Running Monte Carlo Simulations With Xilinx HLS

Swetha Ananthanarayanan
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
sanantha7@gatech.edu

Brett L. Baxley
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
bbaxley7@gatech.edu

Bowen Zuo
Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
bzuo6@gatech.edu

***Abstract*—Monte Carlo Simulations are used to predict the outcome of a problem with multiple random probabilistic variables. To properly assess the final outcome of a given problem, multiple iterations with vast amounts of data are required. We are proposing a hardware-accelerated solution with a Pynq-Z2 FPGA board, utilizing Vitis HLS pragmas to increase the performance of processing Monte Carlo simulations, while reducing the hardware acceleration turnaround time. The HLS optimized results provided optimizations for random number generation and geometric brownian motion paths separately, but a combined result reduced performance.**

***Index Terms*—Monte Carlo, FPGA Acceleration, Xilinx HLS, Sale Forecasting**

I. Introduction

The Monte Carlo simulation is a process that produces the probability of a given scenario, given random inputs. It uses random sampling to model the behavior of complex systems with a large variable distribution, with highly probabilistic behavior. The main uses for this method involve Risk Analysis, Design Optimization, and Reliability Analysis. For this research, we will utilize the Monte Carlo Method for European Stock Option Price Estimations.

II. Mathematical Models for Stock Price Estimation

When analyzing stock prices, we intend to utilize the Black-Scholes model, which considers the current stock price, the strike price, the time to expiration, the volatility, and the risk-free rate [1]. Volatility is a metric used to calculate the risk of a stock, which measures the dispersion of a return. This is normally done based on standard deviation or variance measurements [2]. The strike price is the price at which the option can be bought by the stock option holder [3]. The risk-free rate is the expected rate of return on an option if there is no risk involved [4]. We can assume that both the risk-free rate and volatility of the options calculated are known and constant [1]. The mathematical notations for a call and put option are denoted as follows:

$$\mathrm{C}_t = S_t N(d_1) - K e^{-rT-t} N(d_2) \qquad (1)$$

$$\mathrm{P}_t = K e^{-rT-t} N(-d_2) - S_t N(-d_1) \qquad (2)$$

In (1) and (2), N(x) is the cumulative distribution function of the standard normal distribution, K is the strike price, T is the time to expiration, t is the time to maturity and r is the risk-free rate [1]. The variables d1 and d2 are defined as the following.

$$\mathrm{d}_1 = \frac{\ln\left(\frac{S_t}{K}\right) + \left(r + \sigma^2\right)(T-t)}{\sigma\sqrt{T-t}} \qquad (3)$$

$$\mathrm{d}_2 = d_1 - \sigma\sqrt{T-t} \qquad (4)$$

This application is utilized to determine whether the stock option should be exercised early or not. At each time step, the value of exercising the option immediately is compared to holding the option until expiration. The decision to exercise the option early depends on the optimal stopping rule that is defined. Once this determination is calculated, the steps are repeated for every time step and every path. Finally, the average value of the options across all paths is obtained to estimate the final price of the American option.

III. Current Technologies And Previous Research

The current technologies utilized in the field of Monte Carlo Simulation involve current HPC architecture, such as utilizing GPU hardware [5] and Cloud Computing resources. Previous research has been conducted to measure performance over specialized hardware accelerators using FPGA's as well. In [6], the FPGA's which were specialized for simulations outperformed General Purpose CPU architectures.

In [5], Cvetanoska et. Al. utilized a high-performance computing algorithm with GPU programming libraries such as CUDA to improve the performance of Quasi-Monte Carlo simulations for American Options, based on backward induction. The techniques used to optimize the performance on the GPU included Reduction Technique, Global Memory Bandwidth, Dynamic Partitions of the Shared Memory Resources, Data Prefetching and Instruction Mix. The results concluded with speed-up times up to 65000 for simulations with one million paths. The strengths of this research involve the drastic improvement in performance, but this is implemented on GPU hardware and not an FPGA.

F. Y. Wang completed a benchmark in [6], displaying how different hardware architectures improved the performance of Monte Carlo Simulations. The author performed benchmarks across an IBM Cell, a Celoxica FPGA, and a ClearSpeed Accelerator. The relative final results concluded that the Celoxica and ClearSpeed implementations provided the best performance speedup. Another relative conclusion is that there is concern with FPGA deployment, involving resource utilizations.

