The University of Queensland

COMP7500: High-Performance Computing

Semester 2 2021

European Basket Options

Project Report

Compiled and Authored by

Joel Thomas

44793203

Table of Contents

Chapter 1
Serial Implementation

1.1 Introduction

This project report serves to investigate high-performance computing techniques applied to the mathematical finance problem of European Basket Options pricing. Initially, an introduction to the context and terminology associated with the problem is provided. This is followed by a gentle overview of the underlying sophisticated mathematics that governs the underlying problem. Given that there is no closed-form solution, Monte Carlo simulation together with the Euler method is introduced as a simple technique to achieve a numerical solution to an otherwise extremely challenging task. An algorithm for the serial implementation written in pseudocode is provided to the reader before translating this algorithm into the popular C++ programming language and conducting manual optimisations and profiling. Next, the runtime test results for problem scaling with increasing input size and differing compiler optimisation flags are showcased. This is to highlight the performance scaling ability of the serial implementation to the problem.

1.2 Background Theory

1.2.1 European Options Pricing

To motivate the discussion of European options, an example scenario is given. Consider the case of farmers who grow agricultural commodities such as wheat or corn for example. Since it is difficult for them to predict destructive environmental conditions such as droughts or floods over the long run, they may wish to protect themselves financially against downside price movements in financial markets where their produce (various commodities) is actively traded (Doherty, 2020). If they anticipate adverse events like these and they have the choice of locking in into a financial contract where their selling price to buyers is fixed rather than variable, they would clearly be much better off. This is even better if the downside loss is limited to only the premium they pay to enter into such a contract, in case otherwise favourable market conditions. Note that options contracts are not limited to farmers alone. The biggest entities in options markets consist of retail investors, institutional traders, broker-dealers and market-makers, all involved in either speculation or hedging (like the farmers).

An options contract in essence is a formal financial agreement between two interested parties desiring to facilitate a potential transaction on an underlying asset at a fixed predefined price prior to or on the contract expiration date (Hargrave, 2003). In the example above, it is clear to see why these are so popular – they offer utility, flexibility, a cost-efficient solution to risk management and open a doorway to several strategic alternatives when bought and sold combinatorically (Ianieri, 2006).

A European option simply limits execution of the transaction to only on the date of contract expiry, as opposed to an American option which allows for pre-expiry execution as well. The key parameters here are:

- S_t which represents the underlying asset's price at time t.
- K which is the contract strike price.
- $t \in [0, T]$ which represents time, T being the contract expiration date.

Focusing on European options only, it is now important to differentiate between European call and put options contracts. European calls represent a financial agreement where:

- The contract holder has the right, but not the obligation, at time T to buy from the contract writer the underlying asset for fixed price K.
- The contract writer is obliged to sell at time T, should the contract buyer decide to buy.
- The payoff at maturity is given by $C_T = \max\{S_T - K, 0\} = (S_T - K)^+$.
    - Higher payout (profit) as the underlying asset's price appreciates.

Compared to European puts:

- The contract holder has the right, but not the obligation, at time T to sell to the contract writer the underlying asset for fixed price K.
- The contract writer is obliged to buy at time T, should the contract buyer decide to sell.
- The payoff at maturity is given by $C_T = \max\{K - S_T, 0\} = (K - S_T)^+$.
    - Higher payout (profit) as the underlying asset's price depreciates.

The goal of options pricing in general is to find an analytic or numerical solution to C_0 given the payoff function C_T at time T and sufficient information about the underlying asset's dynamics and associated parameters. This is because if one can model S_t for $t \in (0, T]$, calculating C_0 only requires calculating C_T and then appropriately discounting the price back to time $t = 0$ due to the time value of money (Chen, 2007). Furthermore, having a way to calculate C_0 ensures that the option premium the holder pays (writer receives) to enter into the contract is fair and not under or overvalued. The latter case opens up financial arbitrage opportunities which are considered severe market inefficiencies according to the Efficient Market Hypothesis (Downey, 2003).

The most popular model and industry standard by far for calculating plain vanilla European option prices is the Black-Scholes-Merton (BSM) model initially published by world renowned economists Fisher Black and Myron Scholes in 1973 (Black et al., 1973). There are two general approaches available that yield the same end solution as shown in Figure 1 below, but only the risk-neutral pricing method (right) is used in this project. This is because European Basket Options, as explained in the section below, can be represented by a high-dimensional PDE but this suffers immensely from the curse of dimensionality. On the contrary, Monte Carlo simulation is a much simpler and faster computational approach that can handle multi-dimensionality and path dependency very well.

