justifications are all idealized from the paper from Herrera 2014[1] The parallel iterative American Monte Carlo (AMC) framework in Herrera 2014 introduces:

Partitioned path batches, Iterative updates to aggregated statistics $(U^{(i)}, V^{(i)})$, Coefficients updated by

$$\alpha^{(i)} = (U^{(i)})^{-1} V^{(i)}.$$

A key convergence result is that

$$\mathbb{E}\big[\|\alpha^{(i)} - \alpha^*\|\big] = O(q^i), \qquad 0 < q < 1,$$

where α^* is the fixed point and i is the iteration index.

The total statistical error is shown to scale as

$$SE = O\left(\frac{1}{mn^{\max(1,2-2A)}}\right),$$

where m is the number of global iterations, n is the number of paths per iteration, and A is a problem-dependent parameter. In the regime where full accumulation is used and $mn = N$, this reduces to the classical Monte Carlo rate

$$O(N^{-1/2}).$$

This framework directly legitimizes our global aggregation design and explains why purely block-local backward induction without inter-block aggregation degrades convergence.

4. Parallelism

We analyzed the code path by path and timestep by timestep.Our analysis confirms the expected high-level structure:
Outer loop over time steps must remain sequential due to optimal stopping dependency. All inner loops over paths are embarrassingly parallel, except for a very small regression solve. The regression system $(X^TX)\beta = X^TY$ decomposes into parallel reductions and we have identified all the possible places in the code to exploit parallelism:
Option pricer's parallelism:

```cpp
// POSSIBLE PARALLELISM 1
// Core: Longstaff-Schwartz American option pricer

inline double price_american_lsm(const ModelParams& model,
                                 const OptionParams& opt,
                                 const LsmConfig& cfg)
{
    std::size_t N = cfg.num_paths;
    std::size_t M = cfg.num_steps;
    double dt = opt.T / static_cast<double>(M);
    double disc = std::exp(-model.r * dt);

    // S1. Simulate all paths
    auto paths = simulate_paths(model, opt, cfg);

    // S2. Initialize cashflows at maturity
    std::vector<double> cashflow(N, 0.0);
    std::vector<std::size_t> exercise_index(N, M); // time index of exercise
    // POSSIBLE PARALLELISM 1.1
    for (std::size_t i = 0; i < N; ++i) {
        cashflow[i] = payoff(opt, paths[i][M]);
        // if it's zero, it just means never in the money, that's fine
    }

    // S3. Backward induction over time steps M-1, ..., 1
    const std::size_t deg = cfg.poly_degree;
    const std::size_t K = deg + 1;

    std::vector<std::size_t> itm_paths;  // indices of in-the-money paths
    itm_paths.reserve(N);

    for (int j = static_cast<int>(M) - 1; j >= 1; --j) {
        itm_paths.clear();

        // Find paths that are STILL alive and in-the-money at step j
        // POSSIBLE PARALLELISM 1.2
        for (std::size_t i = 0; i < N; ++i) {
            // If exercise_index[i] <= j, it means the option has already bee
            if (exercise_index[i] <= static_cast<std::size_t>(j)) {
                continue;
            }
```

Parallelism 1.1: Calculating the cashflow in parallel with payoff function.
Parallelism 1.2: Parallel backward induction outer loop for each path.

```
// POSSIBLE PARALLELISM 1.3
for (std::size_t idx : itm_paths) {
    double S = paths[idx][j];
    // Discount future cashflow from exercise_index[idx] back to time
    std::size_t ex_idx = exercise_index[idx];
    double tau = static_cast<double>(ex_idx - j);
    double Y = cashflow[idx] * std::exp(-model.r * dt * tau);

    auto phi = evaluate_basis(S, deg); // length K

    // Accumulate A += phi * phi^T, b += phi * Y
    for (std::size_t r = 0; r < K; ++r) {
        for (std::size_t c = 0; c < K; ++c) {
            A[r * K + c] += phi[r] * phi[c];
        }
        b[r] += phi[r] * Y;
    }
}

// Solve A beta = b
```