IV. Proposed Solution

Generic hardware and HPC systems are utilized for multiple purposes. The increase in capabilities involves more energy consumption and slower performance for specialized tasks. FPGAs are exceptional for solving complicated mathematical solutions due to the ability for rapid updates available because of the Hardware Description Languages, which allow the hardware to be designed and optimized for specific purposes. Hardware Description Languages such as Verilog or VHDL may improve the turnaround time of producing specialized hardware, but many development challenges still arise with Hardware Description Languages, considering the complex nature of the integrals involved. The turnaround time to update the HDL of a specific function is still more complicated than updating code written in a compiled language such as C or C++. Xilinx's HLS suite provides a solution for this development time constraint, by providing software that allows engineers to develop their system in C++ or Python, and then convert this code into an HDL that can be loaded onto a specific FPGA. We propose to utilize the Pynq-Z2 board, which allows engineers to develop their solutions in Python while accelerating the computations onboard.

V. Implementation

Since we are utilizing a Pynq-Z2 board to implement the Monte Carlo Simulation, we have access to an onboard Processing System (PS) and a Programmable Logic (PL) board as well. Because of the PS, we have an entire Linux Operating System that can utilize Jupyter Notebooks with Python to build out the simulation. We can then optimize the numerical calculations on the PL board utilizing HLS. To achieve the maximum performance, we must first analyze where the simulation can be improved via FPGA implementation.

The bottlenecks in performance for this problem will involve generating the random numerical distribution for each time step and then running each simulation in parallel. The larger the number of samples in the distribution, the longer it will take to develop the final solution. The benefit of a larger distribution is that the result converges to a more accurate solution, which is calculated by taking the average of all the last items in each path. The parallelization of the Monte Carlo method is why it is often chosen to perform larger-scaled simulations. We generate a trivial solution in python to simulate how the geometric brownian motion will display results.

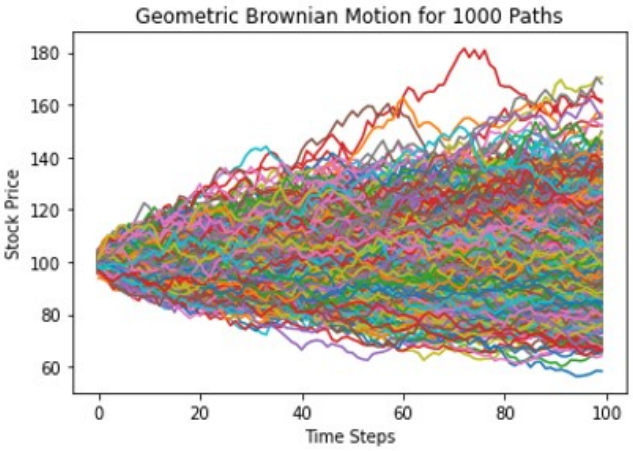

Fig. 1 Random paths taken for stock options using geometric Brownian motion.

Xilinx's HLS suite allows us to optimize code through pragma statements that are associated with specific variables and loops in the code. The simplicity of these implementations allow the user to focus on the logic that the FPGA, must implement on board, instead of creating the HDL output, which improves the development lifecycle for hardware acceleration, while improving the overall usage of the PL.

One solution is to still allow for the random distributions that are used to still be calculated on the PS, and then run each path separately on the PL. Another solution is to both generate the random normal distributions and run all the paths on the PL as well. We will investigate both options to discover which can perform the best optimization, utilizing performance techniques to ensure that we can utilize the limited resources efficiently.

A. Geometric Brownian Motion

Geometric Brownian Motion creates the projected simulated paths for the stock option prices, options, given random events, or "noise", happening in real time. Figure 1 is an example of a Brownian Motion. In GBM, the motion of a particle (or asset price) is influenced by two main components: a deterministic trend and a stochastic (random) component. The deterministic trend represents the general direction of the particle's motion, while the stochastic component represents the random fluctuations. The combination of these two components allows GBM to capture both the systematic and the unpredictable aspects of a phenomenon. The importance of the randomly generated paths is that we can track the volatility of a stock, given random events. To properly generate these events, we require random number generation on the PL, which cannot be properly done in HLS with existing C++ libraries such as rand() or srand().

