GPU Option Pricing

Simon Suo
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
sdsuo@uwaterloo.ca

Ruiming Zhu
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
ruiming.zhu@uwaterloo.ca

Ryan Attridge
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
rjattrid@uwaterloo.ca

Justin Wan
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
jwlwan@uwaterloo.ca

ABSTRACT

In this paper, we explore the possible approaches to harness extra computing power from commodity hardware to speedup pricing calculation of individual options. Specifically, we leverage two parallel computing platforms: Open Computing Language (OpenCL) and Compute United Device Architecture (CUDA). We propose several parallel implementations of the two most popular numerical methods of option pricing: Lattice model and Monte Carlo method. In the end, we show that the parallel implementations achieve significant performance improvement over serial implementations.

Categories and Subject Descriptors

G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*; G.4 [**Mathematical Software**]: Parallel and vector implementations; I.6.8 [**Simulation and Modeling**]: Types of SimulationMonte Carlo

General Terms

Algorithms

Keywords

GPU, option pricing, CUDA, OpenCL

1. INTRODUCTION

Option pricing has been a fundamental activity in the financial sector. The mathematical formulation has been well studied in the literature and has been known as the Black-Scholes model. The key basis of the model consists of the following insights and assumptions:

- The stock price follows geometric Brownian motion,
- One can perfectly hedge the option by buying and selling the underlying asset in just the right way and consequently "eliminate risk", and
- There are no arbitrage opportunities, i.e. all risk-free portfolios must earn the risk-free rate of return.

The Black-Scholes equation, a partial differential equation describing the price of the option over time [1], is then derived by setting the price of a perfectly hedged portfolio to the risk-free rate:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0,$$

where V is the value of the option, r the risk-free rate, and σ is the volatility. Typically, the equation is solved backward in time from the option expiry time $t = T$ to the present $t = 0$.

In practice, it is desirable to compute the value of an option quickly and accurately. In this paper, we explore the use of parallel computing platforms, and in particular, graphics processing units (GPUs), to speedup pricing calculation of individual options.

1.1 Parallel Computing Platforms

In this section, we briefly introduce two parallel computing platforms that are the primary basis of the research topic: OpenCL and CUDA.

Open Computing Language (OpenCL) is "a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors." It is similar to Nvidia's Compute Unified Device Architecture (CUDA) framework but has the advantage of not being restricted to Nvidia GPUs. CUDA, on the other hand, has the advantage of better double-precision support and more mature libraries. Since CUDA is architecturally and

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
WHPCF 2015, November 15-20 2015, Austin, Texas, USA
Copyright 2015 ACM 978-1-4503-4015-1/15/11.
http://dx.doi.org/10.1145/2830556.2830564.

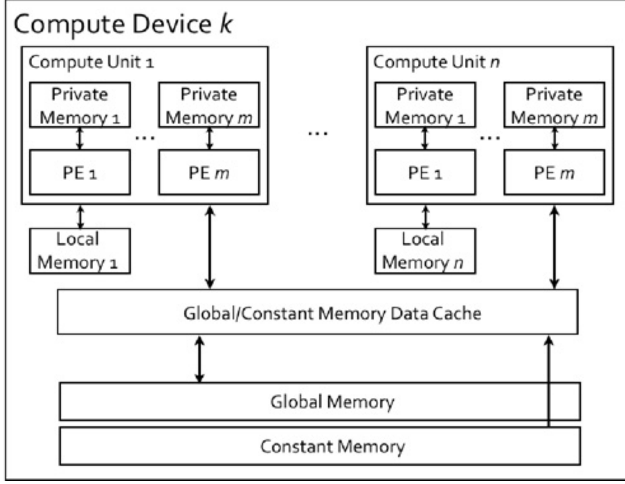

Figure 1: Memory model of OpenCL framework

functionally similar to OpenCL for the purpose of this paper, we will discuss OpenCL as a representative.

OpenCL programs are defined in terms of two components: the host program and the kernel functions. The host program runs on the CPU, sets up connection with and dispatches work to the processing units. The processing units, then, execute the kernel functions to perform data processing.The kernel functions are executed in threads called work-items. These work-items can be assigned to workgroups, which allow intra-workgroup synchronization. Since work items from different work groups cannot be synchronized with each other, work done in each work group needs to be independent of each other. This is important to consider when optimizing the kernel program.

