CUDA-accelerated Monte Carlo for HPC Applications in Exotic Option Pricing

Ahmer Nadeem Khan

21 March, 2025

Abstract

Many Monte Carlo methods are embarrassingly parallel due to independence, and can be computationally dense, and thus benefit greatly from the GPU architecture. The canonical Monte Carlo problem is the generation of pseudo-random numbers (PRNGs), in particular numbers from the Normal distribution, which is a key component in financial simulations. Various non-standard methods exploit the latest generation of fully programmable GPUs for efficiency, including the Ziggurat and the Wallace Method, or other hybrid generators. Standard methods like Box-Muller are still robust in the GPU framework, but speed-accuracy trade-offs still need to be considered. We investigate first the transferability of Monte Carlo from sequential to parallel execution, and then discuss the pricing of exotic options such as Asian, Look-back, and Barrier options as an example. We are primarily interested in the CUDA platform, and discuss details of implementation, experiments and numerical results on this framework.

1 Introduction to Parallel Processing

1.1 GPU vs CPU

The Graphics Processing Unit (GPU) provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU. The CPU is optimized to perform sequential tasks, called a *thread*, as efficiently as possible, and can perform a few tens of these tasks in parallel, but the chip resources and transistors in the GPU are designed to perform thousands of these in parallel. There are more resources dedicated to computations (e.g. Floating Point Operations, *FLOPS*) in a GPU rather than data caching and flow - thus each single thread is slower, but the overall cost is amortized because of parallelism. Thus, GPUs have a *massively parallel* nature.[2]

The CPU design has *latency cores*, with the goal of minimizing the latency of each thread so there are big on-chip caches (to fetch data faster), and a very sophisticated control logic (e.g. branch prediction). The GPU instead is throughput-orientated, with many big ALUs (Arithmetic Logic Units) included in the architecture, which rely heavily on SIMD (Single Instruction, Multiple Data) based logic to execute the same instructions ("code") on multiple pieces of a data (e.g. array elements). This works perfectly for Monte Carlo simulations. GPUs also provide several advantages over CPU clusters for HPC applications - CPU clusters use more energy and space, while GPUs are small, fast and use a fraction of the energy used by CPU clusters. [7]

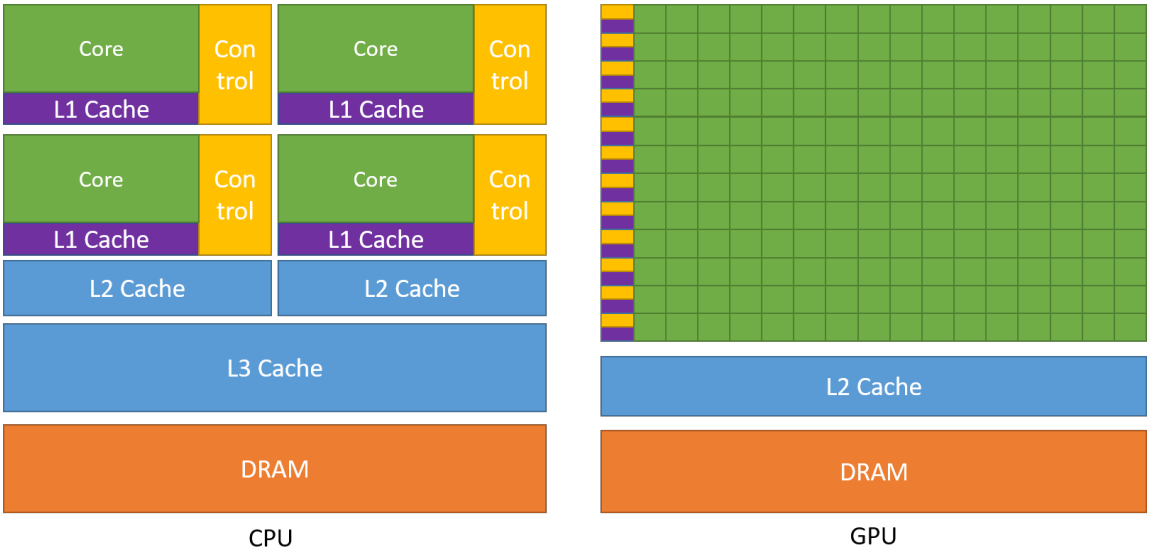

Figure 1: GPU dedicates more transistors to Data Processing

1.2 CUDA Basics

In November 2006, NVIDIA introduced **CUDA** (Compute Unified Device Architecture) as a general purpose programming platform and model for optimized parallel processing usage in various computational applications e.g. in AI, Physics, Graphics, and High-Performance Computing. It comes equipped with a toolkit, libraries, and headers/API to interface with code. CUDA allows us to easily and efficiently exploit the hardware capabilities (i.e. the parallel compute engine) that the GPUs provide. With CUDA, we can define and launch kernels using threads (organized into blocks or grids), and in many cases, we can have high-level parallelism for our code. It also provides memory management and libraries for different applications, as well as performance evaluation methods. The primary language for many CUDA based code is C++. The hierarchy of thread groups, shared memories, and barrier synchronization are the key abstractions at the core of CUDA. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. Thus, CUDA guides the program into a subdivision into independent threads or blocks of threads, and sub-problems that are solved in parallel. [2]