B. Number Randomization

Generating random numbers in a higher-level language such as C/C++ or Python is as simple as calling a random library to generate the numbers for you. To maintain consistency in

results, a seed value can also be utilized in the function to ensure random results can be replicated. It is not as simple when performing randomization on an FPGA. Normally, a specific block must be added to the synthesis cycle to generate random numbers, or pseudo-random number generation must be done via HLS.

For this project, we are also proposing to implement the Mersenne Twister pseudorandom number generated method proposed by Makoto Matsumoto and Takuji Nishimura. In [7], the authors claim this algorithm provides a 623-dimensional equidistribution, with up to 32 bits of accuracy, while consuming a working area of only 624 words. The method has also been tested as portable C-code, passing statistical tests such as diehard. Using this pseudo-random number generation algorithm gives us the advantage of rapidity and repeatability of the sequences, thus requiring less memory for the algorithm storage. A caveat we may run into is properly improving the process speed of generating seed numbers, since each time step requires information from the previous state. This can be accomplished via pipelining and state management, and will be a task we can improve performance with. Getting random numbers will also require our team to decide how to handle processing mutliple threads that require information from other threads and states as well. We could potentially parallelize the underlying code if we take these items into account, however, we could also make multiple calls to pseudorandom generation function, and parallelize the aqcuisition of random numbers, while leaving the underlying code sequential.

C. Path Generation Performance Improvement.

Another area that performance can be improved is in performing the final call and put simulations for each random path generated separately. This can be accomplished by using HLS memory partitioning and pipelining pragmas to improve performance. Memory access is a major bottleneck when developing hardware solutions. Partitioning memory allows for greater access to memory, and thus reducing computing cycle time to fetch and store memory. Pipelining is a process where multiple instructions utilize the same resources on a processor, but allocate different parts depending on the clock cycle that the instruction is currently being processed.

In addition, for each random path, there is a need to take into account the continuation values in option pricing. This represents the variation in the option price if the user chooses to not exercise it at that particular time step. Hence, polynomial regression is used to implement the same. Considering the fact that regression is heavy on the resources, there is a need to optimize the process on the FPGA.

VI. Continued Work

A. Completed Work

We are currently continuing work to implement Monte Carlo simulations on the Pynq-Z2 board. The current state of our project is still in the research phase and has high-level implementation. We have been researching random number generation for normal distributions, along with implementing black-scholes methods for simulations in Python. To develop a stronger understanding of how to implement the actual algorithm, we utilized a Juypyter Python notebook to implement the simulations, allowing us to develop a stronger intuition on how to disassemble the problem and begin performing HLS optimizations. We will need to unravel each of these computations and perform each computation separately in a looped manner in C++. After developing an intuition from the high-level Jupyter Notebook scripts, we duplicated the trivial solution in C++. This includes the implementation of the random number generator, and standard random normal distribution using the Mersenne Twister PRNG and the Box-Meuler Algorithm for Geometric Brownian Motion Generation of stock prices at varying timesteps. With this work completed, we are about to start the HLS optimization of the codebase.

VII. Midterm Progress

A. Iterative Problem Discovery

Our final step in the process will be to break the problem into two iterations. One iteration will be to use the Pynq-Z2's PL to generate Psuedo-Random numbers, and another involves running the generated paths separately. We will combine both problems and perform benchmark analysis to provide results on how much speed-up is accomplished by utilizing FPGA logic to perform calculations instead of the PS. Because the Pynq-Z2 has limited resources available compared to a standard CPU Architecture, we currently plan to run the number generation separately from the final path calculations. This factor will also force us to discover how to properly allocate resources for handling problems larger than the PL can handle separately, and dynamically being able to account for problems of varying time-steps and simulation sizes. This part of the process relied heavily on researching and developing the baseline code for the current problem.

B. Pseudorandom Number Generation Performance Testing

An implementation of the Mersenne Twister method was referenced and developed locally in C++. The code was run through simulations to ensure accuracy and then applied to multiple threads to discover constraints behind using number generation across multiple threads. The High-Performance Computing library OpenMP was used to easily develop a benchmark test to ensure that running a random number generator across multiple threads would produce viable results. OpenMP was selected due to its simplistic methods to easily improve loop performance by applying multi-threading pragmas. When multi-threading the pseudo-random normal generation code, the test proved to produce random numbers with more latency than running the same code sequentially. This proved that the number generation would not be thread-safe.

