Parallelizing Adaptive Multilevel Monte Carlo Method for European Option Pricing in CUDA

Kyle Lee (kylel), Matt Wei (mwei2)

December 2025

1 Summary

We parallelized an Adaptive Multilevel Monte Carlo approach for forecasting the payoff of European stock options in CUDA. We extended our implementation to utilize multiple GPUs by combining CUDA with MPI. On a single NVIDIA RTX 2080 single GPU we achieve an up to $> 178\times$. When utilizing 4 NVIDIA Tesla V100 GPUs we achieve an up to $> 5430\times$ speedup.

2 Background

Options are a financial instrument with value derived from an underlying asset, such as a stock. In the case of European options, the owner gains the right to buy (call) or sell (put) at the strike price (K) on a chosen date (T). For a call option for example, this means that if after T time the underlying asset increases to price $K + x$ then the owner profits x.

Traders utilize options to profit from the volatility of stocks. To effectively utilize options traders use models to predict the payoff of an option.

2.1 Monte Carlo Methods

The payoff of an option is directly proportional to the value of the underlying asset after time T. Monte Carlo methods are appropriate because they utilize repeated random sampling to model the volatility of assets. A simple Monte Carlo estimator for option payoff would be 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

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

taking the average over N payoff samples.

One of the common methods in the literature to compute $P(\omega^{(n)})$ is to model each sample as a stochastic differential equation. At a very high level for each time step dt we have:

$$\text{Price Change} = \text{Constant Drift} + \text{Random Volatility} \tag{2}$$

Because the true equation is continuous we must approximate the value with a discretization. We utilize the Milstein update equation. The general form is:[2]

$$\hat{S}_{n+1} = \hat{S}_n + a\,h + b\,\Delta W_n + \frac{1}{2}\frac{\partial b}{\partial S}\,b\,(\Delta W_n)^2. \tag{3}$$

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 independently at random from Normal(0,1).

Now applying this equation to asset pricing where r is the risk free interest rate, generally approximated as the yield on government securities like Treasury bills, and σ is the approximate underlying volatility of the asset:

$$S_{n+1} = S_n + rS_nh + \sigma S_n\,\Delta W + \frac{1}{2}\sigma^2 S_n\big[(\Delta W)^2 - h\big]. \tag{4}$$

Then the final estimated payoff of the sample is:

$$\max(S_N - K, 0) * e^{-rT} \tag{5}$$

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

In our algorithm we take K (strike price), T (time to maturity), r (risk free interest rate), and σ (underlying volatility) as user parameters representing the option.

2.2 Adaptive Multilevel Monte Carlo

In an adaptive approach instead of the user specifying how many samples to take, we continue sampling until some notion of convergence is achieved. We utilize the idea outlined in Giles' 2023 lecture on Monte Carlo methods which involves sampling until we achieve first-order weak convergence. The MSE of the SDE path simulation is:

$$N^{-1}V[\hat{P}] + (E[\hat{P}] - E[P])^2 \tag{6}$$

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

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 approach, which is what we'll be parallelizing, is more efficient and converges to an MSE of $O(\epsilon)$ in $O(\epsilon^{-2})$. The proof for why is described in the linked Giles literature. We instead focus on explaining the procedure. 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 samples at the coarse levels (0, 1, etc) and few samples at the fine grained levels so that we take fewer total timesteps across all samples in comparison to taking all samples at the same timestep granularity. We then approximate the final payoff as:

$$E[\hat{P}] = E[\hat{P}_0] + \sum_{l=1}^{L} E[\hat{P}_l - \hat{P}_{l-1}] \tag{7}$$

In other words, we approximate the total payoff as the average of the samples at level zero plus the sum of the differences between samples at each following level. We provide psuedocode for the sequential algorithm below:

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

Algorithm 1 Sequential Multi-level Monte Carlo Option Pricing

procedure MLMC($\varepsilon, T, r, \sigma, K$)
  $L \leftarrow L_{\min}$
  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 at level l
    $dN_l \leftarrow N_0$ if $l \leq L_{\min}$
  end for
  while not converged **do**
    for $l = 0$ to L **do**
      if $dN_l > 0$ **then**
        for $n = 0$ to dN_l **do** ▷ dN_l level l samples
          Compute fine ($\hat{P}_l$) and coarse ($\hat{P}_{l-1}$) payoff diff via (4), (5)
          Update running sums for mean and variance at level l,
          but for $l = 0$ simply store the sum as suggested by (7)
        end for
      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 $dN_l \approx 0$ for all l **then**
      Check weak convergence:
      if bias too large **and** $L < L_{\max}$ **then**
        Add a new level $L \leftarrow L + 1$
        Update additional samples dN_l for all levels
      else
        converged $\leftarrow$ true
      end if
    end if
  end while
  Compute final MLMC estimator $P = \sum_{l=0}^{L} m_l$ via (7)
  return P
end procedure

2.3 Workload Analysis

Effectively parallelizing Adaptive Multilevel Monte Carlo on GPUs is still an open research problem because work balancing is challenging. At the start of an iteration, we know how many samples at each level to take and the number of timesteps each sample needs. We then need to effectively map this workload to our available compute resources.

The computationally intensive part of the application is computing the sam-

ple payoffs, with coarse samples taken on the order of billions and fine samples having timesteps taken up to 2^{40} per sample.

This sheer amount of computation provides multiple avenues for parallelization: paralellizing between levels, between samples, and between timesteps. Levels are generally (see below) independent and can be computed in parallel. Samples are also independent and can be computed in parallel. Lastly we can rewrite the math for computing the payoff of one sample to make the generation of Normal variables and most of the arithmetic independent for taking timesteps within a sample.

For Multilevel Monte Carlo to converge, samples between fine and coarse levels need to be coupled. This introduces dependency and increases the challenge.

In terms of locality we want timesteps within a sample to have both spatial and temporal locality with one another and samples within a level to have locality so that we can reduce them into an average payoff efficiently.

Much of the computational time comes from generating Normal random variables so we'll explore how to use random generators efficiently in parallel.

3 Setup and Initial Benchmarking

3.1 Test Suite

We developed a test suite representing real world options to benchmark our implementations on options with diverse underlying assets. Our test suite is as follows:

	T	σ	K
AAPL	1.0	0.25	280.0
TSLA	0.5	0.30	430.0
SPY	2.0	0.10	680.0

We picked this set because each stock represents a different option strategy. AAPL is a somewhat volatile option, TSLA a shorter higher volatile option, and SPY a stable low volatility option. For each option we test at $\varepsilon \in \{0.005, 0.0025, 0.001\}$ so we benchmark on 9 cases in total.

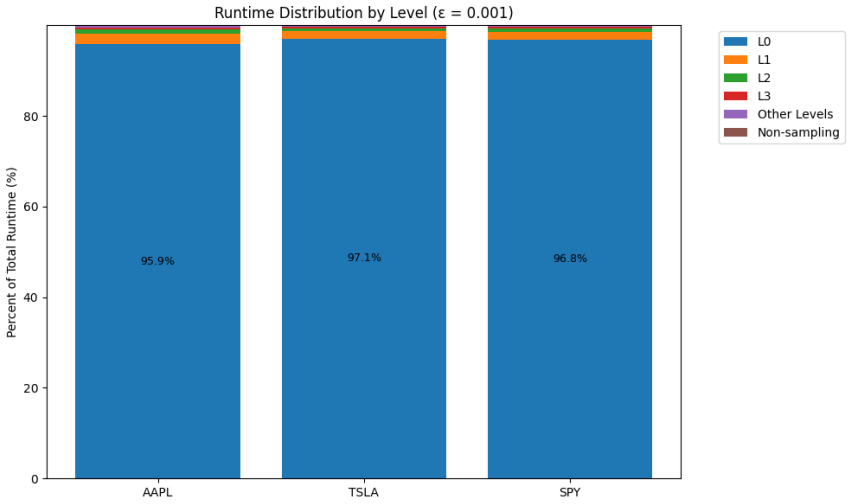
3.2 Sequential Workload Analysis

We first benchmarked our sequential implementation on GHC.