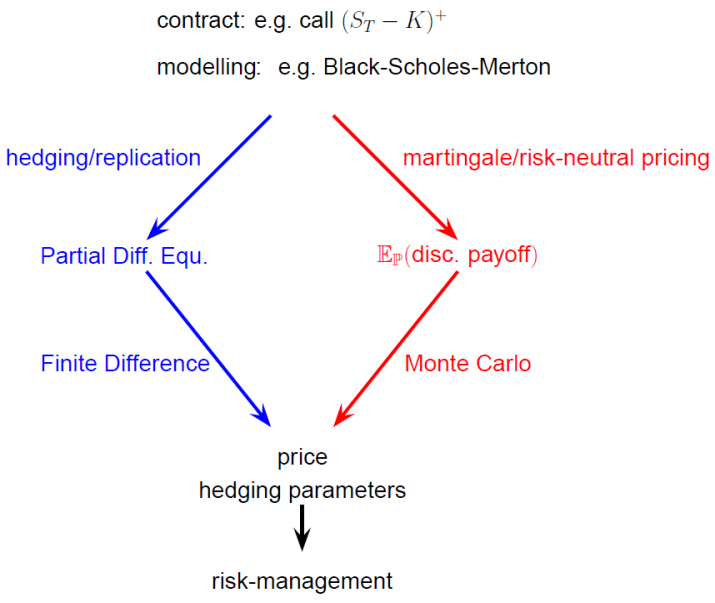

Figure 1: Main approaches used for pricing options contracts.

1.2.2 European Basket Options Pricing

Exotic options are special contracts that are different to plain vanilla options in terms of their payoff structures, maturity dates or strike prices and exist solely to provide higher flexibility and customisability to interested parties (Chen, 2003). European basket options different to vanilla European options in that they are simply priced on multiple underlying assets instead of just one. They offer a more cost-effective method instead of paying commissions and transaction fees for multiple individual contracts but at the cost of reduced contract profit volatility affected by individual underlying price jumps (Mitchell, 2003).

For example, one of the simplest payoff functions for a European **call** option written on a basket of D underlying assets is $C_T = \left(\frac{1}{D}\sum_{d=1}^{D} S_T^{(d)} - K\right)^+$. The superscript d here is simply an indexing variable and does not imply exponentiation. Whilst the weights on the individual assets need not be equal, the expression $\frac{1}{D}\sum_{d=1}^{D} S_T^{(d)}$ represents the average maturity price of an equally weighted basket of D assets. Equal weights shall be used for the remainder of the project due to simplicity.

1.2.3 Underlying Mathematics

It is extremely challenging to describe the underlying mathematics involved without delivering a full course in Probabilistic Measure Theory and Financial Calculus. However, an attempt has been made to provide a gentle yet sufficient introduction. Some important terminology must be initially explained:

- A stochastic process is a mathematical concept that describes the evolution in time $t \in [0, T]$ of something that behaves randomly.
    - Here, $\left\{S_t^{(d)}\right\}_{t\in[0,T]}, 1 \le d \le D$ are valid continuous-time stochastic processes.
- A Brownian Motion $\{W_t\}_{t\in[0,T]}$ is a continuous-time stochastic processes where the initial value $W_0 = 0$, $\{W_t\}_{t\in[0,T]}$ has independent, Gaussian (normally distributed) increments and is continuous in t.
- A Geometric Brownian Motion (GBM) is similar to a BM except the stochastic process can never go below the value 0 (lower bound). This is extremely useful for modelling price dynamics of an asset where the price cannot go below 0 as it is in real life.
- A stochastic differential equation (SDE) is similar to an ordinary differential equation except, one or more of the terms involved are stochastic processes. As a result, the solution is also a stochastic process.