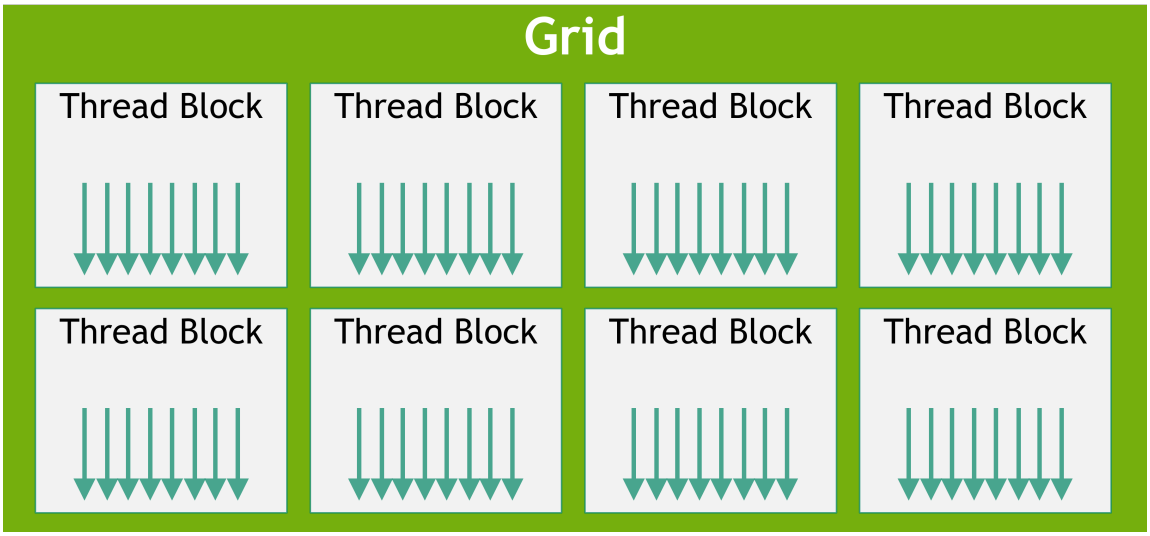

Figure 2: Abstraction of Thread Blocks

CUDA C++ extends C++ by allowing the programmer to define C++ functions, called *kernels*, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same streaming multiprocessor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. You can have 1,2, or 3 dimensional blocks and the indexing is handled using CUDA functionality. Thread blocks are required to execute independently. It must be possible to execute blocks in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order and across any number of cores; this allows for easy scalability. [2]

2 Monte Carlo Methods

2.1 Crude Monte Carlo

A Monte Carlo simulation in Finance, in particular pricing of exotic options, usually entails a particular model of a financial asset (e.g. a stock), and an expression of the value of a derivative (e.g. an option) in terms of an expected value of an equation involving a stochastic component (given by the particular model). The easiest example is the price of a European Call in the Black-Scholes model of a stock which uses Geometric Brownian Motion (GBM), notwithstanding analytic solutions given by the Black-Scholes equation. Exotic options usually do not have analytic solutions, thus the need for Monte Carlo simulations. The (forward) Euler discretization of the log-normal stock model in BS is given by the following equation:

$$\log S(t_i) - \log S(t_{i-1}) = \left(\mu - \frac{\sigma^2}{2}\right)(t_i - t_{i-1}) + \sigma\left(W(t_i) - W(t_{i-1})\right)$$

where

$$W(t_n) - W(t_{n-1}) = \mu(t_n - t_{n-1}) + \sigma\sqrt{t_n - t_{n-1}}\, Z_n, \quad \text{where } t_0 = 0 \text{ and } W(0) = 0$$

is the random-walk construction of Brownian motion (with drift). Both these recursively generate the stock path. Note that for implementation, these only require floating point operations, once the numbers Z_i (independent samples/realizations from the standard normal variable) are available: this is the primary complexity bottleneck for our computations. More directly, the Log-Normal equation becomes:

$$S(t_i) = S(0) \exp\left(\left(r - \frac{\sigma^2}{2}\right) t_i + \sigma\sqrt{t_i} Z_i\right)$$

In a Monte Carlo simulation, we compute the value of the aforementioned expression by simulating many independent paths (e.g. stock paths) and then computing averages. We can already guess the *embarrassingly parallel* nature of these simulations.

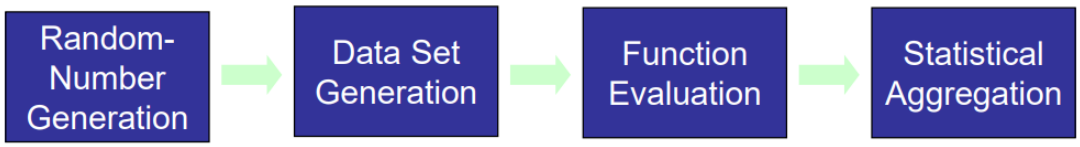

Figure 3: Elements of a Monte Carlo Method [5]

2.2 Exotic Options

Consider pricing a **Down-and-In Barrier Binary Call option** i.e. the contract comes into being only if the stock price crosses a barrier below some fixed price called the *barrier* in between the current time and expiry time for the option, and the payoff is \$1 in the money or \$0 out of the money. The payoff function becomes

$$\theta_{Knock-in} = \mathbf{1}\{S(T) > K\} \cdot \mathbf{1}\left\{\min_{1 \le k \le m} S(t_k) < H\right\}$$

where K is the strike price, and H is the barrier. The stock price is observed at $t_1, t_2, \ldots, t_m = T$. We assume the stock price follows the lognormal model.