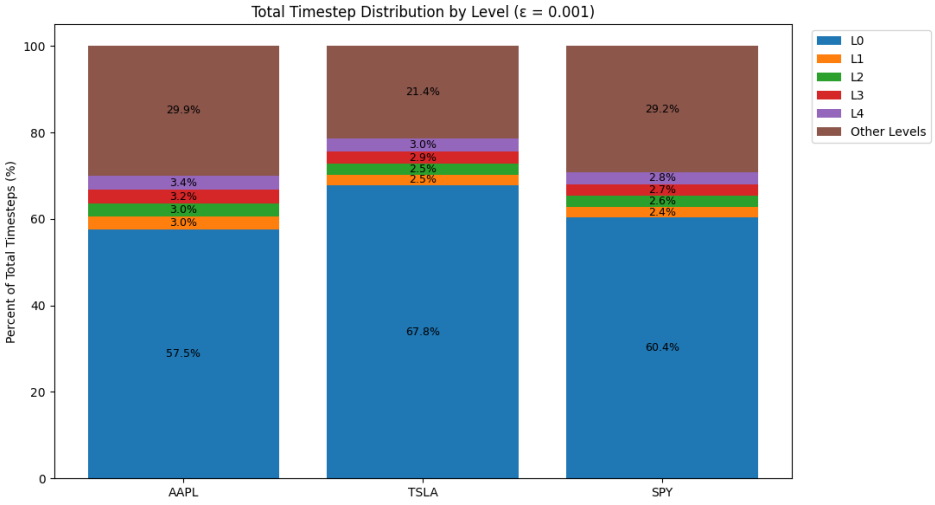
Table: Sequential Runtimes

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	6.027 (s)	24.994 (s)	167.43 (s)
TSLA	8.35 (s)	35.374 (s)	227.92 (s)
SPY	11.677 (s)	48.257 (s)	314.34 (s)

As T and σ increase the runtime will generally increase as well. As ε decreases runtime increases. We then further decomposed the runtimes into per level sample computation and other work to determine where the most expensive operations are.



We found that the vast majority of runtime ($> 95\%$) comes from sampling at level 0. The non-sampling work is negligible, which tells us that we should be able to achieve significant speedup by parallelizing the sampling process. The $\varepsilon \neq 0.001$ cases show similar patterns. We then investigated why this is the case by seeing how many total timesteps are taken at each level:



We see that level 0 accounts for $> 55\%$ of the total timesteps computed. The runtime being disproportionately large ($\approx 95\%$ vs. $\approx 55\%$) is likely because in our sequential implementation each level 0 timestep taken is a single sample which corresponds to a function call. The number of function calls is probably the overhead that makes level 0 take more percent of time than its percentage of samples. Even so, we should expect the percent of runtime level 0 takes to dominate the total runtime. Level 0 is a special case because it's the only case where samples are only one timestep and we don't need to do any sample coupling. Therefore we started our approach by parallelizing level 0 before attempting to parallelize other levels.

3.3 Correctness Validation

We can use the sequential outputs to verify the correctness of our parallel implementations. We know that by the previous analysis about convergence that a correct parallel program will output a final payoff within 2ε of the sequential output.

4 CUDA Approach

We began by parallelizing level 0 as a special case, optimizing, then extending these optimizations to determine how to parallelize the rest of the levels. All the following results in this section were benchmarked and tested on GHC57 to avoid using up too much PSC credit.

4.1 Initial Approach

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 $dW \sim N(0, T)$
        $dW \leftarrow dW * \sqrt{T}$
        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

Our initial approach was to focus on sample level parallelism by dividing up the samples between threads, using a kernel to compute each sample, and at the end reducing the total together via Thrust. We initialize the GPU vectors as Thrust vectors.

Each thread is given a unique slot in the sums GPU array and accumulates its samples together via summing the payoffs in its slot. We do this to minimize global writes since this enables a single thread to take n samples and do only 2 global writes (one for sums and one for squared sums which we need to later compute the variance).

We then decomposed our runtime into kernel launch+computation and reduction time to identify the bottleneck:

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.

4.1.1 Starter Code

In order to implement our solution we first implemented a sequential implementation by working off Mike Giles' C++ general Multi Level Monte Carlo and Milstein Discretization code.[4]. We had to streamline the code together to be specific to European call options and used the starter code to gain intuition on the math behind the algorithm.

Giles also has a CUDA implementation which we read for inspiration, but we did not copy any of it and took a very different approach. His approach simulates dynamic work balancing with CUDA whereas we rely on optimizing sample level parallelism with static assignment.

4.2 Optimization and Refinement

4.2.1 Exact Sampling

In our initial approach we simplified the math by rounding samples per thread up to $\lceil N/NUM\_THREADS \rceil$. Our initial justification for this approach is that if all threads run in parallel then the bottleneck should be the thread computing the most samples, so if all threads compute the same number of samples the

[4]"Multilevel Monte Carlo software", Mike Giles

runtime should be similar.

However some flaws in this hypothesis are:
1. Normal distribution sampling may not be fully independent so increasing the total number of samples may slow down the whole program
2. More threads taking the max number of samples increases the possibility of a longer running straggler
3. If $N << NUM\_THREADS$ then we may take significantly more samples than needed
We tested our hypothesis by comparing each thread taking the ceiling with taking the exact number of samples we need and early returning threads that would compute unnecessary samples.

Table: Speedup With vs. Without Exact Sampling

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.799×	1.066×	1.1701×
TSLA	0.717×	1.211×	1.342×
SPY	0.904×	1.194×	1.021×

We see a moderate speedup in all but the $\varepsilon = 0.005$ cases. We believe the slowdown when $\varepsilon = 0.005$ is because when $\varepsilon = 0.005$ we compute very few samples per thread so early exiting vs. computing the few samples may be similar runtime so we don't experience savings in this case. Since in the majority of cases we experience a speedup, we proceed with this optimization.

4.2.2 Dynamic Array Size

Initially we always made the GPU array size $MAX\_THREADS$ even if we early exited some threads. Here we make the GPU array size equal to the number of threads that'll be used to hopefully cut down on initialization overhead and the cost of the Thrust reduction.

Table: Speedup With vs. Without Dynamic Array Sizing

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	1.284×	1.002×	1.137×
TSLA	1.273×	1.006×	0.937×
SPY	1.268×	1.042×	0.872×

We experience speedup in all but the TSLA and SPY $\varepsilon = 0.001$ cases. We see that as ε decreases the speedup also generally decreases. This makes sense because when ε is small we're more likely to have levels where we don't need to take as many samples as $MAX\_THREADS$ and therefore making the array the minimum size we need has cost savings. However when ε is larger we're likely

to use all MAX_THREAD lanes and doing the resizing adds some overhead that leads to a slowdown. Since this optimization benefits our speedup in the majority of cases we apply it.

4.2.3 Pre-computation

We observed that in the MLMC kernel some computation can be precomputed. For each sample we compute:

$$X = K + r \cdot K \cdot T + \sigma \cdot K \cdot dW + 0.5 \cdot \sigma^2 \cdot K \cdot (dW^2 - T) \tag{8}$$

Observe that $(K + r \cdot K \cdot T), (\sigma \cdot K), (0.5 \cdot \sigma^2 \cdot K)$ are the same across all threads since K, r, T, σ are constants. Additionally each thread computes $\sqrt{T}$ for each sample but this is also a constant. We tried passing these values as constant to the kernel call. The tradeoffs would be whether the memory access + moving the constants from CPU to GPU would be more expensive than the recomputation at each iteration.

Table: Speedup With vs. Without Pre-computation

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	1.013×	1.109×	0.981×
TSLA	1.083×	1.043×	1.068×
SPY	0.914×	0.943×	1.373×

We once again observe that in the majority of cases we see a speedup. We see the most significant speedup from SPY when $\varepsilon = 0.001$. This is significant because we know this to be the most computationally intensive test case so this shows us that this optimization improves our scalability. We proceed with this optimization.

4.2.4 Pre-initialization

In order for cuRAND to be thread-safe each thead must have its own independently generated cuRAND state. In our initial implementation we generated such state on each kernel launch. However this leads to duplication when we do multiple iterations. We could instead generate the states once, save them to device memory, and reference the memory instead of recomputing each iteration.