As explained above, an estimate for C_T can be obtained if each of the underlying assets' price dynamics is modelled appropriately. The frictionless markets assumption is made here to induce a simpler setting – no taxes, dividends, commissions, transaction costs or market impact, fractional trading allowed and infinite liquidity available. It is further assumed that each asset in the basket is a continuous-time stochastic process that follows a Geometric Brownian Motion (GBM) such that each $S_t^d \in [0, \infty)$. The stochastic differential equation (SDE) each asset represents is given by:

$$dS_t^{(d)} = \mu_d\left(S_t^{(d)}, t\right) S_t^{(d)} dt + \sigma_d\left(S_t^{(d)}, t\right) S_t^{(d)} dW_t^{(d)}, \qquad 1 \le d \le D, \qquad t \in [0, T]$$

For simplicity, it is assumed that the functions $\mu_d\left(S_t^{(d)}, t\right)$ and $\sigma_d\left(S_t^{(d)}, t\right)$ are:

- The same for each of the assets i.e. $\mu_d\left(S_t^{(d)}, t\right) = \mu(S_t, t)$ and $\sigma_d\left(S_t^{(d)}, t\right) = \sigma(S_t, t)$
- Constant and non-random through time i.e. $\mu(S_t, t) = \mu$ and $\sigma(S_t, t) = \sigma$

The SDE then simplifies to $dS_t^{(d)} = \mu S_t^{(d)} dt + \sigma S_t^{(d)} dW_t^{(d)}$. Explaining each of the terms:

- The SDE itself describes the change in each asset's price brought about in an infinitesimal period of time dt.
- μ is the constant drift of the stochastic process (expected return for asset d).
- σ is the constant variance or diffusion of the stochastic process (volatility for asset d).

- The BMs for different assets can be correlated i.e. $Corr\left(dW_t^{(i)}, dW_t^{(j)}\right) = \rho^{(i,j)}, 1 \leq i,j \leq D$. Therefore, $\rho^{(i,j)} \in [-1,1]$ for $i \neq j$ and $\rho^{(i,j)} = 1$ for $i = j$.
    - Observe that these stay constant through time (time homogeneous) and when grouped together, a correlation matrix $\mathrm{P} \in \mathbb{R}^{D\times D}$ is formed.

Monte Carlo simulation can be easily used for this problem to find the time-0 price C_0 for the options contract. The basic idea here is to generate M simulated trajectories (samples) for each individual $\left\{S_t^{(d)}\right\}_{t\in[0,T]}$ till contract maturity at $t = T$. This is done by dividing the time interval $[0,T]$ up into N sub-intervals where the timestep size $\Delta t = \frac{T}{N}$ is common to all assets in the basket and then using the given price dynamics for each asset to "timestep" one time period $t_n = n\Delta t$ into the future repeatedly. This method is known as the Euler method or Euler Timestepping. This is sensible because as N gets larger, the finer the movements become between time periods. Each time period becomes almost instantaneous, and this resembles well the infinitesimal change described by the SDE.

Next, it is important to account for the fact that since the BMs for each SDE are potentially correlated, a mechanism is needed for generating correlated standard normal random numbers. Given the constant correlation matrix P of correlations between the BMs, it is necessary to identify a matrix $L \in \mathbb{R}^{D\times D}$ such that $LL^T = P$. This is obtained via performing Cholesky decomposition. If $Z_n \in \mathbb{R}^{D\times M}$ represents a matrix of standard normal random numbers (each element is $\sim N(0,1)$), one for each sample for each asset at a given time sub-interval t_n, then $LZ_n \in \mathbb{R}^{D\times D}$ is a matrix of correlated standard normal random numbers with correlation $Corr(LZ_n) = LL^T = \mathrm{P}$ as required (proof beyond scope of project).

Finally, Euler timestepping is performed on M samples for each of the D assets for a total of N periods. The higher the number of samples generated, the higher the confidence in the final result. In summary, the desired Euler scheme is given by:

$$\hat{S}_{n+1}{}^{(d,m)} = \hat{S}_n{}^{(d,m)} + \mu\hat{S}_n{}^{(d,m)}\Delta t + \sigma\hat{S}_n{}^{(d,m)}(LZ_n)^{(d,m)}\sqrt{\Delta t}, \qquad 1 \leq d \leq D, \qquad 0 \leq n \leq N-1, \qquad 1 \leq m \leq M$$

Figure 2 below illustrates pictorially a simpler variant of this scheme in the case of a single $D = 1$ underlying asset (tgood, 2017). The messy plot lines display the possible simulated price paths the asset could take using this simpler scheme.

