Parallelizing Adaptive Multilevel Monte Carlo Methods for European Option Pricing (Milestone Report)

Kyle Lee (kylel), Matt Wei (mwei2)

December 2025

1 Additional Background

As a follow-up from the feedback on our proposal to help explain the intuition behind how Monte Carlo is used for option pricing we include the following explanation:

A European option is one that can only be exercised on a specified date. To price a European option we want to estimate the value of the underlying asset on that date. In the most general case the Monte Carlo estimator is of the form:[1]

$$E[P] \approx N^{-1} \sum_{n=1}^{N} P(\omega^{(n)}) \tag{1}$$

Where $E[P]$ is the expected payoff, N is the number of samples, and $P(\omega^{(n)})$ is the payoff of the ith sample. In plainer English, we approximate the payoff by taking the average over N payoff samples. So how do we compute each payoff sample?

The method found in the literature is to model each sample utilizing a stochastic differential equation. At a very high level for each time step dt we have:
Price Change = Constant Drift + Random Volatility
There exist multiple methods to approximate and discretize this price change in the literature. We utilize the Milstein update equation. The general form is:[2]

$$\hat{S}_{n+1} = \hat{S}_n + a\,h + b\,\Delta W_n + \frac{1}{2}\frac{\partial b}{\partial S}\,b\,(\Delta W_n)^2. \tag{2}$$

Here $\hat{S}_n$ represents the approximation of the asset price at timestep n. a and b are constants. h is the timestep ΔT and $\Delta W_n = \sqrt{h} \cdot Z$ s.t. Z is sampled

[1] "An introduction to Multilevel Monte Carlo methods". Mike Giles. April 18, 2023

[2] "Improved multilevel Monte Carlo convergence using the Milstein scheme", Mike Giles. December, 2006.

independently at random from Normal(0,1).

We now apply this general equation to asset pricing. We take as input: r, which is the risk free interest rate, generally approximated as the yield on short term government securities like Treasury bills. K which is the strike price, which in the call option case guarantees that you can buy the stock at price K even if the price went up. T which is the time to maturity, which is how far in the future the option exercise date is. And finally σ which is the approximated underlying volatility of the asset. Given these parameters we can apply the Milstein discretization as follows:

$$S_{n+1} = S_n + rS_n h + \sigma S_n \, \Delta W + \frac{1}{2}\sigma^2 S_n \big[(\Delta W)^2 - h\big]. \tag{3}$$

Then the final estimated payoff of the sample is:

$$\max(S_N - K, 0) * e^{-rT} \tag{4}$$

The e term represents the discounted value given the amount we could've made from investing in risk free securities.

In a non-adaptive single level Monte Carlo approach we would generate a fixed number of samples at some fixed timestep granularity dt and take the average over the payoffs to approximate the expected payoff of the option.

In the real world instead of generating a fixed number of samples we take samples until we achieve first-order weak convergence. The mean squared error of the SDE path simulation is:

$$N^{-1}V[\hat{P}] + (E[\hat{P}] - E[P])^2 \tag{5}$$

Where N is the number of samples, P is the true theoretical payoff, and $\hat{P}$ is the estimator. Although we don't know the true value $E[P]$, we know that the weak error of the Milstein discretization is $(E[\hat{P}] - E[P]) = O(h)$ where h is the timestep size. Therefore the MSE decreases as N increases. Given this information the literature concludes that the time complexity of converging to an MSE of $O(\epsilon)$ is in $O(\epsilon^{-3})$.[3]

The multi-level adaptive approach is more efficient and converges to an MSE of $O(\epsilon)$ in $O(\epsilon^{-2})$. The mathematical proof for why it converges in $O(\epsilon^{-2})$ is described in both of the linked Giles papers. We'll instead focus on outlining the core ideas of the multilevel approach:
We have L levels, where level 0 takes $2^{0 \cdot 2} = 1$ timesteps per sample, level 1 takes $2^{1 \cdot 2} = 4$ timesteps, level 2 takes 16 timesteps and so on. The key idea is that we take many low resolution (cheaper) samples at the coarse levels (0, 1,