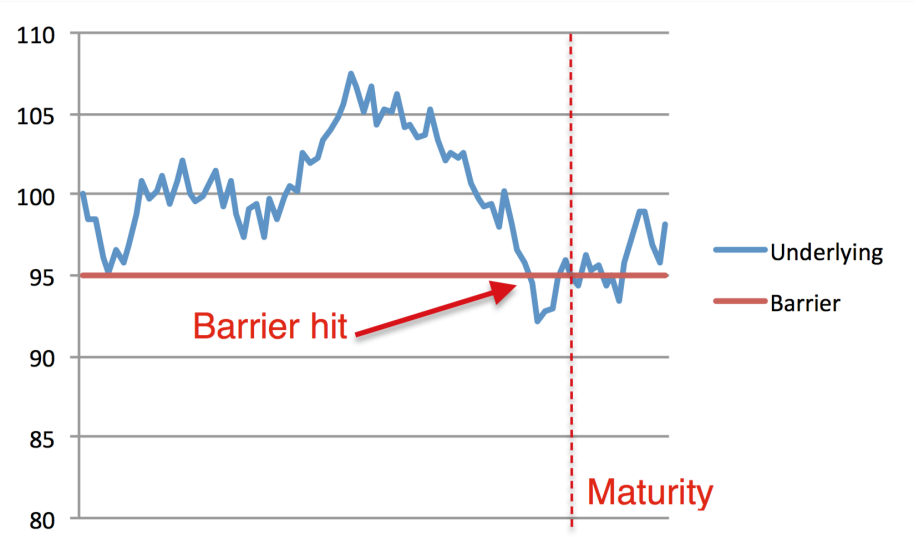

Figure 4: Knock-in Option (Down-and-In)

The (crude) Monte Carlo approach is obvious. After estimating and fixing parameters, simulate M independent stock paths using N time-steps for each path. Then, compute the payoff function for each path (this is simply two *if* conditions, and can be optimized using the recursive path generation). Each path gives an independent realization of the possible payoff $\theta_N^{(i)}$, and to compute the expected payoff, we simply average:

$$\text{Price} = \frac{1}{M}\sum_{i=1}^{M}\theta_N^{(i)}$$

Note first that the *Law of Large Numbers* guarantees that we converge to the price given by risk-neutral theory (i.e. that this is an unbiased estimator). Also, for these types of options in particular, there is concern about the larger variance of our estimates, so variance reduction methods are typically employed in tandem, but these are not difficult to implement (the order of time complexity is the same) and the primary cost in these simulations is the generation of pseudo-random numbers for each simulation path. Note that in financial applications, it is generally the Normal distribution or t-distribution that is most relevant. CUDA provides the library *curand* that employs efficient PRNGs for this purpose, and we will discuss efficient number generation on GPUs in more detail in the next section.

The final step of statistical aggregation also needs to be optimized for GPUs using parallel sum techniques and exploiting the particular task parallelism used (i.e. nodes, clusters etc.). CUDA Thrust provides a lot of this data handling functionality.

An **Asian option** uses the average price of the underlying to determine the final strike price, so the price depends on both where the underlying price ends up and what path it took to get there. An Arithmetic Asian option has the pay-off function

$$\theta_{Knock-in} = \max(\bar{S}_A - S_T)\,,\, 0)$$

where T is the time of expiry and

$$\bar{S}_A = \frac{1}{T'} \sum_{i=t_0}^{T'} s_{t_i}$$

where $t_0, \ldots, T'$ are discrete observation times between the time of conception and expiry for the option. For the option to be in the money at T, only the average value of the underlying stock needs to be larger than the final value. The Monte Carlo pricing approach is readily applied to this type of option as well. Note that we can track a sum variable using pre-defined observation times and then perform a division to find the average to avoid local memory overhead per thread computation.

A **Lookback option** is another path-dependent option where we use the maximum or minimum price of the stock over the lifetime of the option in the payoff e.g. the payoff may look like

$$\theta_{Lookback} = \max \left(\max_{0 \leq t \leq T} (S_t - S_T), 0 \right)$$

for a continuous maximum variant. This offers the maximum advantage to the holder since they need not worry about timing of exercise (compared to the American analogue) and are guaranteed the maximum payout over that period (e.g. for the call or the put). Another variant of the Lookback option has the payoff determined by the sum of the positive differences between the asset price at each time step, and the final asset price. This can be thought of as drawing a horizontal line across the asset path from the terminal price and using the area above the line and below the path as the payoff:

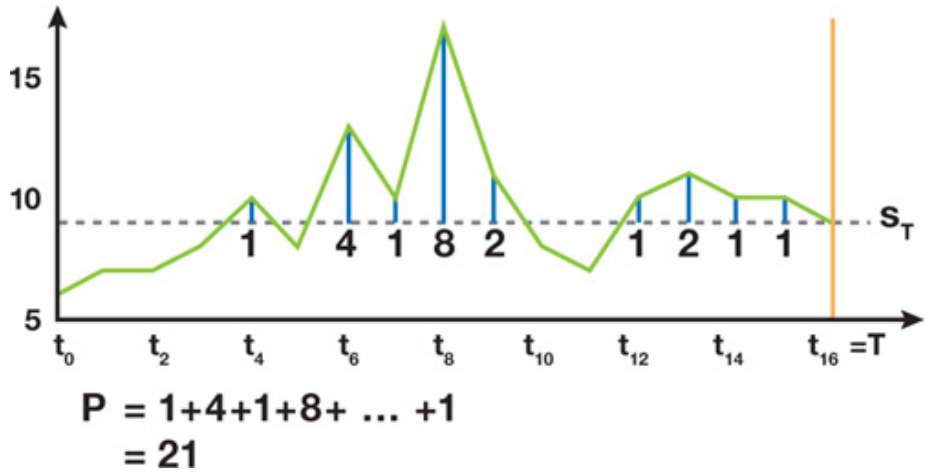

Figure 5: Lookback Payoff Calculation [3]