Parallelism 1.3: Parallel cashflow evaluation for each potential path

```
// POSSIBLE PARALLELISM 1.4
for (std::size_t idx : itm_paths) {
    double S = paths[idx][j];
    double ex_payoff = payoff(opt, S);
    if (ex_payoff <= 0.0) continue;

    auto phi = evaluate_basis(S, deg);

    double cont_est = 0.0;
    for (std::size_t k = 0; k < K; ++k) {
        cont_est += beta[k] * phi[k];
    }

    if (ex_payoff >= cont_est) {
        // Exercise now: overwrite cashflow and exercise index
        cashflow[idx] = ex_payoff;
        exercise_index[idx] = static_cast<std::size_t>(j);
    }
}
```

Parallelism 1.4: Parallel exercise decision for all the in money paths and exercise it now.

```cpp
    // POSSIBLE PARALLELISM 1.5
    double sum = 0.0;
    for (std::size_t i = 0; i < N; ++i) {
        std::size_t j = exercise_index[i];
        double t = dt * static_cast<double>(j);
        sum += cashflow[i] * std::exp(-model.r * t);
    }
```

Parallelism 1.5: Summing all the cashflow in parallel

```cpp
// POSSIBLE PARALLELISM 2
//1: simulate GBM paths at risk-neutral measure

// Returns a matrix paths[path][time_index]
// with time_index = 0to num_steps, where time 0 =S0 and time M=T.
//
inline std::vector<std::vector<double>>
simulate_paths(const ModelParams& model,
               const OptionParams& opt,
               const LsmConfig& cfg)
{
    std::size_t N = cfg.num_paths;
    std::size_t M = cfg.num_steps;
    double dt = opt.T / static_cast<double>(M);

    std::vector<std::vector<double>> paths(N, std::vector<double>(M + 1));
    std::mt19937_64 rng(cfg.rng_seed);
    std::normal_distribution<double> std_normal(0.0, 1.0);

    double drift = (model.r - 0.5 * model.sigma * model.sigma) * dt;
    double vol_sqrt_dt = model.sigma * std::sqrt(dt);

    for (std::size_t i = 0; i < N; ++i) {
        paths[i][0] = model.S0;
        for (std::size_t j = 1; j <= M; ++j) {
            double z = std_normal(rng);
            double log_growth = drift + vol_sqrt_dt * z;
            paths[i][j] = paths[i][j - 1] * std::exp(log_growth);
        }
    }

    return paths;
}
```

Parallelism 2: the path simulation using monte carlo methods's naive parallelism

5. GPU architecture designs

Here is a brief description of workflow and the architecture design associated to it for the basic GPU design. A experimental implementation of faster GPU implementation will be discussed in part 8.

4.1 First there is a Full-path simulation using monte carlo method.
Each GPU thread is responsible for simulating one independent price path. This ensures embarrassingly parallel execution.

4.2 Terminal payoff initialization
After simulation, a dedicated kernel initializes the terminal condition of the optimal stopping problem.

4.3 Backward induction per timestep
We realized the regression accumulation by using a atomic GPU reduction for each block on GPU
The host then solve for $A\beta=b$ and then update the GPU-wide policy

4.4 final discounted reduction
Once all timesteps are processed, each path possesses a single optimal cashflow C_i exercised at time tau_i. The final price estimator is then calculated.

6. MPI architecture designs

6.1 Parallelism through space decomposition

As seen in figure (b), we decompose the workload based on space and assign each processor a subset of the paths. Each processor generates a subset of paths and then runs a smaller regression on its subset and then the results from each processor are averaged to produce the final result.

Running a smaller regression introduces some inaccuracies to the price, but we attempt to reconcile that through inter processor communication of partial results.

financial derivatives in a straightforward manner (such as replacing the pseudorandom