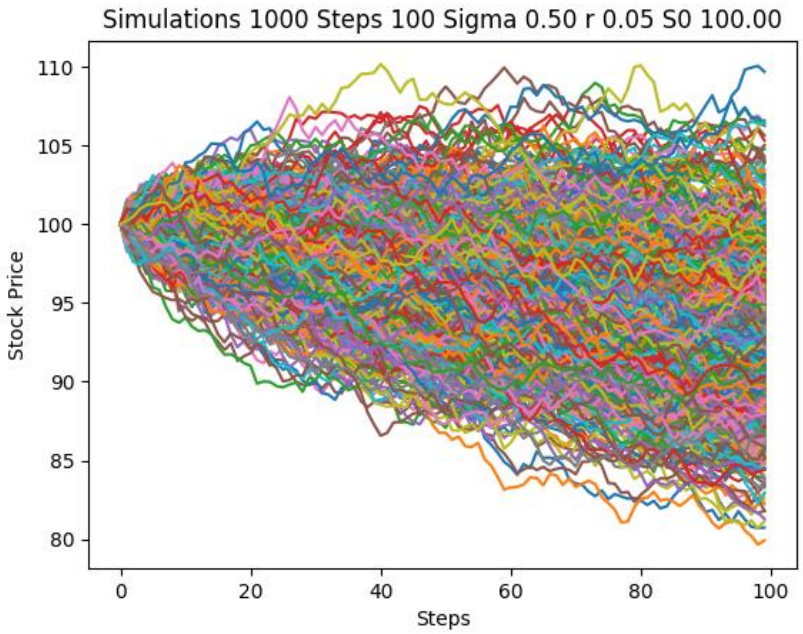

Figure 2: An example illustrating the possible simulations resulting from a simpler Euler scheme. The key relevant parameters here are $D = 1, M = 1000, N = 100$.

1.3 Implementation Details

1.3.1 Algorithm

The pseudocode for the initial serial implementation is provided in Table 1 below.

	Algorithm 1: *Serial Implementation for MC Simulation for European Basket Options Pricing*
1:	**initialise** $D, N, M, (\boldsymbol{S_0}, \boldsymbol{\mu}, \boldsymbol{\sigma}) \in \mathbb{R}^{D\times 1}, K, r, T$;
2:	**initialise** $\hat{S}_0 = [\boldsymbol{S_0}, \dots, \boldsymbol{S_0}] \in \mathbb{R}^{D\times M}, \mathrm{P}$;
3:	$L \leftarrow cholesky\_decomposition(\mathrm{P})$;
4:	$\Delta t \leftarrow \frac{T}{N}$;
5:	**for** $0 \le n \le N-1$ **do**
6:	**declare** $Z_n \in \mathbb{R}^{D\times M}$;
7:	**for** $1 \le d \le D$ **do**
8:	**for** $1 \le m \le M$ **do**
9:	*// Store Z_n in same variable as Z_{n-1}*
10:	$Z_n^{(d,m)} \leftarrow {\sim}^{iid} N(0,1)$;
11:	**end**
12:	**end**
13:	**declare** $LZ_n \in \mathbb{R}^{D\times M}$;
14:	$LZ_n \leftarrow L \times Z_n$;
15:	**for** $1 \le d \le D$ **do**
16:	**for** $1 \le m \le M$ **do**
17:	*// Store $\hat{S}_{n+1}$ in same variable as $\hat{S}_n$*
18:	$\hat{S}_{n+1}{}^{(d,m)} \leftarrow \hat{S}_n{}^{(d,m)} + \mu_d \hat{S}_n{}^{(d,m)} \Delta t + \sigma_d \hat{S}_n{}^{(d,m)} (LZ_n)^{(d,m)} \sqrt{\Delta t}$;
19:	set $\hat{S}_{n+1}{}^{(d,m)} \leftarrow \max\left(\hat{S}_{n+1}{}^{(d,m)}, 0\right)$;
20:	**end**
21:	**end**
22:	**end**
23:	**declare** $\boldsymbol{Y} \in \mathbb{R}^{M\times 1}$;
24:	**for** $1 \le m \le M$ **do**
25:	$Y^{(m)} \leftarrow \exp(-rT) \times \max\left(\frac{1}{D}\sum_{d=1}^{D} \hat{S}_N^{(d,m)} - K, 0\right)$;
26:	**end**
27:	$\hat{C}_M \leftarrow \frac{1}{M}\sum_{m=1}^{M} Y^{(m)}$;
28:	$\hat{\sigma}_M \leftarrow \sqrt{\frac{1}{M-1}\sum_{m=1}^{M}\left(Y^{(m)} - \hat{C}_M\right)^2}$;
29:	$z \leftarrow 1.96$;
30:	$CI\_left \leftarrow \hat{C}_M - z\frac{\hat{\sigma}_M}{\sqrt{M}}$;
31:	$CI\_right \leftarrow \hat{C}_M + z\frac{\hat{\sigma}_M}{\sqrt{M}}$;
32:	$radius \leftarrow z\frac{\hat{\sigma}_M}{\sqrt{M}}$;
33:	**print** $D, N, M, \hat{C}_M, [CI\_left, CI\_right], radius$;

Table 1: Pseudocode for the serial implementation.