[3]"Improved multilevel Monte Carlo convergence using the Milstein scheme", Mike Giles. December, 2006.

etc) and less high resolution (expensive) samples at the fine grained level, reducing the number of high resolution samples in comparison to a fixed timestep granularity, and thus reducing total computation cost. We then approximate the final payoff as:

$$E[\hat{P}] = E[\hat{P}_0] + \sum_{l=1}^{L} E[\hat{P}_l - \hat{P}_{l-1}] \tag{6}$$

In other words, we approximate the total payoff as the average of the samples at level zero plus the sum of the average differences between samples at each following level. The multilevel scheme introduces interesting challenges to parallel algorithms because different levels will exhibit different amounts of work per sample.

Parameters.

$L_{\min}$: minimum level of refinement ($\geq$2)
$L_{\max}$: maximum level of refinement ($\geq L_{\min}$)
N_0 : initial number of samples (>0)
ε : desired rms error (>0)
T : time to maturity
r : treasury rate
σ : volatility
K : strike price

Algorithm 1 Sequential Multi-level Monte Carlo Option Pricing

procedure MLMC($L_{\min}, L_{\max}, N_0, \varepsilon, T, r, \sigma, K$)
    $L \leftarrow L_{\min}$
    Initialize arrays for sums, variance V_l, cost C_l, additional samples dN_l
    for $l = 0$ to $L_{\max}$ **do**
        $C_l \leftarrow 2^{2l}$ ▷ Cost per sample
        $dN_l \leftarrow N_0$ if $l \leq L_{\min}$
    end for
    while not converged **do**
        for $l = 0$ to L **do**
            if $dN_l > 0$ **then**
                Take dN_l additional MLMC samples at level l:
    Compute fine and coarse paths X_f, X_c
    Compute payoffs P_f, P_c and difference dP
                Update running sums for mean and variance
            end if
        end for
        for $l = 0$ to L **do**
            Compute mean m_l and variance V_l from running sums
        end for
        Compute optimal number of additional samples dN_l per level
        if all $dN_l \approx 0$ **then**
            Check weak convergence:
            if bias too large **and** $L < L_{\max}$ **then**
                Add a new level $L \leftarrow L + 1$
            else
                converged $\leftarrow$ true
            end if
        end if
    end while
    Compute final MLMC estimator $P = \sum_{l=0}^{L} m_l$
    return P
end procedure

2 Work Progress

- Conducted literature review
  - Learned from the 2018 ICM lecture "An introduction to multilevel Monte Carlo methods" by Michael Giles to understand multilevel Monte Carlo methods conceptually and read his 2006 paper "Improved multilevel Monte Carlo convergence using the Milstein scheme"
  - Read the 2023 paper "A massively parallel implementation of multilevel Monte Carlo for finite element models". Learned about their MPI CPU implementation utilizing master-slave architecture to schedule work.
  - Read "Efficient Asian option pricing with CUDA" and understood their parallel reduction approach + utilization of shared block memory.
  - Read "Pricing American Options with Least Squares Monte Carlo on GPUs". Learned about using CURAND for random number generation on CUDA.
  - Read "On the implementation of multilevel Monte Carlo simulation of the stochastic volatility and interest rate model using multi-GPU clusters"
- Wrote sequential Multilevel Monte Carlo code for European call option pricing. Built off Mike Giles' existing C++ code as planned in our proposal.
- Wrote a test suite consisting of three underlying assets each at three different ϵ values.
- Benchmarked the sequential implementation on our test suite. Results described and analyzed in the preliminary results section.
- Parallelized and optimized sampling at level 0 on CUDA.
- GitHub Repo Here

3 Preliminary Results

We initially benchmarked the sequential implementation and tracked the runtime across levels. Each cell of the following figure is the runtime at that level in seconds.