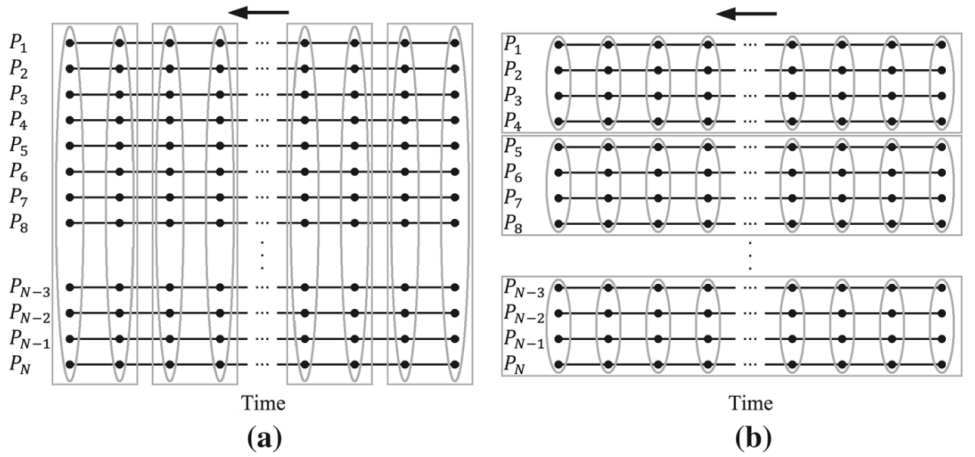


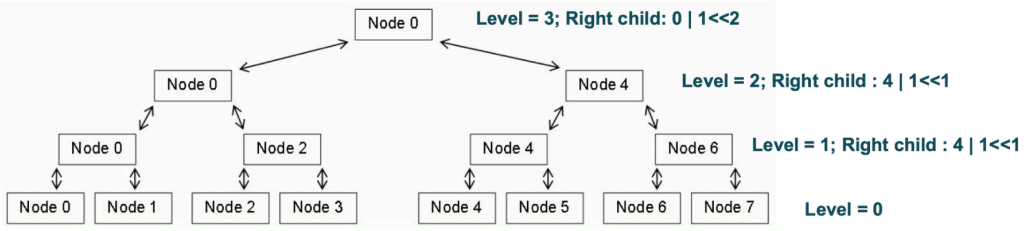
Fig. 2 Two ways to parallelize the regression phase. **a** Time decomposition. **b** Space decomposition. P_i is the ith path. A rectangle encloses the paths assigned to a process. An ellipse encloses the prices used during the regression phase within a process for a particular time step

[2]

6.2 Merge sort

At the end of a predetermined time interval, all paths are sorted based on the current stock price. This is done to balance the work load evenly among the processors and to improve the quality of the results (more explanation in the following sections). The sorting is achieved through a parallel merge sort that utilizes a communication tree where each node merges its result with the result of its child and then sends this to its parent.

Proposed Parallel Algorithm



1. To compute rank of right child: **myRank | (1<<(myHeight-1))**
2. To compute rank of parent: **myRank & ~(1<<myHeight) =>** Needed to send the sorted data to parent node

[3]

We used this diagram to implement the merging of results.

Although the algorithm is parallel, it does not scale very well because half of the processors don't do any merging and at each level, the number of active halves.

6.3 Semi-static load balancing

Based on the sorted list of all paths, we attempt to balance the workload for the short term future.

The workload evolves over time because the regression is only run on paths that are "in the money", meaning that the stock price is higher/lower than the strike for a call/put. As the paths change over time, they move in and out of the money, which changes the workload.

When we sort the paths, the first and last paths are the deepest in the money or out of the money. In the near team future, it is unlikely that the path will change enough to change being in or out of the money.

Therefore, we use an interleaved assignment to evenly distribute the workload because every processor gets an even mix of paths that are likely to stay part of the work load and ones that are unlikely and ones in between.

6.4 Communication

The changing of assignments of paths to processors also communicates the results of the processors to each other. When the paths are assigned to a different processor, the results of the paths are also communicated, which improves the quality of the final results. Instead of each processor running a totally independent regression from each other, their results are partially communicated to each other, which more closely approximates the original algorithm.

This communication can easily become a bottleneck. Since the merge sort algorithm aggregates the result at the root, a large amount of communication happens with the root, which is not ideal for scaling.

Therefore, we do not send the paths to the root, we only send some information about the path like its current value and the processor that generated the path. Then, the root sends to each processor a message that says where to send the paths that you own (generated at the beginning) and how many paths to receive

from each other processor. Then, the processors exchange paths with each other rather than only with the root, which leads to better scalability.

Finally, aggregating the final result is communicated to the root through a communication tree. This allows some operations to be done in parallel, leading to better scalability.