VIII. Pseudorandom Number Generation Implementation

The selected random number generator must provide high-quality numbers in order to develop distinctive future routes

and establish the optimum option pricing. This led us to choose the Mersenne Twister pseudo-random generator (PRNG) for our application because it has undergone extensive testing and has successfully passed a significant number of statistical tests pertaining to Monte Carlo simulations. An equally distributed uniform pseudo-random number generator is the Mersenne Twister. As the PRNG is innately slow, we sought to test its implementation on the FPGA to guage its acceleration capabilities. We have created our C++ implementation of Mersenne Twister based on Dr Marsumoto's paper [7]. Our version of Mersenne Twister is used to generate pseudo-random numbers to be used in the implementation of Geometric Brownian Motion. A seed number is used to ensure consistency in the generation of random numbers. This is similar to the functioning of the standard rand function from the random library in python. This doesn't indicate the generated numbers are the same, but it means that we can generate different groups of random numbers by feeding in different seeds for our Mersenne Twister backbone code. The uniformly distributed random numbers that are generated are normalised using the box-mueller transform. We can also generate N numbers as we wanted on C++ backbone. Based on the backbone, we have also built a top-level method for Vitis to run. Since the Vitis implementation requires a fixed-size structure to allocate memory during the synthesis step, one of the parameters of the top-level method's N number is set to be 1000 and is stored in an array of the double type. The other parameter is an int array of size 1 that stores the seed number for the backbone, so we can vary our generated random numbers. This is also more convenient for comparing results between the C++ outputs and the synthesis outputs. During the testing phase, the FPGA outputs were validated against the outputs acquired from the Jupyter notebook.

IX. Geometric Brownian Motion Implementation

A. On-Board Algorithm Implementation

When calculating a single path in a Geometric Brownian Motion, a recursive mathematical computation is applied to sequential time steps in iterations. Referencing the mathematical model from [6], the following equation was applied to each time step.

$$S(t) = S(t\text{-}1) * e^{(\mu - \frac{\sigma^2}{2}) + \sigma\sqrt{t-(t-1)}} \qquad (5)$$

B. Baseline Testing

To ensure the Synthesized code was properly designed to output the same numbers, a random normal distribution was generated and saved for 100 paths and 1,000 paths. The code was then loaded into the python implementation and the designed C++ code to compare results. This was not reproduced for 10,000 paths, since the test results were already completed for 100 and 1,000 paths.

C. HLS Optimization Applications

When these test results passed, HLS pragmas and design changes were applied to the code. The first optimization that was to create a "buffer" array. This array allows for a certain number of paths to be loaded in from Dynamic Ram into Block RAM, which is local to the PL block. Items loaded into B-RAM can be operated on faster due to its location with respect to the PL. For 100 paths, all 100 were loaded in, and operated on. For 1,000, and 10,000 paths, 500 paths were loaded into BRAM and operated on concurrently. 500 paths were chosen due to BRAM capacity limits on the board, while also ensure that the amount of paths loaded into the on-board buffer was easily divisible over the number of paths being generated.

The second design process involved operating on each buffer block. Each path that was generated in this block were run with an HLS pipeline pragma, which allows for each timestep to be operated on in a pipelined architecture, similar to what is used in MIPS and RISC-V Processors. Pipelining allows for a second operation to start after the previous operation is further along in the processing cycle. To optimize this process, array partitioning was also applied to the buffer array, allowing for the buffer to be accessed by multiple processes at the same time. This reduces data transmission constraints on the mathematical computations. The final process in the path motions is to write the computed paths back to off-board memory, or the Dynamic RAM. After the operated path is written back to memory, the next group of paths is loaded from memory, and the process starts over.

Other optimizations were attempted as well. One method that was attempted to speedup the entire process was to utilize ping-pong buffers to remove the dependancy of the recursive algorithm. This method did not provide speedup. Loop reordering for pipelining is another method that is normally used, but due to the recursive nature of the function, the only embarrassingly parallel portion of this process is the different paths that are being operated on.