The discrete variant payoff becomes:

$$\theta_{Lookback-varient} = \sum_{i=0}^{T-1} \max \left(S_{t_i} - S_T, 0 \right)$$

In the cited paper, the GARCH model was used to price the variant of the Lookback options instead of the log-normal random walk we described earlier. We will not discuss this model here, but the The task-parallelism implementation for pricing is very similar. [3]

2.3 Parallel Paradigms

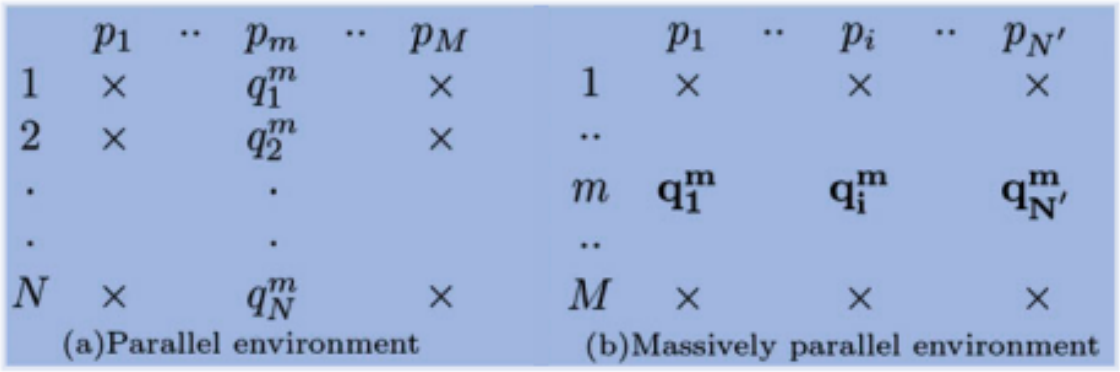

Figure 6: The two parallel processing paradigms [7]

Since each estimate in an MC simulation is generated independently, we can assign the m-th processor (i.e. the m-th thread) to compute the estimate θ_N^m, so that each thread has depth N. This can be called the *parallel* environment. [7] To do this, each thread requires a full stream of N independent samples from the standard normal distribution, which can be pseudo-random or quasi-random (note that in applications N is usually in the millions, and M is big enough for statistical accuracy, e.g. 50-100). Methods like leapfrogging, blocking or parameterization can be employed as well to achieve this. For example, in leap-frogging, a single stream is generated but it is sequentially assigned to threads one-by-one so that each thread obtains a sample realization.

In a massively parallel environment, depicted by the second diagram in Figure 4, where the number of processors N is much larger than M, it can be a lot more efficient to completely *transpose* our computing strategy. Now the processors $p_1, \ldots, p'_N$ run simultaneously (for a total of M times) to generate the sequence $\boldsymbol{q}_1^1, \ldots, \boldsymbol{q}_M^N$ to compute $\boldsymbol{\theta}_m^N$, as $m = 1, \ldots, M$, where $\boldsymbol{q}_m^i$ is part of the sequence $\{q_i^m, q_{N+i}^m, q_{2N+i}^m, \ldots\}$ which is assigned to the ith processor.

The choice of the two paradigms, which we call *parallel* and *massively parallel*, determines how the underlying sequence (pseudo-random or low-discrepancy) should be generated. In the parallel paradigm, a recursive algorithm for generating the underlying sequence works best since each processor generates the entire sequence. This paradigm is appropriate for a computing system with distributed memory, such as a cluster. For the massively parallel paradigm, a direct algorithm that generates the nth term of the sequence from the index/counter n is more appropriate (sometimes called 'counter-based' since it computes the term at the value of a counter in a sequence). [7]

3 Efficient Parallel Random Number Generation

3.1 Parallel Random Number Generators

The methods to first obtain pseudo-random or quasi-random sequences from the Uniform distribution and then applying transformations or algorithms to obtain samples from the Normal distribution, or methods to obtain Normal samples directly differ in their considerations of accuracy, speed, convergence, and reliability. This discussion is then ultimately affected by implementation

on specific hardware architecture like CPUs or GPUs (or FPGAs in older literature).

Since Monte Carlo simulations involve dense numerical calculations (as we will see for generating Normal samples) and are embarrassingly parallel, they are ideal for the GPU architecture. Random number generators must meet the conflicting goals of being extremely fast while also providing random number streams that are indistinguishable from a true random number source. [3] Thus, there is always a speed-accuracy tradeoff for different methods, where quality/accuracy is measured as randomness through statistical tests. The major quality challenge is avoiding artifacts, predictability, and correlation among different threads, blocks, cores and even nodes. Design of tractable algorithms must also be considered. The advantages of PRNGs and Monte Carlo on GPUs is speed, and the co-location advantages of the data which are exploited to minimize latency. Different memory structures are essential to consider as well. There are many choices for Parallel Random Number Generators, with traditional choices including XORWOW, Sobol, Tausworthe, RASRAP, Mersenne Twister etc. [7] Classical methods like LCGs, Lagged Fibonacci Generators usually have poor statistics, and methods with very good statistics like Mersenne are usually too slow. [3] There is an extensive body of literature discussing these generators (usually assuming a sequential implementation).