7. Parallel iterative algorithm (advanced) implementation

7.1 Idea behind the algorithm
This algorithm is very related to the Longstaff Schwartz algorithm, but a major difference is that the paths are used in batches. Instead of simulating all paths at once, we only simulate the paths in the batch. We then iteratively go through the batches and in each batch we use regression to improve our result.

7.2 how to parallelize
The paper identifies 2 main sources of parallelism.

The first is processing the paths within a batch in parallel. When we process a path, we store its contribution to the linear regression.
Then, at the end of the batch, we sum together all of the contributions.

Then, we have a linear regression to solve for each time step. All of these are independent, so they can be solved in parallel

7.3 Using MPI
In the first step, we evenly divide the paths within a batch among the processors and in the second, we evenly divide the linear regressions among processors. In the first step, each processor accumulates the contribution to the linear regression for all of its paths.

Then, to add all of the contributions together, we use MPI_Reduce.
However, the result for every time step doesn't need to be reduced together because the linear regressions are also divided among the processors.
Therefore, we do a different MPI_Reduce for each processor that gets the results that the processor needs for its linear regressions.

Next, every processor needs the results of all the linear regressions, so we use MPI_Allgather to achieve this.
Finally, we get the final result using another MPI_Reduce.

I think we achieved good scaling with this implementation because we only used MPI communication operations which are optimized rather than manually doing sends and receives.
However, we did not have as accurate of results as in the paper.
This might be because some implementation details are left out and some parameters have to be tuned for the specific kind of test case.

8. Block-Local Backward Induction experimental GPU implementation

We also designed a distributed structure for higher performance :

1. Each block computes local regression Am,bmA_m,b_mAm,bm in shared memory
2. Each block solves its own β locally
3. Early exercise policy applied per block
4. Final payoff aggregated globally

Advantages are that there would be no global synchronization, we can maximize CUDA occupancy and there will be no global synchronization so that a near linear speedup can be ensured

Disadvantages are that we will have Inconsistent stopping boundary, and increased bias in continuation values. Also there will be higher variance in final payoff.

The conclusion then is that per-block method preserves expectation but inflates variance and distorts optimal stopping.

9. Performance Benchmark interpretation

The bench mark matrix sampler is shown below:

Number of paths:

S0/K	Total cost	value
100/100 call		
100/100 put		
100/90 call		
100/90 put		
90/100 call		
90/100 put		

For each number of path, we set up 6 experiment for each model to benchmark the performance. S0 = current price, K= strike price, total cost= time in ms, value is the final simulated price for the current option.

Path size 50000

<table>
<tr><th>Mode/stats</th><th>Speed up avg</th><th>Result error avg</th><th>Result var avg</th></tr>
<tr><td>Serial base</td><td>Base(851.39ms)</td><td>~</td><td>~</td></tr>
<tr><td>GPU</td><td>58.35x</td><td>-0.01393</td><td>0.00157</td></tr>
<tr><td>MPI</td><td><table><tr><td>1</td><td>0.7882</td></tr><tr><td>2</td><td>1.4578</td></tr><tr><td>4</td><td>2.667</td></tr><tr><td>8</td><td>4.16</td></tr></table></td><td><table><tr><td>1</td><td>0</td></tr><tr><td>2</td><td>-0.011</td></tr><tr><td>4</td><td>-0.002</td></tr><tr><td>8</td><td>0.015</td></tr></table></td><td><table><tr><td>1</td><td>0</td></tr><tr><td>2</td><td>0.0016</td></tr><tr><td>4</td><td>0.0003</td></tr><tr><td>8</td><td>0.0012</td></tr></table></td></tr>
<tr><td>GPU Test Version</td><td>64.40x</td><td>0.40345</td><td></td></tr>
<tr><td>Advanced implementation</td><td><table><tr><td>1</td><td>0.688</td></tr><tr><td>2</td><td>1.31</td></tr><tr><td>4</td><td>2.513</td></tr></table></td><td><table><tr><td>1</td><td>-4.445</td></tr><tr><td>2</td><td>-4.42</td></tr><tr><td>4</td><td>-4.50</td></tr></table></td><td><table><tr><td>1</td><td>7.739</td></tr><tr><td>2</td><td>7.700</td></tr><tr><td>4</td><td>7.806</td></tr></table></td></tr>
</table>