In terms of memory mode, OpenCL employs three levels of memory: global, local, and private. As shown in Figure 1, global memory is accessible by all work-items, whereas local memory is shared within workgroups, and private memory is exclusive to individual work-items.

In this paper, we limit the scope of discussion to GPUs, but the algorithms developed should be applicable regardless of the underlying computing units.

2. OPTION PRICING METHODS

2.1 Lattice Method

The lattice method is one of the most popular numerical method for option pricing. It models the movement of the underlying security with geometric Brownian motion in discrete number of time-steps. As illustrated in Figure 2, the price of the stock has a pseudo-probability of p of going up by factor of u, and $1-p$ of going down factor of d, where p, u, and d are parameters derived from the underlying volatility σ, duration t, and number of time-steps N:

$$u = e^{\sigma\sqrt{\frac{t}{N}}}$$

$$d = e^{-\sigma\sqrt{\frac{t}{N}}}$$

$$p = \frac{e^{r\frac{t}{N}} - d}{u - d}$$

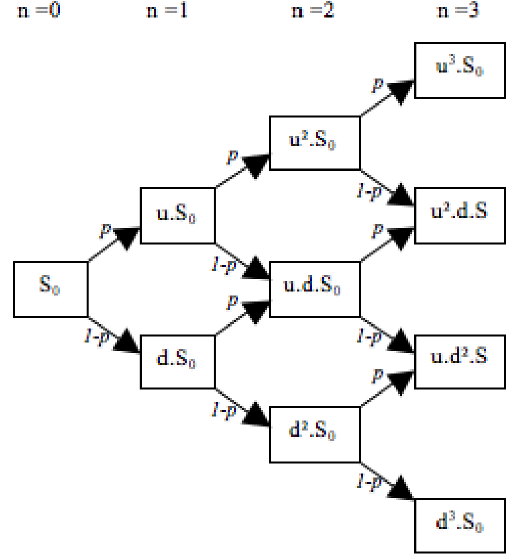

Figure 2: Illustration of lattice tree

The high level algorithm of option pricing with the lattice method consists of two steps. First, we calculate option values at expiry with the payoff formula. For a put option, the values are calculated as:

$$V_j^N = max(K - S_j^N, 0); j = 0, \dots, N$$

Then, we sequentially calculate option values at each preceding node:

$$V_j^n = e^{-r\frac{t}{n}}(pV_{j+1}^{n+1} + (1-p)V_j^{n+1})$$

$$n = N-1, \dots, 0$$

$$j = 0, \dots, n$$

Finally, we obtained the present value of the option as V_0^0. The run-time complexity of the algorithm is $O(N^2)$ with respect to the number of time-steps N.

2.2 Monte Carlo Method

The Monte Carlo method is another popular numerical method for option pricing. Unlike the binomial lattice model, it takes the approach of generating individual paths of asset price movements instead of the entire lattice tree. Each simulation uses a sequence of random walk to arrive at a final asset price, as illustrated in Figure 3.

The price movements are modelled by a geometric Brownian motion with drift, where the change is the sum of the risk-free rate and a random volatility. The random volatility follows a normal distribution with a mean of 0 and standard deviation equal to the asset's volatility. If S_i is the price of the asset at time step i, then:

$$S_{i+1} = S_i + S_i(\mu\Delta t + \Phi^i\sigma\sqrt{\Delta t})$$

where μ is the risk free return, σ is the volatility of the asset's returns, Δt is the size of a time step, and $\Phi^i \sim N(0,1)$.

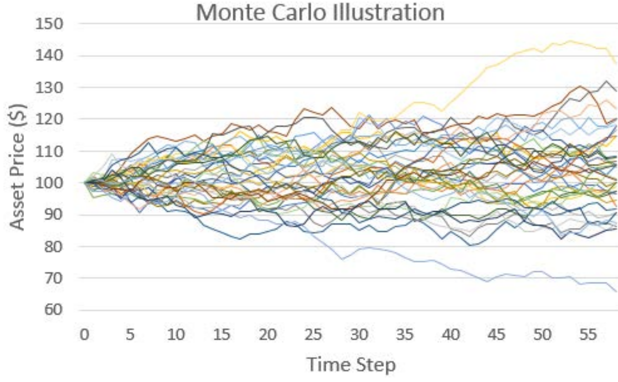

Figure 3: Illustration of Monte Carlo simulations

The present value of the option, then, can be obtained by taking the average option payoff at maturity discounted at the risk-free rate. Due to the forward nature of Monte Carlo method, however, only vanilla European options can be priced by the standard algorithm. American options can be valued by estimating the continuation value of the option and comparing it with the early exercise value. This can be done using least squares regression [2] and other methods.