For many applications, combined or hybrid number generators can also be implemented to compensate for the defects of one with another RNG (e.g. the Combined Tausworthe + LCG generator). Due to the massive compute available, it is generally not possible to pre-compute the sequences, so this is done on the fly to reduces the space complexity of our algorithm. [6]

We have two main general requirements that we wish PRNGs to satisfy [3]:

- **A long period:** Every deterministic generator must eventually loop, but the goal is to make the loop period as long as possible. This is usually the first consideration.
- **Good statistical quality:** The output from the generator should be practically indistinguishable from a true RNG of the required distribution, and it should not exhibit any correlations or patterns. Poor generator quality can ruin the results of Monte Carlo applications, and it is critical that generators are able to pass the set of theoretical and empirical tests for quality that are available. Numerous statistical tests are available to verify this requirement coming from canonical papers of Knuth 1969, Marsaglia 1995, and L'Ecuyer 2006. Knuth recommends about half a dozen tests with certain tests preferred over others.

The parallel implementation further require [3]:

- **The ability to generate different substreams on parallel nodes.** Each node must be given a different portion of the random stream with no overlap. This allows parallelism in the first place.
- **No correlations between substreams on different nodes.** The substreams must appear to be completely independent streams of random numbers. Independence is crucial to obtain theoretical guarantees about convergence.

We discuss two main methods to generate samples from the Normal distribution from the Uniform distribution discussed in the main paper [3]:

1. **Combined LCG + Tausworthe Generator** to produce Uniform samples, and then applying the Box-Muller Transform to generate Normal samples.
2. **The Wallace method** to generate Normal samples directly.

3.2 Hybrid Generators

The Tausworthe generator can be considered as part of a family of generators including the Mersenne Twister. Internally, the Mersenne twister utilizes a binary matrix to transform one vector of bits into a new vector of bits, using an extremely large sparse matrix and large vectors. The Tausworthe generator uses much smaller vectors, of the order of two to four words, and a correspondingly denser matrix. For example, the four-component LFSR113 generator from L'Ecuyer (1999) requires 27 instructions, producing a stream with a period of approximately 2^{113}. This particular generator has bad correlation properties. [3] The LCG-based generators by themselves are also computationally dense due to the prime moduli computations on each step.

However, combining these generators can compensate for these defects. If the periods are coprime, then the combined period is the product of the respective periods. Further if a single LCG with $m = 2^{32}$ is used, then single precision arithmetic automatically applies the prime modulo operation by truncation of bits. A particular example we will observe results for is the three-component combined Tausworthe "taus88" from L'Ecuyer 1996 and the 32-bit "Quick and Dirty" LCG from Press et al. 1992. The combined generator has very good statistical properties, require 4 generator values (initial values) and an overall period of 2^{121}. Thus each thread should receive 4 independent state values, and various stream-skipping techniques can be employed to propagate these values as well. [3]

This generates numbers from $Unif(0, 1)$ and we then apply the Box-Muller transform to two uniform random numbers to produce two numbers from the standard Normal distribution. The algorithm is as follows:

Algorithm 1 Box-Muller: Pseudo random Normal Generator

Input: Two independent random numbers from $U(0, 1)$
Output: Two independent standard normal random variables

1: Generate $u_1, u_2 \sim U(0, 1)$, independently
2: Compute $R^2 \leftarrow -2 \log u_1$
3: Compute $\theta \leftarrow 2\pi u_2$
4: Compute $X \leftarrow \sqrt{-2 \log u_1} \cdot \cos(2\pi u_2)$
5: Compute $Y \leftarrow \sqrt{-2 \log u_1} \cdot \sin(2\pi u_2)$
6: **return** X, Y

It can be proved using the Jacobian of the above transformations that this does indeed generate Normal samples. Note that the Box-Muller was historically discarded on CPUs due to the evaluation cost of the transcendental functions involved: $\cos, \sin, \log$. However, it offers a number of important advantages for a GPU implementation with batched threads, the most obvious of which is that it has no branching or looping (which we will see in detail later): there is only a single code path. It also does not require any table look-ups, or the large numbers of constants found in some methods, e.g Beasley-Moro-Springer. It still has a fairly high computational load, but this is precisely why GPU usage is appropriate.

3.3 The Wallace Method

The fastest method in software to generate Normal samples directly is the Ziggurat method (Marsaglia 2000). This procedure uses a complicated method to split the Gaussian probability density function into axis-aligned rectangles, and it is designed to minimize the average cost of generating a sample.

[1] We will discuss why the Ziggurat method is unsuited for GPU use in the next sub-section.

The Wallace Gaussian generator (Wallace 1996, Brent 2003) is a novel method that is able to generate Gaussian samples directly, without using a uniform generator first. The central idea is to take a pool of k random numbers that are already Gaussian distributed and then apply a transform to the pool, such that another pool is produced that is also Gaussian. After each transform the pool can be output to supply random numbers. The key requirement is to make the output of the transform as uncorrelated with the input as possible, called mixing, while still preserving the distribution properties. [3]

Due to the maximum entropy property, the transformation we use is linear, and does not require transcendental function evaluations like the Box-Muller transform. On each *pass*, a transformation is applied to the current pool of numbers. Ideally, for k numbers, we apply a new $k \times k$ matrix A to the vector of numbers such that A is an orthogonal transformation; an orthogonal transformation is norm-preserving, so the mean-squared sum L for the numbers is preserved. If X is the old vector of Gaussian-distributed numbers, then a new vector is given by $Y = AX$. [4]

However, there is a computational problem, because producing a random $k \times k$ orthogonal matrix is very expensive — orders of magnitude more expensive than just generating k Gaussian-distributed numbers using traditional techniques. Instead, the approach used is to construct an orthogonal transform using lots of much smaller orthogonal transforms, for example by using 2×2 orthogonal matrices, such as 2D rotations. Applying $k/2$ 2D rotations will preserve the length of the overall k vector, but it is much cheaper to implement. The drawback is that the degree of mixing between passes is much lower: by using a full $k \times k$ matrix, each value in the new pool can be influenced by every value in the old pool, but if 2×2 matrices are used, then each value in the new pool depends on at most two values in the old pool. In the cited paper, a size of 4 was used as well, and was still computationally manageable. [3] Note that algorithmically this procedure is achieved by $Y = AX$ just as before, but now A is a block-diagonal matrix where the blocks are dense, orthogonal and of size $j << k$. The overall sparsity amortizes the numerical computations significantly.