	AAPL, 0.005	AAPL,0.0025	APPL,0.001	TSLA, 0.005	TSLA,0.0025	TSLA,0.001	SPY, 0.005	SPY,0.0025	SPY,0.001
L=0	6.07	24.151376	162.124274	8.153709	34.716742	224.814779	11.570659	47.555901	308.061719
L=1	0.14	0.561671	3.844354	0.135128	0.579054	3.678627	0.206689	0.88991	5.590322
L=2	0.06	0.236476	1.586136	0.05597	0.241587	1.533528	0.089414	0.380811	2.414526
L=3	0.028	0.115112	0.727714	0.030836	0.124083	0.767831	0.044002	0.184902	1.163952
L=4	0.014	0.05814	0.362506	0.015356	0.061179	0.382714	0.02188	0.089505	0.553386
L=5	0.0076	0.031087	0.190489	0.007785	0.031035	0.196755	0.011179	0.050238	0.282081
6	0.003919	0.015889	0.098827	0.00419	0.016201	0.101399	0.005847	0.022921	0.154693
7	0.00213	0.00843	0.052119	0.002046	0.009163	0.052709	0.003129	0.012359	0.075518
8		0.00443	0.026821		0.004458	0.026713	0.001602	0.00604	0.038683
9		0.002193	0.014816		0.002168	0.013952	0.000801	0.003346	0.019803
10			0.007688			0.006826	0.000478	0.001945	0.010187
11			0.003482					0.000929	0.005316
12									0.003791
13									
14									
15									
16									
17									
18									
19									
20									
Other	0.004333	0.00027	0.000388	0.00018	0.000242	0.0003050000001	0.000293	0.000373	0.0004290000001
Total	6.329982	25.185074	169.039614	8.4052	35.785912	231.576138	11.955973	49.19918	318.374406

We found that level 0 consists of the vast majority of the runtime. To investigate this we checked the number of total timesteps taken at each level:

	AAPL, 0.005	AAPL,0.0025	APPL,0.001	TSLA, 0.005	TSLA,0.0025	TSLA,0.001	SPY, 0.005	SPY,0.0025	SPY,0.001
0	181,161,280	754,399,616	5,054,174,720	254,965,552	1,081,363,712	6,961,693,184	355,181,856	1,469,022,720	9,574,947,840
1	9,364,072	38,679,012	259,472,736	9,305,356	39,285,656	253,221,440	14,191,092	58,888,812	383,534,976
2	9,566,544	39,992,400	267,352,848	9,434,080	40,205,968	258,894,400	15,099,104	62,481,056	406,953,568
3	10,829,056	43,436,480	277,890,688	11,756,352	47,023,936	293,094,144	16,476,032	65,882,944	424,438,656
4	11,724,288	47,481,088	295,438,592	12,510,720	49,857,536	310,774,016	17,890,816	71,265,024	446,925,056
5	12,768,256	51,858,432	320,876,544	13,188,096	52,517,888	332,180,480	18,933,760	75,949,056	475,458,560
6	13,295,616	54,927,360	341,512,192	14,090,240	56,057,856	349,704,192	20,213,760	79,335,424	499,056,640
7	14,565,376	58,916,864	361,136,128	14,319,616	59,146,240	364,593,152	21,905,408	83,509,248	523,190,272
8	15,204,352	62,259,200	376,569,856	0	57,540,608	375,521,280	21,757,952	84,869,120	543,686,656
9	0	61,865,984	403,963,904	0	59,768,832	393,216,000	22,544,384	94,109,696	557,580,288
10	0	0	434,110,464	0	0	379,584,512	26,214,400	108,003,328	572,522,496
11	0	0	390,070,272	0	0	0	0	104,857,600	599,785,472
12									855,638,016
13									
14									
15									
16									
17									
18									
19									
20									
diff	83,843,720	294,982,796	1,325,780,496	170,361,092	619,959,192	3,650,909,568	159,955,148	579,871,412	3,286,177,184

When we present this data in the final report we'll consider other data visualization approaches such as a stacked bar chart since this raw data is a bit overwhelming.