To extend the standard model, we can consider random jumps in price movements. The jumps can be modelled by the Poisson process, and the jump sizes follow a given normal distribution. Thus, the extended model is a generalization of standard model with non-zero jump probability.

The main use of Monte Carlo is pricing multi-asset options [5]. These are options where the underlying asset is a portfolio of assets and only their aggregate value is considered. Our program can easily be extended to price these options.

3. GPU IMPLEMENTATIONS

3.1 Lattice Method 1: Group

Consider Figure 2 for N time steps. The standard lattice method works backward; i.e. from n=N back to n=0. An intuitive parallel implementation of the lattice method is to group the option values, $\{V_j^N\}$, at final time into pairs and then assign each pair to a thread in GPU. Each thread will then perform one step of the lattice algorithm in parallel to compute $\{V_j^{N-1}\}$. However, in order to proceed; i.e. update V_j^{N-2} in the next time step, thread j would need to communicate with thread $j+1$ to obtain the value V_{j+1}^{N-1}. Thus, it creates a synchronization point in each time step, which can be very costly.

One way to reduce synchronization is to assign each thread more than two option values at final time. Thus, each thread can perform the lattice algorithm for R time steps in parallel before communication occurs. However, it is important to note that the blocks of option values have to overlap with one another. Otherwise, after R time steps, the threads will only complete partially the lattice pyramid computation as shown in Figure 5 with $R = 3$ and two threads. In other words, we reduce communication at the expense of duplicate computation.

More precisely, we use $N + 1$ work-items to calculate the $N + 1$ option values at expiry ($V_j^N; j = 0, \ldots, N$) with the

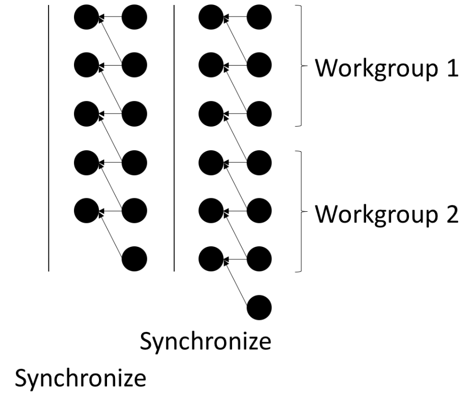

Figure 4: Illustration of intuitive implementation

payoff formula. Now, we implement the kernel function "group-reduce", which performs R reductions (R time steps) on $\frac{n-R+1}{K-R+1}$ groups of $K+1$ option values with $\frac{n-R+1}{K-R+1}$ workgroups of K work-items. After each kernel execution, each V_j^n is reduced to V_j^{n-R} and the number of option values is reduced by R. To obtain the present value of the option, we execute the kernel function $\frac{N}{R}$ times.

The parameter R can be tweaked at run-time to adjust the balance of synchronization frequency and duplicate calculation. With $R = 1$, no duplicate calculation is performed, but we need full synchronization after every time-step. With $R = K$, we can process K time-steps in one execution of the kernel function, but there is significant duplicate calculation among threads. Figure 4 illustrates two executions of the kernel function with $R = 1$ and $K = 2$.

A critical implementation detail that must be noted, is the use of two global buffers. We use one buffer as input and the other as output in alternating fashion to prevent race condition. Note that there is no obvious way to synchronize different blocks in a GPU. The two global buffers serve as synchronization points.

3.2 Lattice Method 2: Triangle

A drawback of the intuitive approach is the large duplicate calculation to reduce the frequency of synchronization. The waste of computational effort eventually affects the performance of the parallel algorithm. In this section we discuss the improved version of the algorithm with a more complex implementation.