The use of smaller orthogonal transformations means that we need to use block orthogonal transformations so the full size is still $k \times k$, but the production of such a matrix ix still easy. To mitigate the lack of mixing this causes, we can use an idea from RASRAP [7] by applying a permutation to the k-vector X before every pass of the method. This will ensure that in effect, and after sufficient passes, each new vector component depends on all previous ones, once we ensure sufficient quality of permutation (note that this will be an independent cost, just as in RASRAP). Thus one step of the algorithm becomes

$$Y = AP_iX$$

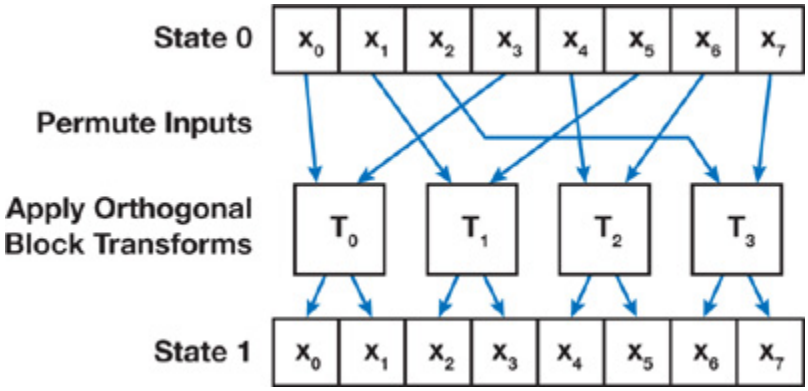

Figure 7: One Step of the Wallace Method [3]

A second fix is to apply the pass multiple times before using it to output numbers as the new pool, so we get

$$Y = A^r X$$

where r is the number of passes applied before an output is generated.

In practice, both mixing methods have their drawbacks, since it is expensive to set up a fully random permutation on every step, and multiple passes are obviously taxing on the speed. Typically, a combination of both is used, where some randomness of the permutation is ensured, and a small number of passes are done on each step i.e. [3]

$$Y = A^r P_i X$$

The quality of the Wallace method depends heavily on the quality of the permutation; using the unique thread ID for each thread and passing it into a low-quality LCG actually works well enough to provide a random permutation for each thread while preserving the uniqueness. A *Walsh-Hadamard matrix* can be used in place of A. Initialization of the random number pool is a separate process from the Wallace generator itself and hence requires a separate random number generator. This initialization can be performed on the CPU: for example, by using the Ziggurat method in concert with a Mersenne twister. Although this generator will not be as fast as the GPU-based Wallace generator, it needs to provide only the seed values for the Wallace pools; once the threads start executing, they require no further seed values. The Wallace methods in summary, provides the following advantages: [3]

- Direct generation of normally distributed values
- A state pool that can be operated on fully in parallel, allowing high utilization of processor-local shared memory resources.

Note however that when the limited local shared memory is needed for simulation in its entirety, the Wallace Method is not appropriate. We can of course make use of constant Global memory, but this most likely has a significant adverse effect on performance. We also see limits on the number of registers we can use, because the larger computation complexity of combining the Wallace transform with the simulation loops increases the register requirement, but this can likely be addressed by reducing thread count. [3]

4 GPU-related Optimization

4.1 Wrap Divergence

Due to the latency management in GPU using low-level multi-threading, it is prone to issues of branching or looping that can harm performance. We have thousands of active threads in a single processor so we want to ensure that stalling is minimized, and that when it does occur, we can schedule a different thread quickly. A thread wrap is abstracted as a group of 32 threads within a block in NVIDIA GPUs. The threads are assigned to wraps using indexing, e.g. in CUDA as we discussed before. Warp branching occurs when threads in the same warp take different paths, in an if or switch statement for example. Consider the simplified code [6]:

```
// Example of warp divergence
if (threadIdx.x % 2 == 0)
    doA();    // even threads
else
    doB();    // odd threads
```

In this case, half the wrap runs "doA()" while the other half waits. Then this switches. This halves performance, and this is known as wrap divergence. To avoid this, we need to write control logic where all 32 threads in a warp follow the same path, and shared memory conflicts are avoided. We want to keep the threads in a single wrap on the same path for computations.

Another issue is looping. In Monte Carlo, consider the simplified acceptance-rejection algorithm, [6] introduced by John Von Neumann (1951):

```
u = UnifRng();
x = Candidate(u);
if (Accept(x))
    return x;
else
    return Slow();
```

A cheap test is done for acceptance, but whenever rejection occurs, a slower route is taken e.g. if the thread has to loop over again to generate a number. The time for the warp is total time to cover paths of all threads, and since some paths can take the slower route, this increases the total time for the wrap [6]:

Thread 0

```
x = Candidate()
    ;
if (Accept(x))
    return x;
else
    return Slow
    ();
```

Thread 1

```
x = Candidate()
    ;
if (Accept(x))
    return x;
else
    return Slow
    ();
```

Thread 2

```
x = Candidate()
    ;
if (Accept(x))
    return x;
else
    return Slow
    ();
```