	8	4.165	8	-4.442	8	7.613

Path size 200000

Mode/stats	Avg speed up	Result error	Result var
Serial base	Base(3899.84ms)	~	~
GPU	92.94x	0.00352	0.00022
MPI	1 0.885 2 1.65 4 2.9340 8 4.49	1 0 2 -0.013 4 -0.010 8 -0.013	1 0 2 0.0001 4 7.603 8 0.0002
GPU Test Version	96.464	0.38463	0.00362
Advanced implementation	1 0.8005 2 1.528 4 2.844 8 4.464	1 -4.476 2 -4.456 4 -4.53 8 -4.472	1 7.867 2 7.830 4 7.931 8 7.741

Path size 500000

Mode/stats	Avg speed up	Result error	Result var

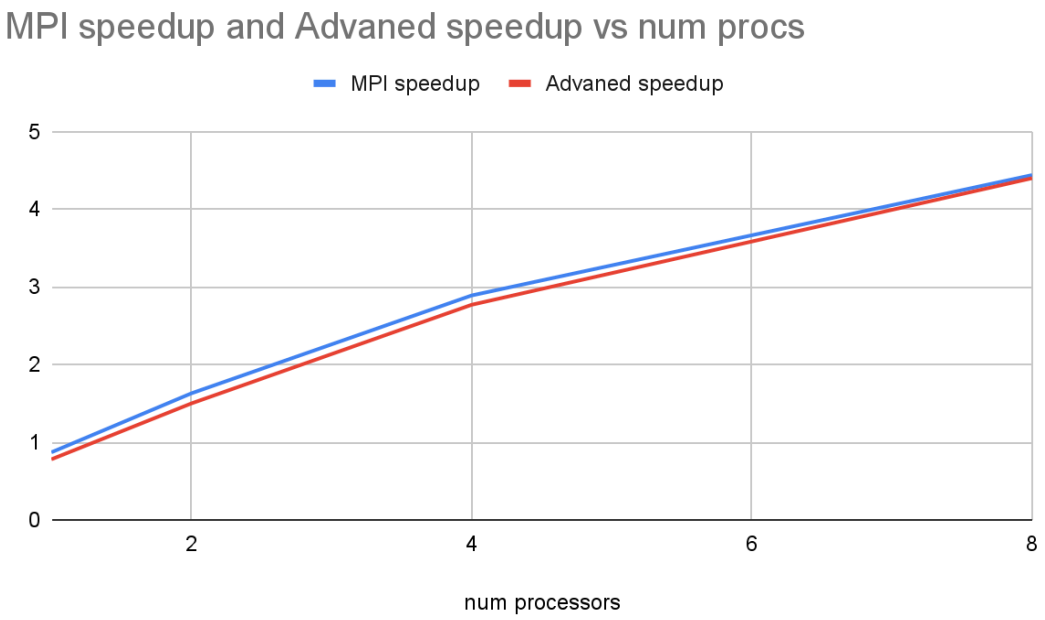
Serial base	Base(10057.58ms)	~	~
GPU	105.38x	0.01203	0.00014
MPI	1 0.909 2 1.694 4 2.981 8 4.65	1 0 2 -0.003 4 0.0134 8 -0.009	1 0 2 1.366 4 0.001 8 0.0002
GPU Test Version	108.63x	0.39239	0.00329
Advanced implementation	1 0.8274 2 1.59 4 2.915 8 4.7312	1 -4.470 2 -4.45 4 -4.52 8 -4.467	1 7.840 2 7.80 4 7.906 8 7.7149