Table: Speedup With vs. Without Pre-initialization of cuRAND

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.747×	1.244×	1.088×
TSLA	0.727×	1.268×	0.964×
SPY	1.001×	1.499×	0.996×

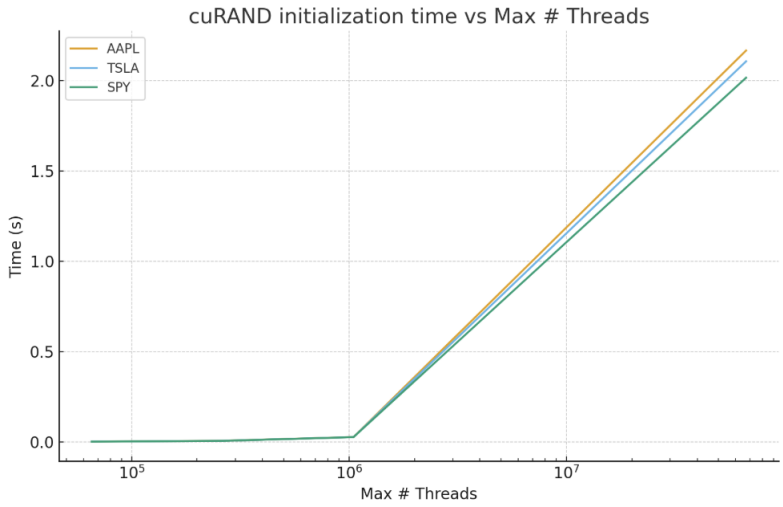
The results are quite mixed. In some cases we see a very significant speedup, like in all the $\varepsilon = 0.0025$ cases and in some cases a very significant slowdown

(like in TSLA $\varepsilon = 0.005$). The results therefore are not conclusive on whether reinitializing each kernel launching or writing and reading from global device memory is more optimal. We continue with this optimization since the benefits may be more pronounced when we can reuse the same state across kernel launches for all levels since this is tested when we only kernel launch for level 0.

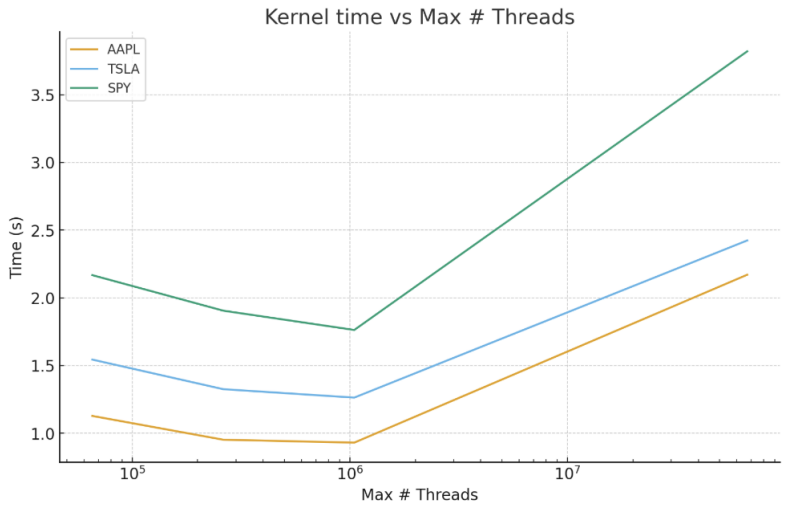
4.2.5 Max Threads

Initially we set max threads to be $2^{13 \cdot 2}$ because we planned for level 13 which has $2^{13 \cdot 2}$ timesteps per sample to be the last level where we can compute all the timesteps for a sample in one batch, we later changed this strategy. However, we found that this introduced significant overhead in all aspects: cuRAND initialization, kernel launches, and reduction.

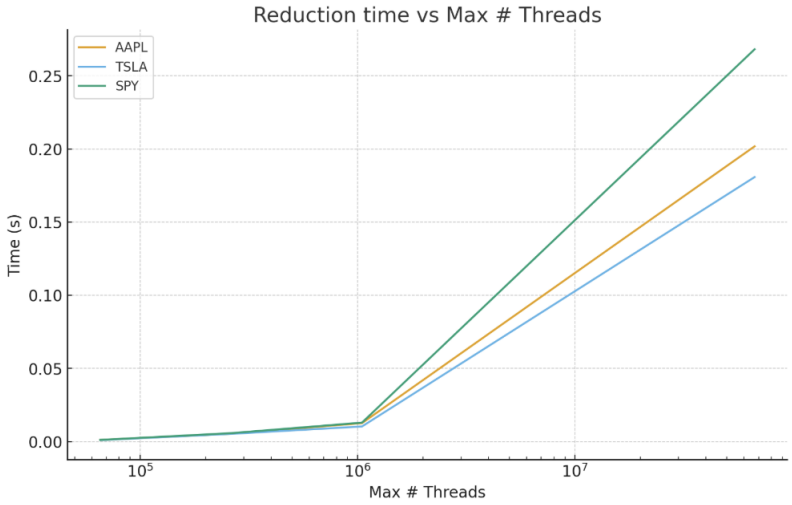
We added timers around the cuRAND initialization block and found that when $MAX\_THREADS = 2^{13 \cdot 2}$ that cuRAND state initialization alone took > 2 seconds for each test case. This overhead dominated the runtime when ε was small. We then varied the max number of threads to find the optimal value. To reduce visual clutter we've graphed only the values around the optimal, but we tested more values. For all the following graphs we've used the $\varepsilon = 0.0025$ data but the data for when $\varepsilon = 0.005, 0.001$ displays the same patterns.



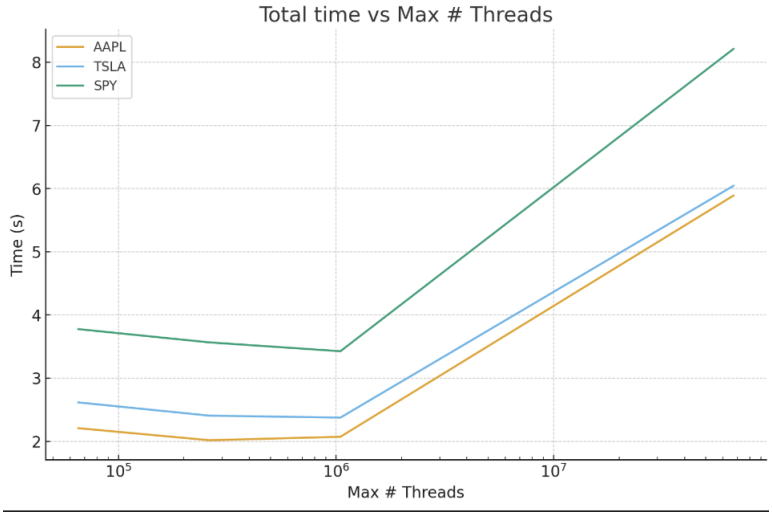
We observe that as the number of threads increases the cuRAND initialization time increases linearly.



The time the kernels take to run is a bowl shape with the time initially decreasing as we add more compute via threads and then increasing linearly as the overhead from launching more threads increases.



We observe that as the number of threads increases the reduction time increases linearly as the size of the GPU array increases linearly with the number of threads which is the size of the array we're reducing over.



We found the optimal max number of threads to be $2^{9.2}$ when we analyzed the best runtimes over $\varepsilon \in \{0.005, 0.0025, 0.001\}$.

4.2.6 Normal Generation Technique

The cuRAND library provides three approaches to generating Normally distributed random variables. One which generates 1 per call, another that generates 2 per call and uses the Box-Muller approach, and another that generates 4 per call utilizing a Philox generator. We test each of these approaches by adjusting out code to generate $X \in \{2, 4\}$ normals using the appropriate generator and then having an internal loop using each of the generated normals to create samples.

Table: Speedup from Normal2 (compared to Normal1)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.697×	1.662×	1.624×
TSLA	2.577×	2.651×	2.675×
SPY	1.336×	1.366×	1.338×

Table: Speedup from Normal4 (compared to Normal1)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.813×	1.621×	1.556×
TSLA	2.353×	2.651×	2.570×
SPY	1.242×	1.301×	1.260×

We find both approaches to provide a speedup over generating only one normal at a time except in the AAPL $\varepsilon = 0.005$ case. This is probably because that case requires few samples so the overhead from using this more complicated normal generator doesn't outweigh the performance gain from generating multiple normals in each call. Similarly in all but the AAPL $\varepsilon = 0.005$ case Normal2 outperforms Normal4. As such we proceed with using Normal2.

4.2.7 Alternative Approach: Batched Generation

We also tried a completely different logic flow to generate the normals up front rather than having each kernel pull from the normal distribution. We thought that generating the normals together in bulk might provide better opportunity for parallelization at that step and therefore a better speedup.