D. Benchmark Method

To properly benchmark the results, the C++ HLS implementation of the Geometric Brownian motion was synthesized and used to generate a bitstream for the Xilinx Pynq's PL. 3 different bitstreams were generated for running 100 paths, 1,000 paths, and 10,000 paths at a time onboard. The bitstreams were then uploaded to a Jupyter Notebook on the Pynq's PS. A random normal distribution was generated and loaded onto the PS, and then the calculated results were offloaded from the PS. The computation time to generate the Geometric Brownian motion paths were recorded. The same random normal distribution was then run through our implemented python code equivalent of the same geometric brownian estimation algorithm. The benchmark times were recorded and the results were compared.

Each bitstream was run through different path generation tests, to analyze how different bursts compared to path generation for path numbers greater than the maximum paths onboard. For example, the 100-path bitstream was tested with

1,000, 10,000, and 100,000 paths as well, but it was run multiple times. Both the python-generated and HLS-generated paths were averaged to obtain a final option price. The generated option prices were compared after each simulation to ensure the results were accurate.

The Mersenne Twister part is benchmarked with the C++ result and a Python implementation that generates 1000 random numbers that fall within a normal distribution. The accuracy is compared between the FPGA result and the C++ result. However, the Python result is not a replica of the C++ code because we do not create the exact same Mersenne Twister method in Python. So, we just used the Random library method as a comparison. The comparison is conducted under seed number 4357 with N = 1000. The C++ implementation only takes around 0.8 ms to execute. The FPGA synthesis takes about 2.5 ms, while the Python takes around 8.5 ms. The speed-up is significant compared with Python results. If we are concerned about adding the seed implementation (our own seeding method) in Python, the result could be more significant.

X. Combining Random Number Generation and Geometric Brownian Motion Algorithms

A. Separate Bit Streams

A bitstream for random number generation and a bit stream for generating 100 geometric brownian paths were both utilized in a Jupyter notebook. First, the pseudorandom number generation bit was loaded into the PL onboard. The proper memory allocations for the bit stream were then applied, so that the random generated numbers can be read into the Jupyter Notebook. The numbers were calculated and read back from memory. Next, the pseudorandom number generation bit stream was offloaded from the PL overlay, and replaced with a geometric brownian motion bit stream. Like the previous bit stream, the proper memory allocations were also applied and processed. The resulting generated paths were then read back into memory. The performance of this method was compared to an equivalent python implementation, which resulted in a 9x slow-down compared to the python implementation. Combined bitstreams were attempted, but were unsuccessful to synthesise and implement with our current codebase. When benchmarking the results, a complete time capture from start to finish resulted in a 5.25 second latency. Just benchmarking the PS operations resulted in a benchmark of a 52.73 ms latency. The bencharked latency from the trivial python solution was 167.01 ms. Results are as follows in (2).

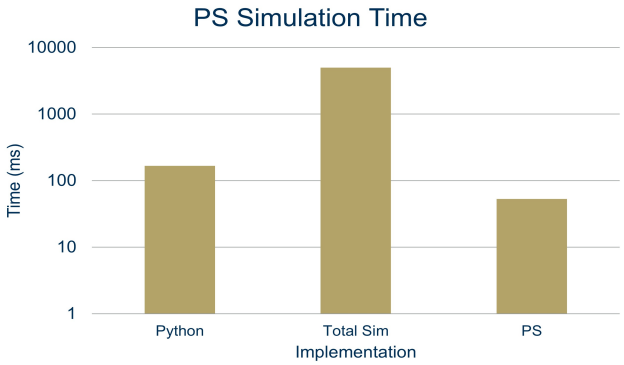

Fig. 2 Benchmark Results for Random Number and Path Generation

XI. Resource Utilization

A. Geometric Brownian Paths

The most utilized part of the Pynq's z7020 board was the BRAM usage, which went up to a 102 percent utilization in the synthesis reports. This was accounted for when creating the actual onboard implementation, with a resource usage of 100 percent. The draw back was that the implementation time may have developed a slowdown due to to meeting physical constraints.