After the extensive discussion on underlying mathematics in section 1.2.2 above, Algorithm 1 follows logically. In essence, the algorithm aims to generate M simulations for the price dynamics of each of the D assets by timestepping one time period at a time into the future till $t_N = T = N\Delta t$. This occurs within the main simulation loop from lines 5 to 22 with the Euler scheme appearing on line 18. For simplicity, in the code $\boldsymbol{S_0} = [100, \dots, 100]^T, \boldsymbol{\mu} = [0.02, \dots, 0.02]^T, \boldsymbol{\sigma} = [0.3, \dots, 0.3]^T$ for $1 \le d \le D$ i.e. all assets' samples share the same initial asset price, constant drift and diffusion. $D = 8$ assets, $N = 100$ timesteps and $M = 10000$ samples are the base simulation settings. Furthermore, $K = 100, r = 0.02, T = 1$ and $\mathrm{P} = \mathbb{I}$ (the identity matrix) to represent fully uncorrelated assets.

There are two key subtleties that require notice. Firstly, whilst it would have been more sensible to join both sections of nested loops on lines 7 and 15, it is impossible and needs to be split because of the need to generate correlated standard normal random numbers to account for correlated BMs as explained in section 1.2.2 above. The entire matrix Z_n needs to be generated prior to calculating LZ_n from which the (d,m)-th element is used in line 18 later. Next, whilst a GBM $\{S_t^d\}_{t\in[0,T]}$ can never go negative, in practice, $\hat{S}_{n+1}{}^{(d,m)}$ can become negative if $(LZ_n)^{(d,m)}$ is negative for sufficiently many periods. Hence, $\hat{S}_{n+1}{}^{(d,m)}$ is updated with $\max\left(\hat{S}_{n+1}{}^{(d,m)}, 0\right)$ to reflect the GBM non-negative property.

The code appearing in lines 23 to 32 shall now be explained. As mentioned previously, once the final time $t_N = T$ asset prices have been simulated from t_{N-1}, the contract price at maturity C_T can be calculated for each MC sample. However, it needs to be discounted back to time $t_0 = 0$ to reflect the time value of money property (money is worth more now than in the future). The exponential function is commonly used for continuous-time discounting in financial mathematics. The contract is discounted from time T back to time 0 by the risk-free interest rate r (assumed constant through time) for a total period of length $T - 0 = T$. This occurs from lines 23 to 26 and is stored in $\boldsymbol{Y}$. After generating M samples of the discounted payoff, taking the equally weighted average of these samples provides the final MC estimate for C_0 given by $\hat{C}_M$ (the M subscript no longer represents time). This is performing the $\mathbb{E}_{\mathbb{P}}[disc.\ payoff]$ operation appearing in Figure 1 behind the scenes. The rest of the algorithm from lines 29 to 32 are to create a 95% confidence interval in the estimate $\hat{C}_M$ that contracts as M increases (implying higher confidence). Finally, all results are printed by the compiler on line 33.

1.3.2 Program Design

Algorithm 1 from Table 1 was implemented in C++, more specifically the C++11 standard. The arguments for parameters N and M were chosen to be provided as input through the terminal at runtime to be able to conduct runtime experiments by changing either while keeping the other fixed at some base value. Since the values of N and M are only provided at runtime, extra attention must be brought to the problem of dynamic memory allocation for arrays – this is explained later. However, variable setting at runtime complemented the use of for loops in the Slurm bash script to help test different combinations of N and M more easily. The base values for all variables were discussed in the algorithm section above.

Whilst Algorithm 1 may appear simple, it is quite inefficient due to the need for multiple nested for loops. One efficient solution to this problem is to vectorise as much of the code as possible with a library that can support well large-scale linear algebra. This led to heavy use of the Eigen C++ library which is a high-performance C++ template library for linear algebra (Jacob et al., 2006). It includes classes for arrays and matrices of differing numeric types that support fast vector, matrix and element-wise arithmetic, numerical solvers for several decomposition algorithms as well as other relevant linear-algebra algorithms. Uses of the Eigen library within Algorithm 1 are now explained. Apart from vectorizing several variables, the $llt()$ method was called on P to perform standard Cholesky decomposition to obtain L. Furthermore, the $cwiseProduct()$ and $cwiseMax()$ methods were used to perform element-wise multiplication and maximum respectively when timestepping $\hat{S}_n$ to $\hat{S}_{n+1}$ and when calculating Y. As a final bonus, Eigen also takes care of the memory management issue by featuring automatic dynamic memory allocation during calls to library functions and memory release as variables go out of scope.