Algorithm 4 Batched Generation Strategy

1: $num\_processed \leftarrow 0$
2: **while** $num\_processed < N$ **do**
3: $batch\_size \leftarrow \min(N - num\_processed,\ NUM\_THREADS)$
4: Allocate vector $l0\_sums[batch\_size] \leftarrow 0$
5: Allocate vector $l0\_squared\_sums[batch\_size] \leftarrow 0$
6: Generate $batch\_size$ normal random variables in parallel:
$normals[i] \leftarrow$ NormalGenerator($seed$) for all i
7: Compute transformed values:
$l0\_sums[i] \leftarrow custom\_transform\_fn(normals[i])$
8: Square each transformed value:
$l0\_squared\_sums[i] \leftarrow (l0\_sums[i])^2$
9: Reduce to obtain totals:
$total\_sum \leftarrow \sum_i l0\_sums[i]$
$total\_squared\_sum \leftarrow \sum_i l0\_squared\_sums[i]$
10: Accumulate:
$suml[1][l] \leftarrow suml[1][l] + total\_sum$
$suml[2][l] \leftarrow suml[2][l] + total\_squared\_sum$
11: $num\_processed \leftarrow num\_processed + batch\_size$
12: **end while**

We use Thrust for all steps: We use a Thrust transform to generate the normals, Thrust with a custom transformation function to convert the normals to samples, Thrust to transform that vector to squared sums, and Thrust to reduce into a final sum. We found this method to perform substantially worse than our original.

Table: Speedup Original vs. Batched Generation

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	1.557×	4.346×	4.748×
TSLA	9.732×	8.696×	8.593×
SPY	3.505×	3.675×	4.011×

We find that the original strategy of doing the Normal generation and computation in our custom kernel to be significantly faster than this batched method. This is because the batched method is bottlenecked by $NUM\_THREADS$ and can only generate one sample per GPU array spot because each spot can only hold one normal random variable. This leads to more global memory accesses and multiplies the number of Thrust calls necessary which significantly increases overhead. Therefore we do not use this alternative approach.

4.3 Parallelizing All Levels

4.3.1 Sample Level Parallelization

Our initial approach to parallelizing the rest of the levels was to do sample level parallelization. The differences between level 0 and the rest of the levels is that in the rest of the levels:

1) At each level we need to compute coupled timesteps, i.e. both a "fine-grained" and "coarse" timestep given the same two normal random variable pulls

2) For each sample we need to chain together multiple timesteps to reach the final estimated payoff

Algorithm 5 Sample Level Parallel Kernel

Require: sums_ptr, squared_sums_ptr, samples_per_thread, total_samples
Require: level l T, r, σ, K, curand state array
if this tid doesn't need to take samples **then**
    return
end if
$nf \leftarrow 2^l, \quad nc \leftarrow nf/2, \quad hf \leftarrow T/nf, \quad hc \leftarrow T/nc$
$num\_samples \leftarrow \min(samples\_per\_thread, \quad total\_samples - (tid - 1) \cdot samples\_per\_thread)$
for $i = 1$ to $num\_samples$ **do**
    $X_f \leftarrow K, \quad X_c \leftarrow K$
    for $n = 1$ to nc **do**
        draw two normals (Z_1, Z_2) from curand_normal2
        $dW_f[0] \leftarrow \sqrt{hf}\, Z_1$
        $dW_f[1] \leftarrow \sqrt{hf}\, Z_2$
        for $m = 0$ to 1 **do**
            $X_f \leftarrow X_f + rX_f hf + \sigma X_f dW_f[m] + \frac{1}{2}\sigma^2 X_f (dW_f[m]^2 - hf)$
        end for
        $dW_c \leftarrow dW_f[0] + dW_f[1]$
        $X_c \leftarrow X_c + rX_c hc + \sigma X_c dW_c + \frac{1}{2}\sigma^2 X_c (dW_c^2 - hc)$
    end for
    $P_f \leftarrow \max(0, X_f - K)$
    $P_c \leftarrow \max(0, X_c - K)$
    $dP \leftarrow e^{-rT}(P_f - P_c)$
    $local\_sum \leftarrow local\_sum + dP$
    $local\_squared\_sum \leftarrow local\_squared\_sum + dP^2$
end for
$sums\_ptr[tid] \leftarrow sums\_ptr[tid] + local\_sum$
$squared\_sums\_ptr[tid] \leftarrow squared\_sums\_ptr[tid] + local\_squared\_sum$

The pipeline we utilize is similar to our sample level strategy with level 0. We first initialize Thrust GPU arrays, populate them with our kernel, and then reduce with summation. Each array spot in the Thrust GPU array corresponds to n samples such that $n \cdot NUM\_THREADS = N$ where N is the total number of samples we need to take.

Observe we apply the optimizations such as utilizing normal2, early exit, etc that we discovered in the previous section to this initial level1+ kernel. We did not apply the precomputation optimization. We tested that optimization here and the performance was worse. We believe this to be because the memory accesses this would take to be more costly than the arithmetic in this case.

4.3.2 Initial Results

Table: Raw Runtime with Sample Level Parallelism

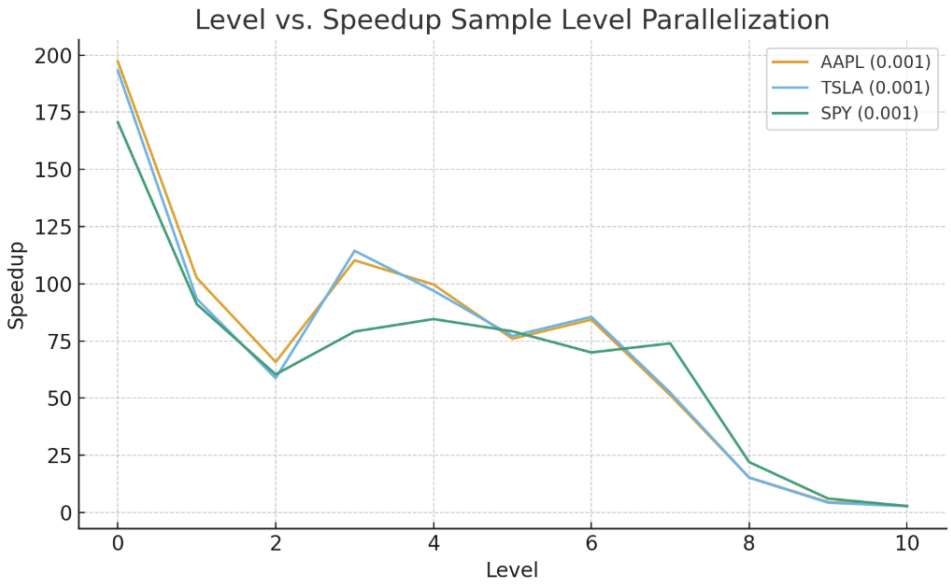
	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	$0.497s$	$0.185s$	$0.923s$
TSLA	$0.081s$	$0.237s$	$1.269s$
SPY	$0.105s$	$0.338s$	$1.981s$

Table: Speedup with Sample Level Parallelism

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	$12.115\times$	$134.667\times$	$181.326\times$
TSLA	$103.529\times$	$148.909\times$	$179.604\times$
SPY	$110.271\times$	$142.373\times$	$158.651\times$

We find that we experience significant speedup in all cases but AAPL when $\varepsilon = 0.005$. We explain why this is in the later final results section.

To better understand what limits our speedup and identify alternative approaches we checked the speedup at each level.



We include only the $\varepsilon = 0.001$ data to avoid cluttering the visual, but the $\varepsilon = 0.005, 0.0025$ data shows the same pattern. We see that as level increases

the speedup generally decreases.

This is likely because when the level gets large each sample needs to take more timesteps, for example 2^{20} timesteps per sample at level 10. This leads us to an alternative approach:

4.3.3 Timestep Level Parallelization

We observed that we can factor out the previous approximation of the asset price in the Milstein update equation, leaving behind a constant factor dependent only on a normally distributed random variable:

$$S_{n+1} = S_n(1 + rh + \sigma\Delta W + \frac{1}{2}\sigma^2\big[(\Delta W)^2 - h\big]). \tag{9}$$

Thus the sample can be computed via a multiplicative reduction over all its Milstein factors. Note that factor is not truly a constant (still depends on per timestep randomness), since if it was truly constant we could use a sequential algorithm to get logarithmic exponentiation per sample anyways.

Continuing our pattern of 1D work assignment, we implemented another kernel that assigns a block a number of samples, and has the threads in the block cooperatively perform the reduction in batches of timesteps for each sample, accumulating the results in a register before writing back to global memory.