Thread 3

```
x = Candidate()
    ;
if (Accept(x))
    return x;
else
    return Slow
    ();
```

In particular, since the Ziggurat method minimizes the average cost of generating a sample. This means that for say 2% of generated numbers, a more complicated route using further uniform samples and function calls must be made. In software this is acceptable, but in a GPU, the performance of the slow route will apply to all threads in a warp, even if only one thread uses

the route. If the warp size is 32 and the probability of taking the slow route is 2%, then the probability of any warp taking the slow route is $(1 - 0.02)^{32}$, which is 47%! Because of thread batching, the assumptions designed into the ziggurat are violated, and the performance advantage is destroyed. Similarly, the Polar Transform (Press et. al. 1992) is simple and relatively efficient, but the probability of looping per thread is 14%. This leads to an expected 1.6 iterations per generated sample turning into an expected 3.1 iterations when warp effects are taken into account. [3] Non-rejection algorithms are usually better for GPUs, e.g. Box-Muller.

4.2 Function Approximation and Floating Point Arithmetic

Consider the CDF inversion problem in Monte Carlo. For many distributions, we can directly compute the inverse on uniform numbers to generate from the required distribution (when the function is available and feasible). But even then, we run into finite-precision issues. Let ϕ be the inverse cdf function of a variable. Then for the lower and upper bounds, we get

$$\phi(0) = -\infty$$
$$\phi(1) = +\infty$$

For the lower bound, we can simply add a buffer to the entire sample, say 2^{-33} in single-precision, to ensure no run-time errors. For the upper bound this will not work, since $U = 2^{32} - 1 + 2^{-33} = 1$ in single-precision. Double-precision may be used, but this slows down memory caching (since words are now twice the length). In many modern systems, this is not as much of an issue anymore. Due to the floating-point arithmetic we also get degrading resolution for larger intervals with larger mantissa sizes.

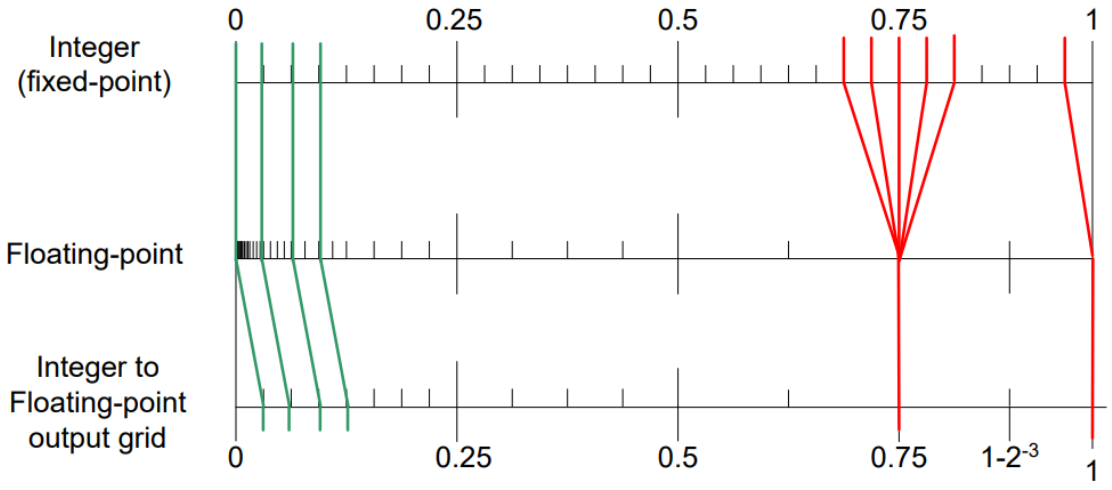

Figure 8: Floating Point Mapping degrades for larger values [6]

For CDF inversion (say for the Normal distribution) this means that the positive tail of the distribution is discretised into only 2^{24} values, and the largest uniform inputs result in infinity – with probability about 2^{-24}. [6] This introduces error into long running simulations, where the upper-tail is not simulated accurately, and the distribution is skewed to have non-zero mean. For RQMC, this means that the effects of low-discrepancy sequences are reduced in the upper half.

A simple solution to this is to first calculate p and check if $p > 0.5$. If it is, then we simply compute from the reflection onto the negative half $(1-p)$, and then use the sign change to calculate the upper tail value. This does hinder performance, but preserves tail behavior. In many cases, double precision removes the issue, and in many applications, tail preservation is crucial.

4.3 Memory

There are two types of memory in the GPU hierarchy: Shared (local) and Global. Shared memory is very fast (can be treated as registers, but indexed) but is limited and small. Global memory is much larger, but extremely slow. In many applications, this is a significant overhead (e.g. in the Wallace method), and Global memory traffic should be minimized whenever possible. We also get some more levels of memory, e.g constant memory, which caches small part of global memory, and texture memory, which caches a larger part of Global memory. These do not use Global memory bandwidth once the cache is loaded, but we can still run into *cache misses* (as opposed to *cache hits*). There is a significant amount of literature concerning memory usage and optimization in High-Performance Computing, and is beyond the scope of this paper.

5 Results

5.1 Performance Results for the Discussed Random Number Generators and Simulations [3]

We first present the results from the cited paper in GPU Gems 3. This implemented the hybrid Tausworthe RNG with Box-Muller and the Wallace Gaussian RNG in CUDA. The results are measured in millions of Gaussian samples produced across the entire GPU every second (MSamples/s).