As discussed in previous section, given a group of $K + 1$ lattice points of option values, we can only obtain a triangle tree of lattice points without additional information. The tree would span K time-steps with each preceding time-step having one less lattice points than the previous [3]. This triangle of intermediate option values is illustrated in Figure 5. To compensate for the missing entries, we will perform another parallel step which will be explained in more details.

Specifically, we propose a kernel function, "triangle-up", that performs the calculations as shown in Figure 5. Then, it is clear that given a pricing request of MK steps and $MK+1$ initial lattice points, we can run M instances of the function in parallel with overlapping first and last nodes to process the $MK + 1$ lattice points.

After the execution of "triangle-up", more than half of the option values between time-steps $M(K-1)$ and MK would

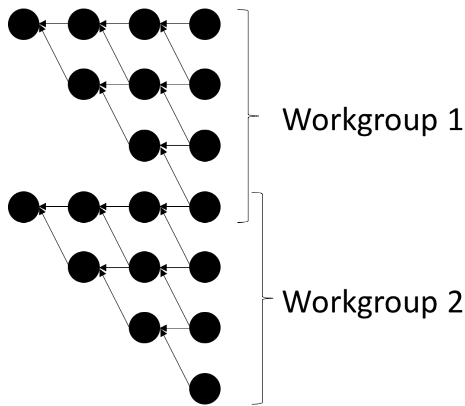

Figure 5: Illustration of triangle tree of lattice points obtained by "triangle-up"

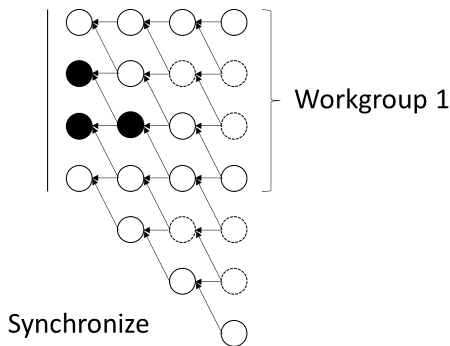

Figure 6: Illustration of reverse triangle tree of lattice points obtained by "triangle-down"

be calculated. As shown in Figure 6, $M-1$ reverse triangle trees of lattice points remain unknown. If all the boundary nodes of the calculated lattice points are stored, we observe that we can use $M-1$ instances of another kernel function, "triangle-down", to fill in the remaining values (black dots in Figure 6). The data flow of the "triangle-down" calculation is indicated by the arrows.

After the execution of and "triangle-up" and "triangle-down", we have full information between time-steps $M(K-1)$ and MK. Effectively, then, we have processed K time-steps and decreased the number of work-items by K. To process the entire option values tree, we just need to iteratively run the two functions M times.

To implement the function "triangle-up", we use a kernel function that process work-groups of $K+1$ work items which correspond to the $K+1$ lattice points. In addition to the main global buffer, we also need a local buffer of $K+1$ for each work-group to store intermediate option values, and another global buffer to store additional boundary node values. To calculate the triangle tree of lattice points, we iterate through K time-steps while synchronizing after each time-step to prevent race conditions and storing intermediate values in local buffer and boundary values in global buffers.

Similarly, we use kernel function that process work-groups of $K+1$ work items to implement the function "triangle-down". To calculate preceding option values, we synchronize each time-step and select the correct lattice points from either the local or global buffers depending on the index and time-step. In the end, we store all final option values back into the original global buffer.

3.3 Monte Carlo Method

The Monte Carlo method is comparatively more straightforward to implement, since the simulations of price movements are independent of each other. We can simply use one work-item per simulation and perform all simulations concurrently.

The main challenge of the Monte Carlo method is generating normally distributed random numbers. When implementing for CPU execution, we can generate normally distributed random numbers with the polar form of Box-Muller transform on uniformly distributed random numbers. With the CUDA framework, we can generate normally distributed random numbers with the CURAND library.

In the parallel implementation, a different random number generator is required for each simulation [4]. To ensure the results are statistically uncorrelated, the generators should have the same seed but different sequence for each simulation. These random numbers are used to calculate the asset price at each time step. The asset price at the final time-step is used to find the simulated option payoff, and then the present value of the average is the final option value.