Algorithm 6 Timestep Level Parallel Kernel

Require: sums_ptr, squared_sums_ptr, samples_per_block, total_samples
Require: level l, T, r, σ, K, curand state array

Let $b \leftarrow$ block index, $t \leftarrow$ thread index within block
$curr \leftarrow b \cdot samples\_per\_block$
if $curr \geq total\_samples$ **then**
    return ▷ this block has no work
end if
$rem \leftarrow total\_samples - curr$
$num\_samples \leftarrow \min(samples\_per\_block,\ rem)$
$nf \leftarrow 2^l, \quad nc \leftarrow nf/2, \quad hf \leftarrow T/nf, \quad hc \leftarrow T/nc$
Initialize $local\_sum \leftarrow 0$, $local\_squared\_sum \leftarrow 0$
Allocate shared array $fine\_factors[0..\text{THREAD\_DIM} - 1]$
Allocate shared array $coarse\_factors[0..\text{THREAD\_DIM} - 1]$
for $i = 1$ to $num\_samples$ **do** ▷ block cooperates on each sample
    $X_f \leftarrow K, \quad X_c \leftarrow K$
    for $base = 0$ to $nc - 1$ step blockDim.x **do**
        $sid \leftarrow base + t$
        if $sid < nc$ **then**
            Draw two normals (Z_1, Z_2) from `curand_normal2` using thread (b, t)'s state
            $dW_f[0] \leftarrow \sqrt{hf}\, Z_1$
            $dW_f[1] \leftarrow \sqrt{hf}\, Z_2$
            $dW_c \leftarrow dW_f[0] + dW_f[1]$
            $a \leftarrow \text{MilsteinFactor}(r, \sigma, hf, dW_f[0])$
            $b \leftarrow \text{MilsteinFactor}(r, \sigma, hf, dW_f[1])$
            $fine\_factors[t] \leftarrow a \cdot b$
            $coarse\_factors[t] \leftarrow \text{MilsteinFactor}(r, \sigma, hc, dW_c)$
        else
            $fine\_factors[t] \leftarrow 1$
            $coarse\_factors[t] \leftarrow 1$
        end if
        barrier across threads in block
        Perform tree reduction on $fine\_factors$ and $coarse\_factors$ in-place
        barrier across threads in block
        if $t = 0$ **then**
            $X_f \leftarrow X_f \cdot fine\_factors[0]$
            $X_c \leftarrow X_c \cdot coarse\_factors[0]$
        end if
        barrier across threads in block
    end for
    if $t = 0$ **then**
        $P_f \leftarrow \max(0, X_f - K)$
        $P_c \leftarrow \max(0, X_c - K)$
        $dP \leftarrow e^{-rT}(P_f - P_c)$
        $local\_sum \leftarrow local\_sum + dP$
        $local\_squared\_sum \leftarrow local\_squared\_sum + dP^2$
    end if
end for
if $t = 0$ **then**
    $sums\_ptr[b] \leftarrow sums\_ptr[b] + local\_sum$
    $squared\_sums\_ptr[b] \leftarrow squared\_sums\_ptr[b] + local\_squared\_sum$
end if

Both the timestep level parallel and sample parallel kernels have the same amount of work, but distributed along different axes (timesteps vs samples). Hence, we condition on level to decide which kernel to launch such that the machine is most saturated, expecting workloads on higher levels (more timesteps, less samples) to fare better with the timestep-level optimization. After fine tuning the level at which we switch to timestep-level parallelism, we find that we reap the most speedup around 7 and 8:

Table: Speedup with Timestep Level Parallelism on Levels 7 and above

	$\varepsilon = 0.005$	$\varepsilon = 0.0025$	$\varepsilon = 0.001$
AAPL	40.36×	134.56×	171.01×
TSLA	103.97×	149.05×	177.60×
SPY	109.63×	155.19×	178.48×

Table: Speedup with Timestep Level Parallelism on Levels 8 and above

	$\varepsilon = 0.005$	$\varepsilon = 0.0025$	$\varepsilon = 0.001$
AAPL	41.97×	129.87×	173.59×
TSLA	99.78×	150.78×	180.15×
SPY	114.11×	153.23×	185.27×

As expected, we encounter a desired peak in performance as we sweep the level condition from low to high. The speedup on the Levels 7 and above case performs very well, as the most significant loss is a only about a 5% decrease at AAPL with $\varepsilon = 0.001$, while in the other cases we either see a way more significant gain in speedup or roughly the same speedup.

Table: Percentage Speedup Differences

	$\varepsilon = 0.005$	$\varepsilon = 0.0025$	$\varepsilon = 0.001$
AAPL	+233.1%	−0.08%	−5.69%
TSLA	+0.43%	+0.09%	−1.12%
SPY	−0.58%	+9.00%	+12.50%

The behavior aligns with our hypothesis, since the extra timestep parallelism promotes more saturated work assignment in specific cases where timestep depth is large relative to the number of sample so the work assignment more effectively fills the GPU (e.g. AAPL at coarse tolerance, SPY at tightest tolerance). The reason why AAPL at $\varepsilon = 0.005$ sees such huge gains is because the computation corresponding to higher levels being optimized by timestep parallelism holds a larger portion of the total work.

5 Multi-GPU with MPI

Algorithm 7 Initial MPI Parallelization

Initialize MPI
$pid \leftarrow$ MPI_Comm_rank()
$nproc \leftarrow$ MPI_Comm_size()
Select GPU device based on pid
Allocate `curand` state on GPU and initialize it
$L \leftarrow L_{\min}$, converged $\leftarrow$ false
while not converged **do**
    Initialize local accumulators local_diff[ℓ], local_sqdiff[ℓ]
    MPI_Bcast (N_ℓ, $\forall \ell$)
    for $\ell = 0$ to L **do**
        Compute per-process sample allocation:
            $dN_\ell^{(pid)} \leftarrow \lfloor dN_\ell / nproc \rfloor$
            ROOT receives the remainder
    end for
    for $\ell = 0$ to L **do**
        if $dN_\ell^{(pid)} > 0$ **then**
            Perform local Monte Carlo kernel evaluation
            Update local sums
        end if
    end for
    MPI_Reduce local_diff $\rightarrow$ root
    MPI_Reduce local_sqdiff $\rightarrow$ root
    if $pid =$ ROOT **then**
        Update global sums and total sample counts
        Compute level means m_ℓ, variances V_ℓ
        Compute optimal new sample counts dN_ℓ
        Check convergence criterion
        if weak error too large and $L < L_{\max}$ **then**
            Add new MLMC level
        end if
    end if
    MPI_Bcast convergence flag
    MPI_Bcast current number of levels L
end while
if $pid =$ ROOT **then**
    Compute final MLMC estimator $P = \sum_\ell \frac{\text{diff}_\ell}{N_\ell}$
    Print diagnostics and timing information
end if
Free GPU memory
return P

At a high level our initial strategy is to split the number of samples to compute for each level approximately equally between each node. At the beginning of each while loop iteration we broadcast the total number of samples per level from the root to all nodes and divide up the number of samples to compute roughly equally with any remainder going to the root.

After all nodes have completed their sampling the result is summed up via MPI reduction. The root is responsible for checking for convergence and broadcasting the convergence flag and the new number of levels.

We first benchmarked our non MPI implementation on PSC. We'll show and discuss these results in the Final Results section later. We then benchmarked utilizing 1, 2, and 4 GPUs.

Table: Speedup with 2 GPUs CUDA+MPI vs. 1 GPU CUDA only

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.566×	0.751×	0.918×
TSLA	0.822×	0.943×	1.078×
SPY	0.672×	0.909×	1.221×

Table: Speedup with 4 GPUs CUDA+MPI vs. 1 GPU CUDA only

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.257×	0.447×	0.574×
TSLA	0.357×	0.549×	0.674×
SPY	0.382×	0.524×	0.840×

Our initial results leave much room for improvement. We actually experience a slowdown when going from one node to multiple except in the cases of TSLA and SPY $\varepsilon = 0.001$ at 2 GPUs. Those cases being the ones to experience slight speedup make sense because they're the most compute intensive so distributing the computation likely has the most significant performance gains in those cases.

To investigate why speedup is so poor we decomposed our runtime into array initialization, curand setup, kernel runtime, reduction, and other (which includes the MPI communication time). With 2 GPUs array initialization, curand setup, reduction, and other all slowdown by ≈ 1.2 to $2\times$. However the kernel runtime speeds up by $\approx 1.8\times$. When we use 4 GPUs array initialization , curand setup, reduction and other all slowdown by ≈ 2.1 to $3\times$. However kernel runtime speeds up by $\approx 3.5\times$.

This is good news because we know that our strategy of partitioning out samples

across threads is indeed speeding up the kernels almost linearly. The slowdown to other is due to the communication cost. The slowdown to array initialization, curand setup, and reduction is surprising. Our theory is that the size of all of these steps is the same when we add nodes and then some nodes take longer at these steps and become a straggler making the rest of the nodes wait. Because the kernel was already optimized in prior steps, each of these sections (kernel, curand setup, reduction, etc) take similar amounts of time. Therefore even though we successfully speed up the kernel almost linearly we still experience a slowdown in total.