Each cell is the total number of timesteps (num samples * num timesteps per sample) per level until convergence is reached. The final row "diff" is how many more timesteps are taken at level 0 vs the rest of the levels combined. From this we observed that the reason level 0 takes the majority of time is because it consists of the majority of timesteps.

This led us to approach by starting with parallelizing level 0 because 1) The majority of computation time occurs at this level and 2) this is the only case where there's only one timestep per sample (all other levels require multiple timesteps).

Our initial approach was as follows:

Algorithm 2 MLMC Level 0 with CUDA

procedure MLMC_L0_GPU(N, T, r, σ, K)
    // Step 1: Initialize GPU vectors for sums
    $l0\_sums \leftarrow$ GPU array of size NUM_THREADS
    $l0\_squared\_sums \leftarrow$ GPU array of size NUM_THREADS
    // Step 2: Compute number of samples per thread
    $samples\_per\_thread \leftarrow \lceil N/NUM\_THREADS \rceil$
    // Step 3: Launch GPU kernel
    KERNEL_MLMC_L0($l0\_sums, l0\_squared\_sums, samples\_per\_thread, T, r, \sigma, K$)
    // Step 4: Synchronize GPU
    cudaDeviceSynchronize()
    // Step 5: Reduce GPU vectors to get total sum and squared sum
    $total\_sum \leftarrow$ sum of all elements in $l0\_sums$
    $total\_squared\_sum \leftarrow$ sum of all elements in $l0\_squared\_sums$
    // Step 6: Update MLMC accumulators for level 0
end procedure

Algorithm 3 GPU Kernel: MLMC Level 0

procedure KERNEL_MLMC_L0($sums, squared\_sums, samples\_per\_thread, T, r, \sigma, K$)
    $tid \leftarrow$ unique thread ID
    Initialize local variables: $local\_sum \leftarrow 0$, $local\_squared\_sum \leftarrow 0$
    Initialize random state for this thread
    for $i = 1$ **to** $samples\_per\_thread$ **do**
        Draw $dW \sim N(0, T)$
        Compute level 0 payoff:
        $X \leftarrow K + r * K * T + \sigma * K * dW + 0.5 * \sigma^2 * K * (dW^2 - T)$
        $local\_sum \mathrel{+}= X$
        $local\_squared\_sum \mathrel{+}= X^2$
    end for
    Write results to global memory:
    $sums[tid] \leftarrow local\_sum$
    $squared\_sums[tid] \leftarrow local\_squared\_sum$
end procedure

We utilized Thrust to initialize the GPU vectors and reduce to get total sum. We initially took samples per thread as the ceiling of $\frac{N}{NUM\_THREADS}$ so we take at least the number of requested samples to simplify the math for an initial implementation. We then decomposed our runtime into kernel and reduction time to identify the bottleneck. Preliminary results:

Test Case	Epsilon	Kernel (s)	Reduce (s)	Total (s)	Speedup
AAPL 1-year	0.005	2.460864	0.067368	3.066150	1.96x
AAPL 1-year	0.0025	7.371398	0.203898	7.905124	3.16x
AAPL 1-year	0.001	12.619622	0.329058	23.042752	7.27x
Tesla 6-months	0.005	2.451412	0.063812	2.951542	2.83x
Tesla 6-months	0.0025	8.600609	0.230272	7.997257	4.42x
Tesla 6-months	0.001	12.930996	0.315235	22.311710	10.17x
SPY 2-years	0.005	3.694445	0.092407	4.383112	2.66x
SPY 2-years	0.0025	9.891194	0.250357	13.830509	3.48x
SPY 2-years	0.001	21.789643	0.534254	34.627897	9.06x

The reduction takes little time compared to the total amount of time the kernel launches + computation takes, so we focus on optimizing the kernel. Over the course of the past week we implemented the following optimizations (we have more detailed analysis which we'll write up in our final report):

- Early exiting the kernel when there are more threads than samples needed
- Experimenting with the maximum number of threads
- Precomputing some of the arithmetic
- Using CURAND Normal2 instead of Normal and taking samples in groups of 2
- Initializing CURAND state in a separate kernel

In our final report we'll explain each of these optimizations in detail and provide quantitative analysis via a table comparing the speedup with each optimization vs. without.