An alternate way of simulating randomness is with the quasi-Monte Carlo method which uses low discrepancy sequences [6]. This involves creating sub-random numbers that cover the area of interest evenly. Due to the deterministic generation in quasi-Monte Carlo it exhibits a faster convergence than traditional Monte Carlo which has probabilistic convergence.

4. RESULT

In this section, we show the benchmarking results comparing efficiency and accuracy of the parallel implementations to that of the standard implementations. The parameters used for each benchmark are shown in Table 1. For the GPU implementations, we used CUDA version 7.0 and OpenCL 1.2. The GPU programs were run on Nvidia GeForce GTX 980 with 4GB memory. The CPU results were obtained from a separate implementation using C. The CPU programs were run on Intel Dual Core Xeon processors, 3.4GHz, with 16GB memory.

Parameter	Lattice 1	Lattice 2	Monte Carlo
Volatility	30%	30%	20%
Initial Price	$100	$100	$100
Strike Price	$100	$100	$105
Risk-Free Rate	2.00%	2.00%	5.00%
Expiry	1.00	1.00	0.50
Type	Put	Call	Call
Black Scholes Price	10.841	12.822	$4.582

Table 1: Testing parameters

4.1 Lattice Method 1: Group

Table 2 and 3 show that the GPU implementation is able to obtain correct option values of up to 10 significant digits and achieve much better runtime at large time-steps. For

number of time-steps less than 10000, the GPU implementation has inferior performance, most likely as a result of the overhead of initializing the GPU kernel functions and transferring buffers.

Time-steps	Runtime (ms)	
	CPU	GPU
10	1	1115
100	3	1117
1000	15	1118
10000	1700	1284
100000	220734	3419

Table 2: Lattice Method 1: European option performance

Time-steps	Option Values	
	CPU	GPU
10	10.54983349	10.54983349
100	10.81191051	10.81191051
1000	10.83849153	10.83849153
10000	10.84115297	10.84115297
100000	10.84141915	10.84141915

Table 3: Lattice Method 1: European option results

Table 4 and 5 show that calculating the payoff value at each time-step for pricing of American options significantly worsens the performance of the GPU implementation. At 100000 time-steps, the CPU implementation runtime increased 4 times, where as the GPU implementation runtime increased 8 times.

Time-steps	Runtime (ms)	
	CPU	GPU
10	7	1106
100	4	1133
1000	50	1285
10000	6760	2282
100000	760680	25720

Table 4: Lattice Method 1: American option performance

Time-steps	Option Values	
	CPU	GPU
10	10.81911079	10.81911079
100	10.99376906	10.99376906
1000	11.01131875	11.01131875
10000	11.01305085	11.01305085
100000	11.01322305	11.01322305

Table 5: Lattice Method 1: American option results

Finally, we observe that the grow rate of the CPU implementation's runtime is roughly quadratic with respect to the number of time steps, while the GPU implementation exhibits a linear growth in runtime for the tested range of time steps. It is probably due to the high parallelism of the GPU implementation which delays the quadratic behaviour to kick in.

Time-steps	Time (ms)	
	CPU	GPU
500	2.47908	17.3468
1000	10.3548	10.0334
2000	25.191	11.7359
4000	104.243	18.0077
8000	359.79	30.4959
16000	1431.35	89.5854
32000	5545.46	176.971

Table 6: Lattice Method 2: European option performance

Time-steps	Option Value	
	CPU	GPU
500	12.815668	12.817036
1000	12.818624	12.819850
2000	12.820103	12.821415
4000	12.820842	12.822145
8000	12.821212	12.819236
16000	12.821397	12.779358
32000	12.821489	12.779887

Table 7: Lattice Method 2: European option results

4.2 Lattice Method 2: Triangle

By comparing Table 2 and Table 6, we can see that the second GPU implementation outperforms the first as expected. It is noted that single precision is used for the GPU implementation for this lattice method. Plus the effect of roundoff, there is a discrepancy between the option values.

4.3 Monte Carlo Method

To benchmark the GPU implementation of Monte Carlo method, we use the generalized version of the algorithm with jump diffusion, but set the jump probability to zero to show convergence to the Black-Scholes price.

In Table 8, we see that the GPU implementation begins to outperform the CPU implementation when the number of simulations is greater than 100,000. Also, we observe that the GPU runtimes are unaffected until more than 1,000,000 simulations or 50,000 time-steps is used. This implies that at lower numbers of simulations and time-steps, the computing capacity of the individual threads is not yet saturated.