5.1 Optimization

5.1.1 Minimizing Messages

In our initial approach we do 3 broadcasts (num samples per level, converged flag, and num levels) per iteration. We also do 2 reductions (diff sums and squared diff sums) per iteration. We optimize by decreasing the total number of messages which reduces communication overhead.

We first decrease the number of broadcasts from 3 to 1 by packing the num samples per level, converged flag, and num levels into one array and send it all at once.

We then decrease the number of MPI reductions from 2 to 1 by packing diff sums and squared diff sums into one send and receive buffer. The performance gain from 1 broadcast + 1 reduction over 3 broadcasts + 2 reductions is:

Table: Speedup with Less Messages (2 GPUs)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.951×	1.021×	1.008×
TSLA	1.014×	1.013×	0.969×
SPY	1.004×	1.0123×	0.984×

Table: Speedup with Less Messages (4 GPUs)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	1.269×	1.193×	1.255×
TSLA	1.284×	1.214×	1.248×
SPY	1.248×	1.263×	1.261×

We observe that with 2 GPUs the performance gains are generally positive, although not significant. The performance gains are much more significant in the 4 GPU case. This makes sense because the overhead and network bandwidth usage of communication will scale with the number of nodes. This optimization enables our program to be more scalable, so we continue with it.

5.1.2 Ring vs. Broadcast

Initially we communicate the iteration parameters using a broadcast from the root. In order to reduce bandwidth contention we instead try a ring communicate strategy.

Table: Speedup with Ring vs. Broadcast (2 GPUs)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	1.011×	1.001×	1.007×
TSLA	0.992×	0.988×	1.003×
SPY	0.988×	0.993×	1.012×

Table: Speedup with Ring vs. Broadcast (4 GPUs)

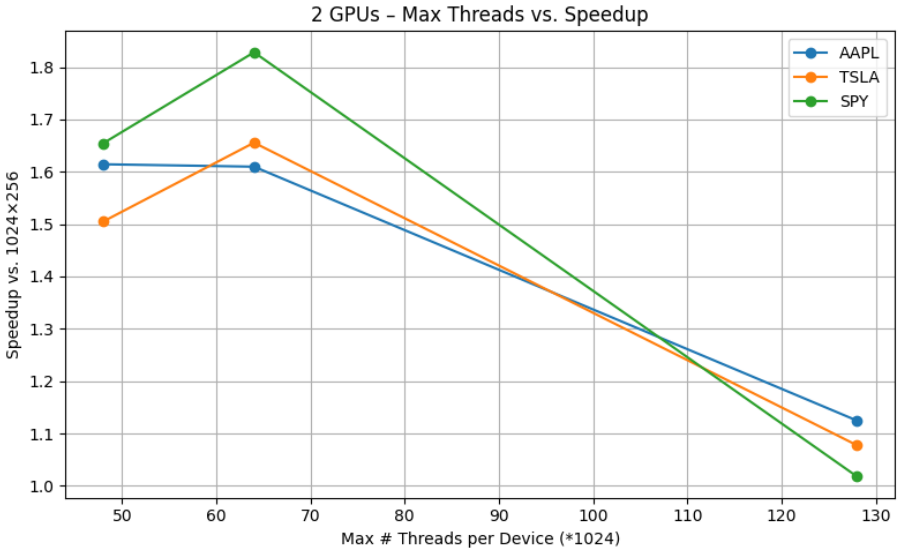
	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	1.024×	1.015×	1.003×
TSLA	1.000×	1.004×	1.002×
SPY	0.988×	0.996×	0.992×

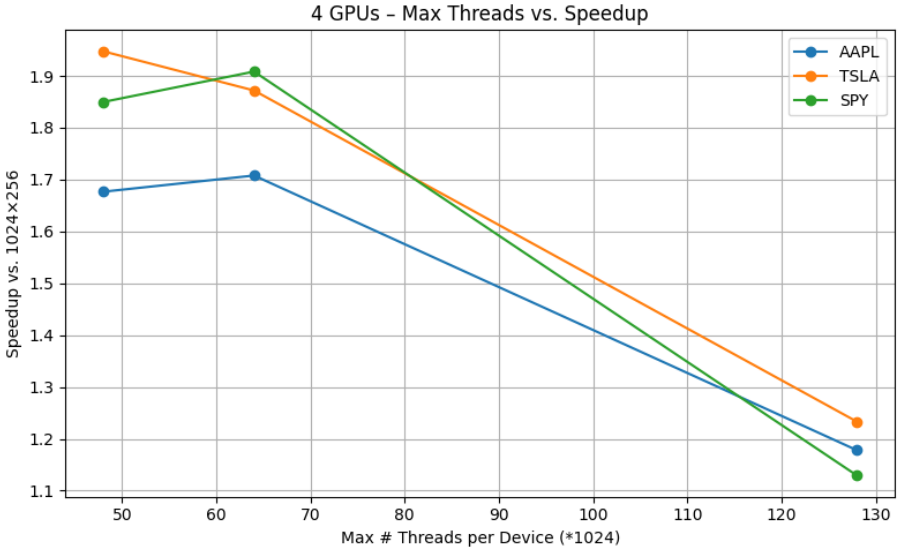
We observe that there is no statistically significant speedup, as the differences in runtime could be due to random non communication method related factors. This also makes sense since we've already reduced the communication to very little and we're not using many nodes.

We've decided to continue with the ring method because it should be more scalable if the number of nodes becomes large since a broadcast would lead to more bandwidth contention at a given point in time since we've compressed all the iteration data into one message making that message larger in our previous optimization.

5.1.3 Adjusting Max Num Threads

We made the observation earlier that the kernel runtime speeds up as desired but other areas slow down. We attempt to speed other areas up by adjusting the maximum number of threads, which will decrease the maximum reduction, init, and curand setup array sizes. We predict that will be a tradeoff between making the kernel runtime slower (less threads total) but making the other areas faster (smaller max array sizes). Our results below are speedups benchmarked against the optimal value we found on GHC ($1024 \cdot 256$).





We find the optimal to be when the maximum number of threads per device is $1024 \cdot 64$ based on where the highest speedup point is for the majority of lines in the above graph. Note in both cases that the highest point for 2/3 lines is at $1024 \cdot 64$ max threads. This performs better because we still have a high number of total threads because our utilizing multiple devices and the smaller max array sizes make the other steps, like Thrust reduction, more efficient.

5.2 Alternative Approach: Level Parallelism

We tried an alternative parallelization method: partitioning out entire levels across devices rather than partitioning out samples roughly equally across devices. We partition out the levels are equally as we can with the "last" (pid such that next pid is root) node has any remainder that couldn't be divided out equally. We compare our work partitioning/balancing strategies.

Table: Speedup with Level vs. Sample Parallelism (2 GPUs)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	1.090×	1.288×	0.673×
TSLA	1.773×	1.095×	0.565×
SPY	1.463×	1.066×	0.474×

Table: Speedup with Level vs. Sample Parallelism (4 GPUs)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	0.961×	1.958×	0.768×
TSLA	2.577×	1.624×	0.575×
SPY	2.528×	1.621×	0.572×

Observe that the results are rather disorderly with some cases experiencing a $> 2\times$ speedup and others experiencing a $> 2\times$ slowdown. Observe also that for both 2 GPUs and 4 GPUs the slowdown in all the $\varepsilon = 0.001$ cases is significant. We believe this is because when ε is larger the number of samples for a given level is smaller because we need less samples to reach convergence. In the cases with fewer samples the level parallelism approach performs well.

However, recall that the majority of samples come from level 0. This effect becomes more pronounced as ε decreases. With level parallelism one processor will get all the level 0 samples. This leads to work imbalances and these imbalances are especially pronounced when ε is small. This is what causes the significant slowdowns in the $\varepsilon = 0.001$ cases.

Therefore we do not use this approach and instead stick with sample parallelism because sample parallelism is more scalable with small ε.

6 Final Results

6.1 GHC

6.1.1 Final Speedup

Table: Final Speedup vs Sequential (GHC) (1 GPU)

	$\varepsilon = 0.005$	$\varepsilon = 0.0025$	$\varepsilon = 0.001$
AAPL	40.36×	134.56×	171.01×
TSLA	103.97×	149.05×	177.60×
SPY	109.63×	155.19×	178.48×

With a single GPU we observe speedups ranging from approximately 40× to 178×, with tighter tolerances (smaller ε) exhibiting higher speedup. This trend is consistent across all three asset types (AAPL, TSLA, SPY) and reflects the fact that decreasing ε increases the parallelizable compute portion of the MLMC simulation.