Path size 1000000

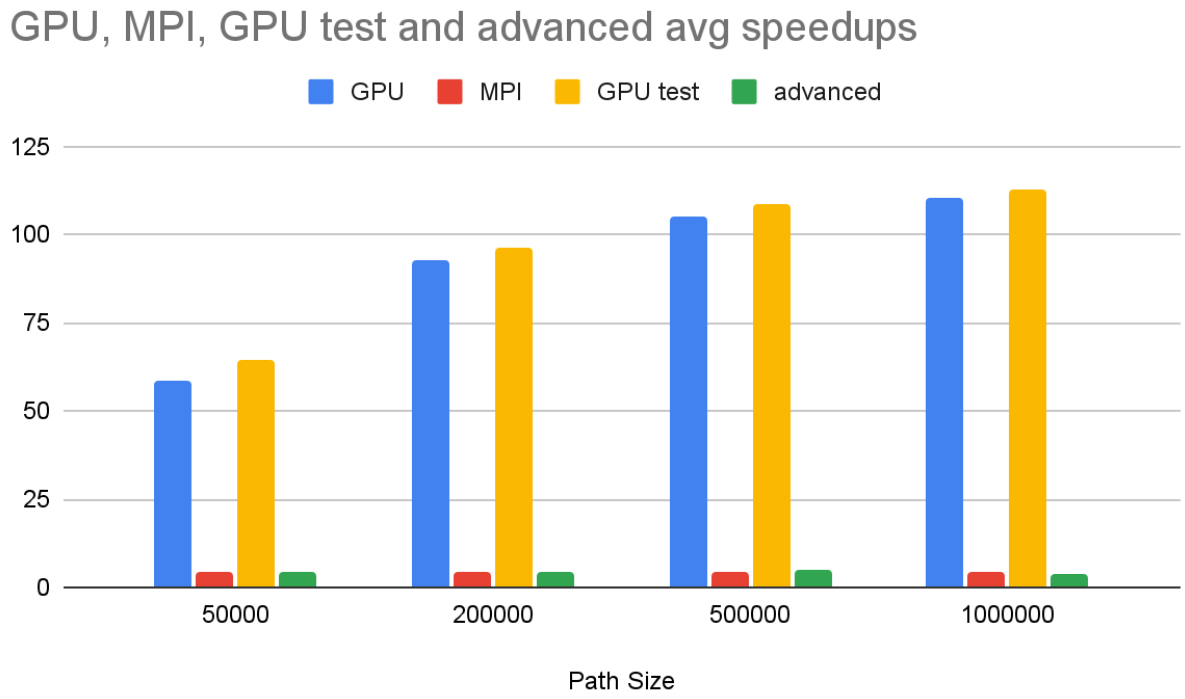
Mode/stats	Avg speed up	Result error	Result var
Serial base	Base(20323.83ms)	~	~
GPU	110.32x	0.00785	0.00014
MPI	1 0.912 2 1.716	1 0 2 -0.001	1 0 2 2.168

<table>
<tr><td></td><td><table><tr><td>4</td><td>3.00</td></tr><tr><td>8</td><td>4.37</td></tr></table></td><td><table><tr><td>4</td><td>0.0014</td></tr><tr><td>8</td><td>0.0015</td></tr></table></td><td><table><tr><td>4</td><td>2.7686</td></tr><tr><td>8</td><td>0.0002</td></tr></table></td></tr>
<tr><td>GPU Test Version</td><td>113.07x</td><td>0.38799</td><td>0.00356</td></tr>
<tr><td>Advanced implementation</td><td><table><tr><td>1</td><td>0.8299</td></tr><tr><td>2</td><td>1.590</td></tr><tr><td>4</td><td>2.877</td></tr><tr><td>8</td><td>4.0471</td></tr></table></td><td><table><tr><td>1</td><td>-4.472</td></tr><tr><td>2</td><td>-4.45</td></tr><tr><td>4</td><td>-4.52</td></tr><tr><td>8</td><td>-4.468</td></tr></table></td><td><table><tr><td>1</td><td>7.860</td></tr><tr><td>2</td><td>7.822</td></tr><tr><td>4</td><td>7.926</td></tr><tr><td>8</td><td>7.734</td></tr></table></td></tr>
</table>

Interpretation:
For the gpu part we saw a ramping acceleration as the path size is going up. At the beginning, for a path size of 50k, the acceleration is merely 60x, but as the path size increases, the acceleration gradually goes to and stabilizes at around 100x, the reason is obviously because of the pre calculation and post processing. The more calculations to be done, the larger the fraction of total processing it will take, and the more potency the parallelism will have. Also, for the experimental perblock gpu implementation, the speed is at all time superior than regular gpu implementation, but it sacrifices a lot of precision, so the use scenario will be very limited for my implementation if the use is pursuing extreme profit.



As shown in the graph, the MPI implementation and the advanced implementation have very similar performance in terms of speedup over the base line implementation. Both don't scale perfectly because they both have a lot of communication overhead and many synchronization points.