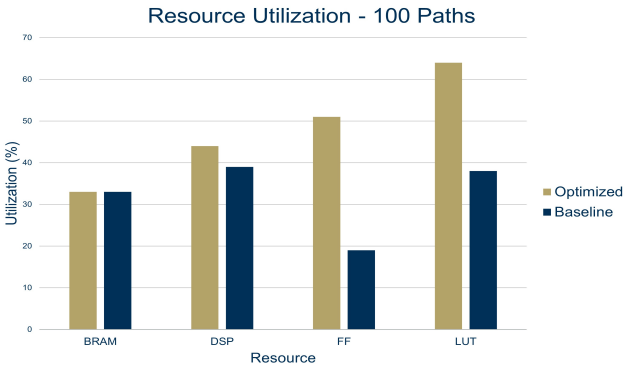

Fig. 3 Estimated Utilization for 100 Paths Bitstream

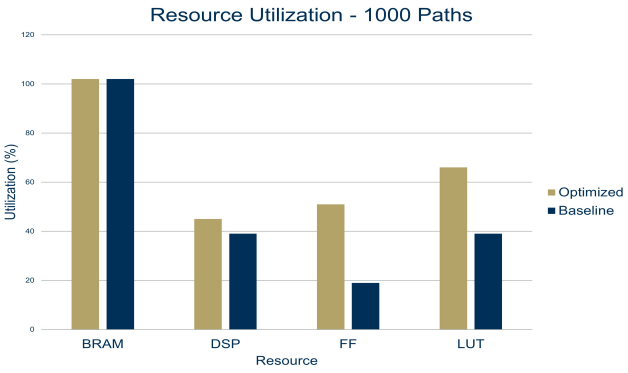

Fig. 4 Estimated Utilization for 1,000 Paths Bitstream

This metric was constant across 1,000 and 10,000 paths, due to the buffer size remaining consistent across both implemen-

tations. In addition, there is an increased usage of LUT and FF when compared to the baseline, which can be attributed to increased resource requirement for complex computations being pipe-lined across successive iterations.

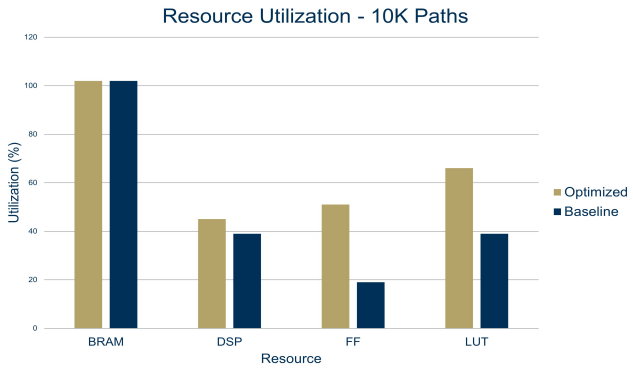

Fig. 5 Estimated Utilization for 10,000 Paths Bitstream

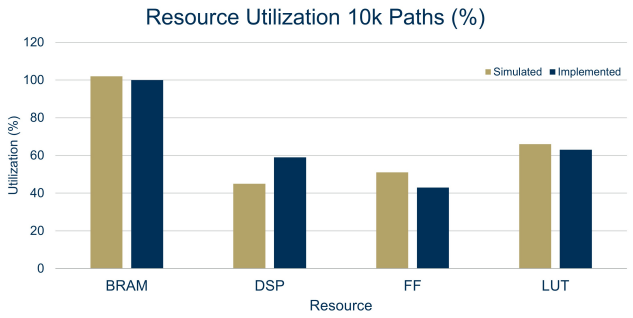

Fig. 6 Estimated vs. Actual Resource Utilization for 10,000 Paths Bitstream

XII. Benchmark Results and Analysis

A. Geometric Brownian Paths

The results for generating the Geometric Paths are shown below. The results show that the bit streams are optimized best for the specific paths they are designed to run for. As the number of paths being generated become greater than the amount the bit stream is designed for, a diminishing return on speedup is observed. The Results also imply that running 10,000 instead of 100,000 runs did not create much of a speedup. The bottleneck for this process may be accounted by the amount of paths that are loaded into the Block RAM did not change, therefore not producing much speedup. Comparing the 1,000 and the 10,000 path bitstreams is a comparison to how much memory can be referenced from DRAM, meaning less bursts are used to generate all the paths.