By comparing the generated option values in Table 9 against the Black-Scholes price in Table 1, we see that convergence occurs by increasing either the number of simulations or the number of time-steps. At 1,000,000 simulations and 100,000 time-steps, we achieve convergence of approximately 4 significant digits. We note that the option values given by the CPU and GPU code are not the same since the random numbers used were not the same and there is no obvious way to generate identical random numbers using CPU and GPU.

5. CONCLUSION

In this paper, we have implemented two standard option pricing methods, namely, lattice and Monte-Carlo, using GPU. We have coded the GPU implementations using CUDA and OpenCL. The former is specifically designed for Nvidia graphics cards and the latter is portable for different GPUs as well as CPUs. For the lattice method, we

Simulations	Time-steps	Time (s)	
		CPU	GPU
10,000	50	0.096	1.403
10,000	100	0.189	1.441
10,000	500	0.939	1.42
10,000	1,000	1.863	1.1
10,000	5,000	9.257	1.462
100,000	100	1.957	1.442
100,000	1,000	18.644	1.504
100,000	5,000	95.216	1.877
100,000	10,000	186.085	2.367
100,000	50,000	981.741	5.938
100,000	100,000	1963.448	10.359
1,000,000	100	19.523	6.329
1,000,000	1,000	187.383	6.547
1,000,000	5,000	955.878	7.597
1,000,000	10,000	2056.048	10.375
1,000,000	50,000	9763.871	44.168
1,000,000	100,000	19635.623	86.821

Table 8: Monte Carlo Method: European option performance

Simulations	Time-steps	Option Value	
		CPU	GPU
10,000	50	4.52419	4.5217
10,000	100	4.6434	4.72211
10,000	500	4.5589	4.58213
10,000	1,000	4.55826	4.67678
10,000	5,000	4.42839	4.58776
100,000	100	4.60462	4.55653
100,000	1,000	4.6239	4.55228
100,000	5,000	4.61841	4.54796
100,000	10,000	4.53072	4.59382
100,000	50,000	4.58821	4.55851
100,000	100,000	4.6035	4.60008
1,000,000	100	4.58641	4.57844
1,000,000	1,000	4.57491	4.58701
1,000,000	5,000	4.5791	4.58298
1,000,000	10,000	4.55927	4.59185
1,000,000	50,000	4.56702	4.5782
1,000,000	100,000	4.58469	4.58804

Table 9: Monte Carlo Method: European option results

have proposed and investigated two parallelization of the standard algorithm. The intuitive approach is easy to understand and implement but may not achieve the best parallel performance due to duplicate computation. The improved approach addresses the issue by performing two parallel steps: "triangle-up" and "triangle-down". The resulting method shows better parallel performance. For all the methods, we have compared the CPU runtimes with GPU runtimes. The latter shows significant gain in computational time for achieving similar accuracy.

In this study, the parallelization of computation explored pertains to the valuation of individual options, as opposed to pricing a large number of options. In that scenario, it would seem natural to compute all the options in parallel, with each one computed sequentially by a single thread. However, it is not clear whether it would be more efficient than computing each option in parallel and processing the options one by one. This will be an interesting topic for future study.

6. REFERENCES

[1] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The Journal of Political Economy*, 81(3):637–654, May 1973.

[2] M. Fatica and E. Phillips. Pricing american options with least squares monte carlo on GPUs. In *6th Workshop on High Performance Computational Finance (WHPCF '13) Proceedings*. ACM, Nov. 2013.

[3] N. Ganesan, R. D. Chamberlain, and J. Buhler. Acceleration of binomial options pricing via parallelizing along time-axis on a GPU. In *Proceedings of Symposium on Application Accelerators in High Performance Computing*, July 2009.

[4] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos. Accelerating financial applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 127–136, March 2013.

[5] S. Trainor and D. Crookes. GPU acceleration of a basket option pricing engine. In *World Congress on Engineering 2013 Vol I, WCE 2013 Proceedings*, July 2013.

[6] L. Xu and G. Okten. High performance financial simulation using randomized quasi-monte carlo methods. *Quantitative Finance*, 00(00):1–19, July 2008.