TSLA and SPY exhibit consistently higher speedup than AAPL at all tolerances, achieving over 100× speedup even at the coarsest tolerance. AAPL, in contrast, achieves only 40× speedup at $\varepsilon = 0.005$. This is due to cuRAND setup and other fixed overheads consuming a larger fraction of the runtime. As ε decreases and total compute increases, these overheads become a less significant portion of the total work, resulting in improved speedup.

It is also notable that the increase in speedup from $\varepsilon = 0.0025$ to $\varepsilon = 0.001$ is modest compared to the increase from $\varepsilon = 0.005$ to $\varepsilon = 0.0025$. This suggests that beyond a certain workload size, speedup becomes bounded by hardware throughput. In other words, once the GPU is fully utilized, additional parallel work does not yield proportional reductions in runtime.

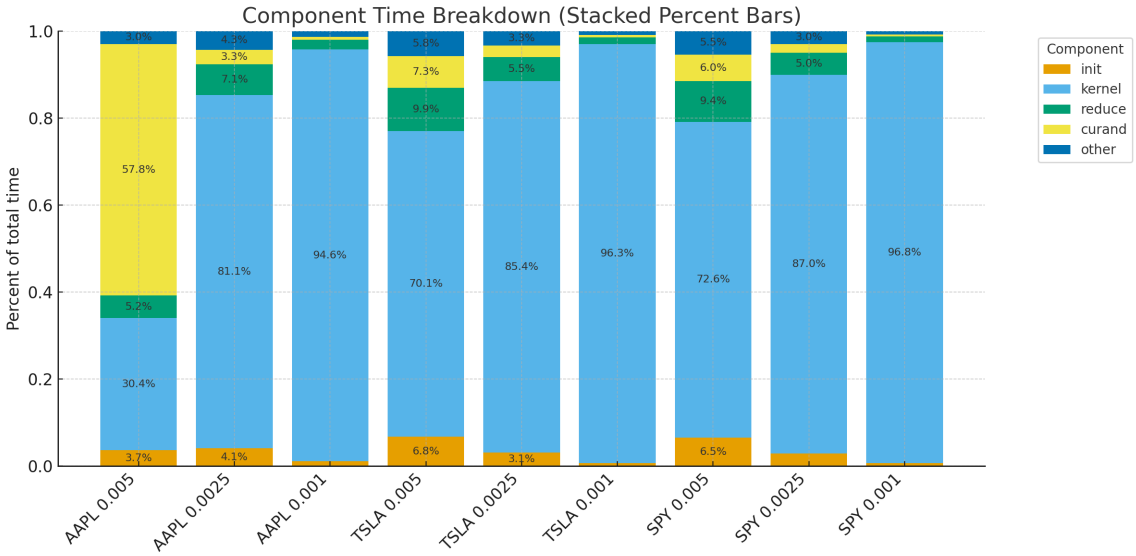
6.1.2 Speedup Limitations

- **Inherent sequential phases in MLMC.**
  - **Adaptive level and sample decisions** The number of additional levels and samples depends on variance estimates computed from the previous batch of samples; future work cannot be scheduled until prior work completes.
  - **Path evolution through time.** Each sample must be advanced through all timesteps before its contribution to the level sums is known; partial work offers no benefit because variance estimators require complete paths.
  - **Per-iteration convergence checks.** MLMC termination depends on global statistics computed from all completed levels; this requires barriers where work is paused until reductions finish and stopping criteria are evaluated.

Thus, kernel execution cannot be streamed indefinitely: synchronization points and adaptive decision-making enforce a minimal critical path that limits total achievable speedup.

- **cuRAND initialization and state setup.** Initializing RNG state is an expensive fixed-cost operation. Because this work must be performed before any samples can be generated, it remains a sequential bottleneck that cannot be parallelized away.
- **Limited benefit from further parallelism in the dominant phase.** In most cases, kernel time accounts for over 95% of runtime. Therefore, even perfect parallelization of all non-kernel work provides only marginal end-to-end gains overall.

6.1.3 Component Analysis



The figure above shows the component time breakdown for three test cases (AAPL, TSLA, SPY) and three accuracy tolerances ($\varepsilon \in \{0.005, 0.0025, 0.001\}$). Each bar reports the percentage of runtime spent in kernel execution, init, and cuRAND setup, Thrust reduction, and all remaining work (denoted other). Although the three assets differ in maturity, volatility, and payoff distribution, we observe consistent structural trends across all test cases.

Kernel time dominates and grows rapidly as ε decreases. For all three assets, the kernel component transitions from a moderate fraction of total time at $\varepsilon = 0.005$ to complete domination ($> 95\%$) at $\varepsilon = 0.001$. This behavior is a direct consequence of the MLMC work model: decreasing ε exponentially

increases the number of samples drawn at the coarsest levels. Since the kernel performs path simulation, the increased sampling cost dwarfs all fixed overheads, causing normalized percentages to converge to nearly all kernel time.

Initialization cost is non-negligible at coarse tolerance but vanishes at scale. Initialization time (mostly device memory allocation and Thrust vector setup) is typically 3–7% of runtime at $\varepsilon = 0.005$, but less than 1% at $\varepsilon = 0.001$. This decay reflects a fixed per-iteration overhead that becomes amortized when the kernel performs billions of timesteps. The scaling also explains why initialization appears more prominent for small problems: the GPU data structures have near-constant size, but the work they support scales with the sample count.

cuRAND setup time exhibits two qualitatively different regimes. For AAPL at $\varepsilon = 0.005$, cuRAND setup dominates the runtime ($> 50\%$). This is because the problem runs so few timesteps per sample that RNG initialization becomes the primary cost. In contrast, for smaller ε values, simulation work expands so dramatically that RNG overhead is amortized to less than 1% of runtime. This nonlinear transition shows that random number generation is not intrinsically expensive, but disproportionately burdens small problem sizes.

6.2 PSC

6.2.1 Final Speedups

Table: Final Speedup vs. Sequential (PSC) (1 GPU)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	42.448×	817.468×	2417.736×
TSLA	413.593×	968.922×	2699.163×
SPY	483.982×	1089.535×	2591.166×

Table: Final Speedup vs. Sequential (PSC) (2 GPUs)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	24.081×	1335.994×	4010.843×
TSLA	516.079×	1596.455×	3931.381×
SPY	519.825×	1825.066×	4815.201×

Table: Final Speedup vs. Sequential (PSC) (4 GPUs)

	ε =**0.005**	ε =**0.0025**	ε =**0.001**
AAPL	15.577×	757.495×	3988.724×
TSLA	383.186×	1246.866×	4575.523×
SPY	389.722×	1350.550×	5430.051×

With 1 GPU we have an average 1280× speedup, with 2 GPUs an average 2063× speedup, and with 4 GPUs an average 2015× speedup. We observe that the speedup generally increases as computational intensity increases (as ε decreases). This makes sense because smaller ε leads to more work the kernel has to do, which increases the benefits our of parallelization. Additionally the scalability with number of GPUs also generally improves as ε decreases, which we'll discuss in more detail in the scalability section.

An anomaly is the AAPL $\varepsilon = 0.005$ case which has a significantly smaller speedup. When we analyzed the runtime in components we found that the majority of time belonged not to array initialization, reduction, kernel runtime, or curand setup, but the other category. This would hint to us that the MPI setup, setting the cuda device, or some other initialization overhead is what's causing the low speedup in this case.