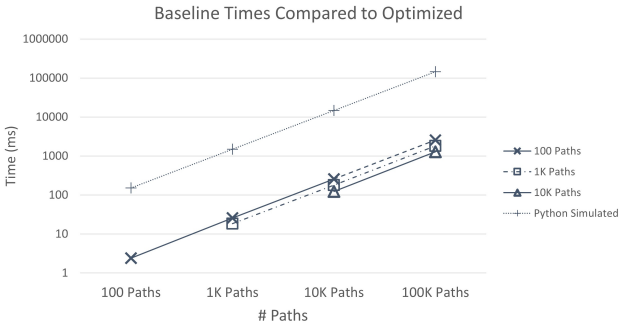

Fig. 7 Logarithmic scale comparing baseline times to HLS Optimized Times.

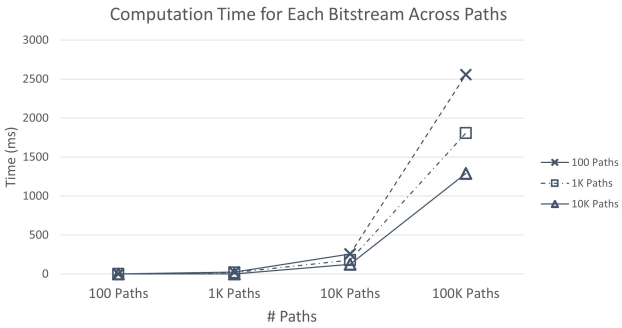

Fig. 8 Computation Times for Optimized HLS Bitstreams with respect to the number of paths generated.

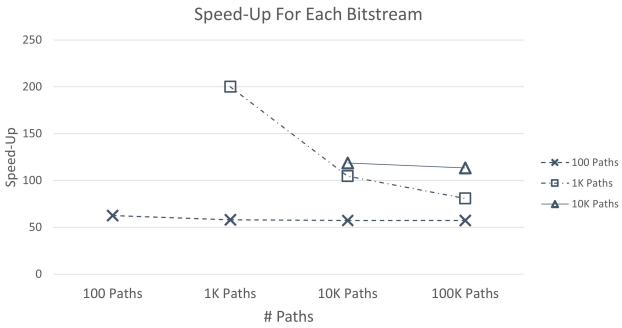

Fig. 9 Speed-Up of Optimized HLS Implementations.

XIII. Conclusion and Continuing Research

A. Results Discussion

Although combining both solutions to perform a single Stock Price Estimated Simulation resulted in a slowdown, both parts separately proposed and implemented displayed performance improvements in their own focuses. Attributions to the combined slowdown may be attributed to the inability to multi-thread random number generation, and how this method was integrated into the geometric Brownian motion implementation.

B. Continuing Research

Continued research will include finding ways to optimize pseudo random normal distribution generation, along with optimizing geometric Brownian motion paths that utilizes random number generation instead of relying on previously generated numbers. Linear Regression will be investigated in future work, as it can provide a better estimate of the optimal option price as opposed to the current implementation of averaging the results from the acquired path. Since the python code that was referenced was a trivial solution, future work will include generating optimized python code to compare results.

REFERENCES

[1] A. Hayes, *Black-scholes model: What it is, how it works, options formula*, Investopedia, 10-Jan-2023. [Online]. Available: https://www.investopedia.com/terms/b/blackscholes.asp. [Accessed: 11-Mar-2023].

[2] A. Hayes, *Volatility: Meaning in finance and how it works with stocks*, Investopedia, 12-Jan-2023. [Online]. Available: https://www.investopedia.com/terms/v/volatility.asp. [Accessed: 11-Mar-2023].

[3] J. Fernando, *Options strike prices: How it works, definition, and example*, Investopedia, 25-Jan-2023. [Online]. Available: https://www.investopedia.com/terms/s/strikeprice.asp. [Accessed: 11-Mar-2023].

[4] A. Hayes, *What is the risk-free rate of return, and does it really exist?*, Investopedia, 19-Jan-2023. [Online]. Available: https://www.investopedia.com/terms/r/risk-freerate.asp. [Accessed: 11-Mar-2023].

[5] V. Cvetanoska and T. Stojanovski, *Using high performance computing and Monte Carlo simulation for pricing american options*, arXiv:1205.0106 [cs], May, 2012.

[6] F. Y. Wang, *Distributed Monte Carlo Simulation for Option Pricing: The first completed benchmark and applications of distributed Monte Carlo simulation model on high-performance computing architecture*, 2009 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009.

[7] Matsumoto, M., & Nishimura, T. (1998). *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*. ACM Transactions on Modeling and Computer Simulation, 8(1), 3-30.