The speedups we were able to achieve when parallelizing only level 0 and leaving the rest of the levels sequential when applying our optimizations is:

Test Case	Epsilon	Runtime (s)	Speedup
AAPL 1-year	0.005	0.850798	10.16x
AAPL 1-year	0.0025	1.199243	20.84x
AAPL 1-year	0.001	7.764797	21.85x
Tesla 6-months	0.005	0.224377	37.27x
Tesla 6-months	0.0025	0.895148	35.519x
Tesla 6-months	0.001	5.674873	40.19x
SPY 2-years	0.005	0.647541	18.04x
SPY 2-years	0.0025	2.593242	18.63x
SPY 2-years	0.001	16.363068	19.2x

To check for correctness we verify the final outputted payoff for each test case is within 2ϵ of the sequential output as this is the guarantee we'd expect from our convergence check.

4 Goals

Based on these preliminary results we believe we'll be able to produce all the deliverables we intended in the initial proposal. Recall our initial goals based on comparing to the 2018 Lay paper were:
$\geq 12\times$ speedup on a single node as the 100 percent goal and $\geq 25\times$ speedup on < 10 GPUs as the 150 percent goal.

We've already achieved our initial performance goal of $\geq 12\times$ speedup for all but the AAPL 1-year case on GHC. We believe our high performance relative to the multi-node Lay paper we referenced in the initial proposal is due to our use of Normal2 which uses a more efficient Normal generation algorithm on CUDA and the Normal generation is the most costly part of the algorithm.

Therefore we're raising the bar for our performance goals as follows:
1. (100 percent goal) $\geq 50\times$ speedup on all test cases on GHC (single node).
2. (150 percent goal) $\geq 100\times$ speedup on ≤ 8 GPUs on PSC.

In terms of "nice to haves" we want to try for the 125 percent goal of utilizing multiple GPUs with MPI because combining MPI and CUDA isn't something we've done in the class yet and seems very interesting.

We've completed the 75 percent mark of our initial proposal. We're defining the "coarse" sample case as the level 0 case since its the only case with a single timestep per sample.

At the poster session we plan to present:
1. Speedup graph on a single node (GHC)
2. Speedup graph using multiple nodes (PSC)
3. Live demo. User enters T, K, σ and we compare the runtimes of sequential vs. parallel live. We'll also have a list of "stocks" they can pick from if they don't want to choose T, K, σ themselves. This demo will be via a command line interface and will print diagnostics of the number of samples being taken at each level until convergence as the program runs.

5 Remaining Schedule

- (Dec 1-3) Parallelize fine grained levels (levels 1+) (Kyle and Matt)
- (Dec 3) Experiment with more optimizations (Kyle and Matt)
  - Try Normal4 (Matt)
  - Try changing method to compute all Normals up front in bulk (Kyle)
- (Dec 4-6) Work on writeup and analysis (Kyle and Matt)
  - Write clear background section

  - Do analysis on what the performance bottlenecks are and what regions of code are dominating the runtime
  - Improve data visualization of benchmarks
  - Document the optimization journey and show the performance gains from each optimization

- (Dec 5-7, stretch goal) Make a parallel MPI implementation and benchmark it. Optimize it if possible and document the performance gains. (Kyle)
- (Dec 6-7) Write a CLI tool for the live demo + include a list of prepopulated stock to K, T, σ values for the demo (Kyle and Matt)
- (Dec 6-7) Make poster and finalize writeup (Kyle and Matt)

6 Remaining Concerns

The only outstanding concern regards PSC GPU node access. Sometimes it takes up to several hours to get the resource allocation after calling the interact command for 2 v100-16 GPUs. We'll work hard to get progress on MPI finished early so we have enough time to wait out the resource allocation time.