6.2.2 Comparison with GHC

Table: Percent Change in Speedup going from GHC to PSC

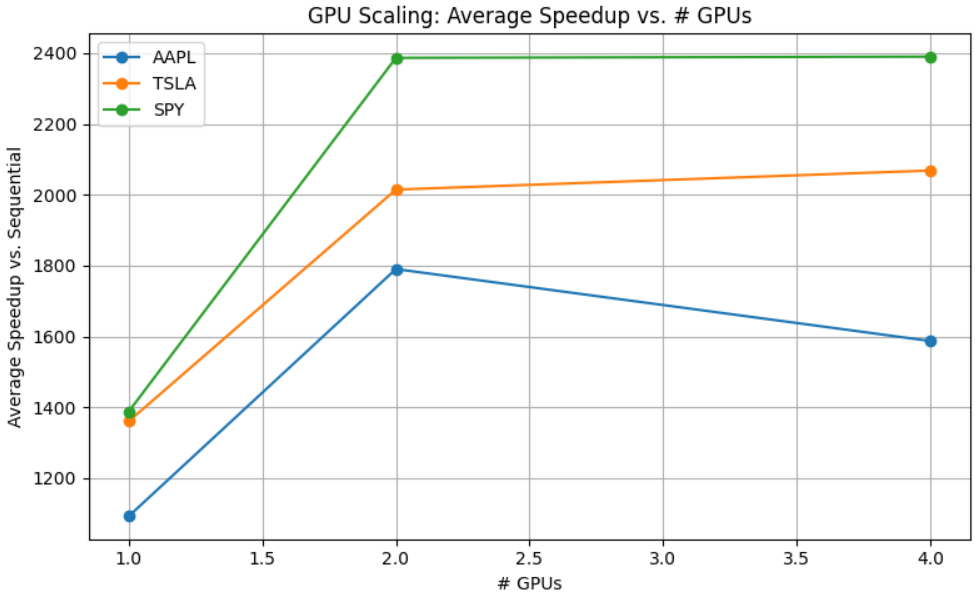
	$\varepsilon = 0.005$	$\varepsilon = 0.0025$	$\varepsilon = 0.001$
AAPL	5.18%	507.5%	1313.0%
TSLA	297.9%	550.0%	1420.0%
SPY	341.5%	602.2%	1352.0%

Although PSC and GHC execute the same MLMC GPU code, PSC consistently achieves significantly higher speedup across all test cases and tolerances. This is because PSC is equipped with an NVIDIA Tesla V100-16 GPU, whereas GHC uses an NVIDIA GeForce RTX 2080 B GPU.

The V100 is a datacenter-class accelerator designed for high-throughput scientific computing, providing substantially higher memory bandwidth, lower memory latency, higher double-precision throughput, and more hardware for compute and memory in general.

As the workload grows, performance becomes increasingly bounded by raw memory and compute; consequently, the V100 continues to scale while the 2080 quickly saturates.

6.2.3 Scalability



For the above plot we've averaged the speedup over $\varepsilon \in \{0.005, 0.0025, 0.001\}$ to compute each data point. We observe that the scalability is very strong from 1 to 2 GPUs, for example with SPY we get a 1.7× speedup from 1 to 2 GPUs. The speedup is positive, but small with 2 to 4 GPUs with the TSLA and SPY case although we experience a slowdown in the AAPL case when we go from 2 to 4 GPUs.

The strong speedup from 1 to 2 GPUs is because we double our total number of threads and compute resources which significantly speeds up our kernel runtime as we discussed in the results of our initial approach section.

The reason the speedup is less significant and in one case we experience a slowdown when we go from 2 to 4 GPUs is by that time the kernel already takes very little time. Our approach speeds up the kernel step but not the reduction, array initialization, or curand setup sections. Because the kernel time is already very little time, although we successfully speed up the kernel time almost linearly (average 3.74× over 1 GPU) the total computation is limited by Amdahl's law since the other steps are not sped up. The slowdown may occur because the other steps have to be computed by all devices and there may be stragglers.

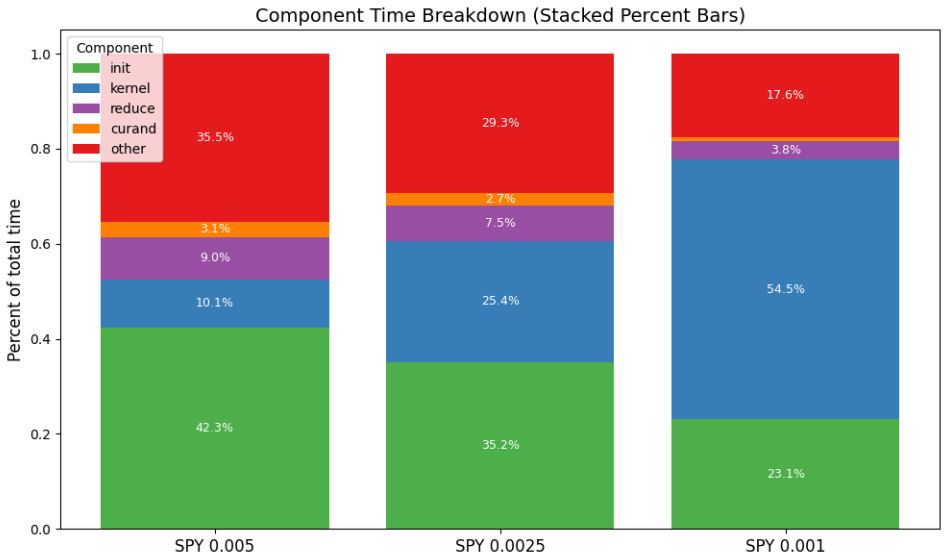
6.2.4 Speedup Limitations

Speedup is limited by the following:

- MPI Communication. We send one message around the ring and do one MPI reduction per iteration.
- cuRAND initialization. Initializing the independent cuRAND states for each thread is constant overhead.
- Inherent sequential steps. Iterations happen sequentially. Although we parallelize taking many samples in parallel we do not know the number of samples we need to take at the next iteration until the current iteration completes. Therefore there is some inherent sequential work in there being multiple iterations. Additionally, checking for convergence and such steps are also sequential, although they are very little of the total computation time as we discuss in the component analysis below.
- Synchronization after kernel run. We need all samples to be fully computed before we call the reduction. We call cuda device synchronize to ensure this. This barrier introduces a sequential step (we cannot reduce until the kernel completes) which limits speedup.
- Logarithmic reduction. We use Thrust to reduce the samples into an end sum. Reduction is generally logarithmic in speed with number of threads so we don't achieve perfect speedup in our reduction step.

6.2.5 Component Analysis

The graph below is for 2 GPUs for the SPY test case. The data for 1 and 4 GPUs and the other test cases show similar patterns. We've focused on this data for visual simplicity.



We see that as ε decreases that init and other time decreases while kernel time increases. This is because as ε decreases the number of samples taken per level iteration increases. This increases the amount of time the kernel takes to run which is why the percentage shifts be more and more heavy on the kernel runtime. init corresponds to initializing the Thrust GPU arrays each iteration. This step initially takes a significant percentage of the time because the max size of these arrays is constant ($= MAX\_THREADS$) even if the number of total samples taken per slot is small (low kernel percent time).

cuRAND setup and Thrust reduction time are both relatively small parts of the computation and become smaller as ε decreases.

Finally the other time percentage decreases as ε decreases as well. Other corresponds to checking for convergence, MPI communication, and any other overhead. This is initially significant when the number of samples taken is relatively small but as we take more samples this overhead becomes less significant.

We can also observe that because the other category is relatively significant in all cases ($> 15\%$ in all cases) that this is one of the limitations to our total speedup.

7 References

- A massively parallel implementation of multilevel Monte Carlo for finite element models, Mathematics and Computers in Simulation, Santiago Badia, Jerrad Hampton, Javier Principe. 2023.
- On the implementation of multilevel Monte Carlo simulation of the stochastic volatility and interest rate model using multi-GPU clusters, Harold A. Lay, Zane Colgin, Viktor Reshniak and Abdul Q. M. Khaliq. May 2, 2018.
- Efficient Asian option pricing with CUDA, A. Yuzhanin, I. Gankevich, E. Stepanov and V. Korkhov. 2015.
- Pricing American Options with Least Squares Monte Carlo on GPUs, Massimiliano Fatica and Everett Phillips. November 18, 2013.
- An introduction to Multilevel Monte Carlo methods. Mike Giles. April 18, 2023
- Improved multilevel Monte Carlo convergence using the Milstein scheme, Mike Giles. December, 2006.
- Multilevel Monte Carlo software. Mike Giles
- Using GPUs for Monte Carlo and Finite Difference Computations. Mike Giles. 2014.
- Analysis of parametric and non-parametric option pricing models. Qiang Luo, et al. November 2022.
- Multilevel Monte Carlo Simulation for Options Pricing. Pei Yuen Lee. 2011.

8 Work Distribution

Kyle and Matt shared the work 60% (Kyle) : 40% (Matt).

Work Completed by Kyle

- Starter Code and Literature Review
- CUDA Level 0 Parallelization
- CUDA Sample Level Parallelization
- OpenMPI Multi-node Parallelization
- PSC Analysis

Work Completed by Matt

- CUDA Timestep Level Parallelism
- CUDA Timestep and Sample Parallelization with Threshold
- GHC Analysis
- GHC vs PSC Discussion