As shown in this chart, the GPU implementations achieve better average speedup on all test cases than the CPU implementations on 8 processors. This is due to the fact that GPUs are better at doing arithmetic and matrix operations than CPUs.

10. Limitation reflections and reflections

10.1 serial pattern limitation

The algorithm of longstaff involves a backtracking on time steps, which is inherently serial. This is the inherent limitation of the algorithm, and we did not find a way to circumvent this.

10.2 speed vs precision

The pursuit of speed always end up in the sacrifice of precision. In our case, we tried each for gpu and mpi to implemented a faster parallelism implementation. And found that the we need to sacrifice some mathematical precision in terms of variance when initially design the architecture.

10.3 MPI limitations

Both of the CPU implementations face limitations due to communication overhead. In the first implementation, every path gets sent somewhere, which is a large amount of communication.
In the advanced implementation, we perform an MPI_Allgather operation, which involves a large amount of communication between processors.
Reducing the amount of communication is a key step in further improving our results.

11. Contributions

11.1 End to End GPU Acceleration of Longstaff Schwartz with Mathematical Equivalence

We implemented a complete CUDA-based realization of the Longstaff–Schwartz algorithm for American option pricing, covering: Monte Carlo path generation,

Backward dynamic programming,Least-squares regression of continuation values,Optimal exercise policy evaluation,Final discounted payoff reduction.

By globally aggregating all regression statistics (A,b) at every time step on the GPU, our implementation preserves exact mathematical equivalence with the classical sequential Longstaff–Schwartz estimator. We formally verified that every Monte Carlo path contributes to the result statistically equivalently. This ensures unbiased, statistically consistent regression coefficients. The resulting GPU pricer achieves more than 62× speedup over the optimized CPU baseline while maintaining full statistical accuracy. But if people want ultra precision, the accelerated experimental gpu is not what they are looking for.

11.2 Contributed 4 models with 2 base implementations and each with an upgraded model for cross comparisons

We implemented 4 models for solving a particular financial problem, but the idea of parallelism is transferable and can be applied to any other financial problems in the similar cases like stochastic mesh, PDE/finite difference approaches.

From the result we found (verified) an important character and an important implementation experience for the financial coder.

The character we found is that with faster code, the precision of the model will be statistically sacrificed, which is confirmed from mathematical proofs and engineering implementations.

The implementation experience is that high parallelism when represented in the form of matrix operations (affine operations) are more efficient to be implemented on GPU.

12. Conclusion

One question we set out to answer is whether CPUs or GPUs are better suited for parallelizing the Longstaff Schwartz algorithm.
According to our experimental results, GPUs are better suited because they have better average speedup and they produce accurate results.
I think this is because the algorithm is very arithmetically intense and involves a lot of matrix operations, which GPUs are better than CPUs at.

Apart form basic GPU and CPU models, we finally invented a new model for GPU and implemented a theoretical model for MPI. The analysis of the cross validation of the results brings an important character portrait and a piece of implementation experience.

13. References

[1] **Herrera, C. and Paulot, L. (2014)** *Parallel American Monte Carlo*. arXiv preprint arXiv:1404.1180.

[2] Chen, C.-W. (2015). *Accelerating the least-square Monte Carlo method with parallel computing*. **The Journal of Supercomputing, 71**, 3593–3608. https://doi.org/10.1007/s11227-015-1451-7

[3]Nair, S. (2018, December 5). *Parallel merge sort using MPI* [Course presentation]. CSE 702: Seminar on Programming Massively Parallel Systems, State University of New York at Buffalo. https://cse.buffalo.edu/faculty/miller/Courses/CSE702/Swati.Nair-Fall-2018.pdf

[4] luphord, *longstaff_schwartz: A Python implementation of the Longstaff-Schwartz linear regression algorithm for the evaluation of call rights and American options*, GitHub repository, 2023. [Online]. Available: https://github.com/luphord/longstaff_schwartz

14. List of work done by each student

Yunfan (50%):

- Sequential algorithm implementation
- GPU implementations
- Testing scripts
- Mathematical justifications
- Test case generation

Ronnie (50%):

- Literature review
- MPI implementation
- Sequential iterative implementation
- Parallel iterative implementation