Category	**Method**	**Raw RNG** (MSamples/s)	**Asian** (MSteps/s)	**Lookback** (MSteps/s)
GPU	Tausworthe plus Box-Muller	4,327	1,769	1,147
	Wallace	5,274	1,877	–
	Constant RNG	–	5,177	2,908
CPU	Ziggurat plus Mersenne Twister	206	32	50
	Constant RNG	–	44	64
Speedup		26	59	23

Table 1: Performance comparison of random number generation and option pricing methods on GPU vs CPU implementations.

Included in the table are performance results for all four processors of a 2.2 GHz Quad Opteron (CPU), using a combination of the Mersenne twister uniform generator and Ziggurat transform. Columns two and three of Table 1 show the performance of simulations for two of the three exotic options presented in the previous section (Arithmetic Asian and the discussed variant of the Lookback). The Raw RNG column measures just the production of random numbers executing into memory with no simulation being driven. For the constant RNG row, the random number generation was removed with only the simulation code proceeding. This gives us an idea of the ratio of time spent in generating numbers and using them in the simulation.

The performance figures are reported in MSteps/s, which is a measure of the number of simulation time steps that can be processed per second. This includes the stock simulation and aggregation steps for the Asian option, for example. When measuring the performance of the Asian option, each simulation executed 256 time steps, with 256 threads per block; so completing one block per second would provide 65 KSteps/s. In the Lookback case, the number of time steps is limited by the amount of shared memory that can be dedicated to each thread. 512 threads per block were used; thus the Lookback option was limited to 8 time steps per simulation.

In the Asian case, we see that the Wallace generator provides only a modest improvement of 1.06 times, compared to the Tausworthe/Box-Muller generators. Even so, the performance is 59 times that of the CPU implementation. In the Lookback case, the Wallace cannot be used because the shared memory is needed by the simulation kernel, and performance is much lower than that of the Asian option. Interestingly, the CPU implementation exhibits the opposite performance: the Lookback is significantly faster than the Asian. However, a 23 times improvement in speed is seen by using the GPU.

5.2 Barrier Options using CUDA

We present here results for pricing the Down-and-in Barrier Option and simply measure the clock-time difference between using a CPU and a GPU. The terminal output from our implementation is displayed here:

```
****************** INFO ******************
Number of Paths: 5000000
Underlying Initial Price: 100
Strike: 100
Barrier: 95
Time to Maturity: 1 years
Risk-free Interest Rate: 0.05%
Annual drift: 0.1%
Volatility: 0.2%
****************** PRICE ******************
Option Price (GPU): 8.45485
Option Price (CPU): 8.49924
****************** TIME ******************
GPU Monte Carlo Computation: 0.956 ms
CPU Monte Carlo Computation: 9698.87 ms
****************** END ******************
```

We used the **NVIDIA GeForce RTX 4090** Series GPU and the CPU used was **AMD Ryzen 7 7800X3D** 8-Core Processor. The CPU in this case is very powerful. Even then, we see an approximately **x10145** improvement in speed by using **5 million paths**, with each path of length 365 (i.e. daily time-step over a year) with block sizes of 256, and the accuracy is comparable; average difference in prices was about **2-3%**, and this is expected due to the independent Monte Carlo tasks (the prices almost always agreed to the nearest dime). This code was run on C++ using CUDA, and the "curand" library was used directly to generate random numbers for our implementation.

6 Conclusion

Monte Carlo methods have enormous applications in GPU usage for High-performance financial applications. The Wallace Method and Hybrid generators provide reasonable alternatives to traditional PRNGs for GPU-based systems, and contingencies related to architecture and algorithm design must be considered, and speed-accuracy tradeoffs are crucial. The advantages of GPUs were demonstrated using numerical experiments for various methods, problems and hardware, using the CUDA platform provided by NVIDIA, and results from literature were presented and expanded upon. Efficiency and reliability of methods in CUDA libraries was discussed and evaluated.

7 Future Survey

We hope in the future to consider in greater detail methods known as Quantile Approximation Methods. These methods attempt to approximate the Inverse CDF function of a distribution (sometimes called the Quantile of a distribution) to give a direct method to produce samples. The major issues for relevant distributions like the Normal, χ^2-distribution or the t-distribution are numerical considerations like the infinite support of the PDFs, tail degradation, parameter-based issues and the analytical technique used for approximation (e.g. polynomial/trigonometric Chebyshev interpolation) and its specific implementation. Many of these algorithms are also amenable to parallel processing as well. To my knowledge, certain techniques like Polynomial Splines to approximate the Quantiles have not been discussed in the literature.

References

[1] Nguyet Nguyen, Linlin Xu, and Giray Ökten. A quasi-monte carlo implementation of the ziggurat method. *Monte Carlo Methods and Applications*, 24(2):93–99, 2018.

[2] NVIDIA Corporation. *CUDA C++ Programming Guide, Version 12.9*. NVIDIA, 2024. Accessed: 2025-05-01, Introduction section.

[3] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Chapter 37: Efficient random number generation and application, 2007. Accessed: 2025-03-21.

[4] Christoph Riesinger, Tobias Neckel, and Florian Rupp. Non-standard pseudo random number generators revisited for gpus. *Future Generation Computer Systems*, 82:482–492, 2018.

[5] Andrew Sheppard. Cuda-accelerated monte-carlo for hpc: A practitioner's guide. Presentation at SC11 Conference, November 2011. Presented by Andrew Sheppard, Fountainhead.

[6] David B. Thomas. Monte carlo implementations on gpus. Presentation, Imperial College London, 2009. Accessed: 2025-03-24.

[7] Linlin Xu and Giray Ökten and. High-performance financial simulation using randomized quasi-monte carlo methods. *Quantitative Finance*, 15(8):1425–1436, 2015.