Furthermore, whilst the C++11 standard introduces the $< random >$ header for random number generation (RNG) and probability distributions, the Ziggurat algorithm for standard normal RNG is far better optimised and was used heavily instead. This is explained further in the authors' paper and in section 1.2.7 below (Marsaglia et al., 2000). Other headers include the $< ctime >$ header used to set the RNG seed, the $< cmath >$ header used for standard mathematical operations such as $exp()$ and $sqrt()$ and finally the $< chrono >$ header for high precision and resolution runtime recording.

Finally, the use of higher-level features in C++ such as classes and structs in objected oriented programming (OOP) were avoided due to the simple nature of the algorithm and hence the lack of need for such features. Even simple

one-liner subroutines were kept within the main simulation loop rather than defined as separate functions to avoid the expensive time overhead attached to multiple function calls, even if passing arguments by reference parameters. The trade-off here is between runtime and reusability together with readability – the $perform\_mc\_simulation()$ function is slightly long consequently.

1.3.3 Verification Procedure

To verify that the serial implementation is programmed correctly, several steps were undertaken. Firstly, the code was written in MATLAB which has a much simpler syntax and is less prone to suffer from nasty hard-to-trace bugs. Using the same base variable settings as in Algorithm 1, the final $\hat{C}_M$ for both programs are quite close as expected (difference existing solely due to stochastic nature of both programs).

Next, a special case of the European basket options pricing problem was observed and used. This special case refers to the scenario where the BMs' correlation matrix $\mathrm{P} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}$ i.e. a matrix of ones. Intuitively, this means that each underlying asset's price dynamics is perfectly correlated to every other asset's price dynamics. If one of the asset's price appreciates over a given time sub-interval, so will the prices for all other remaining assets. This special case together with the initial assumption of common S_0, μ, σ shared between all assets ensures that each asset is identical to every other asset in the basket and price dynamics are the exact same for every asset (direction as well as magnitude of price changes).

The mathematical reasoning behind this is very intuitive. In generating the correlated standard normal random numbers with the modified P, due to the matrix arithmetic in LZ_n, only the rows (samples) will feature different $\sim N(0,1)$ numbers whereas the columns (assets) will feature the exact same number along each row e.g.

$$LZ_n = \begin{bmatrix} 0.8965 & \cdots & 0.8965 \\ 0.0027 & \cdots & 0.0027 \\ \vdots & \ddots & \vdots \\ -0.2811 & \dots & -0.2811 \end{bmatrix}$$

This explains why every asset's prices appreciates or depreciates the same as every other asset. Now, when the row-wise mean of every sample is taken in $Y^{(m)} = e^{-rT} \max\left(\frac{1}{D}\sum_{d=1}^{D} \hat{S}_N^{(d,m)} - K, 0\right)$, the $\frac{1}{D}\sum_{d=1}^{D} \hat{S}_N^{(d,m)}$ term collapses to $\frac{1}{D}\sum_{d=1}^{D} \hat{S}_N^{(m)} = \frac{1}{D} \times D\hat{S}_N^{(m)} = \hat{S}_N^{(m)}$ since the final asset prices are all equal in a given sample at contract maturity ($t = T$). In essence, this is identical to performing MC sampling to simulate just a single asset's price dynamics. From the end of section 1.2.1, it was briefly mentioned that there is a popular closed-form solution known as the Black-Scholes (B-S) formula for pricing a vanilla European call/put option where the underlying follows a GBM. The details of the formula need not concern the reader as MATLAB's Financial toolbox's $blsprice()$ function takes as input a single S_0, K, μ, T, σ and calculates the true C_0 based on the B-S formula. Using $S_0 = 100, K = 100, \mu = 0.02, T = 1, \sigma = 0.3$ returns $C_0 \approx 12.8216$ which exactly matches the C++ serial implementation. Hence, this special case verification method ensures the code works as intended and provides high confidence for the implementation to work on all other types of valid input e.g. varying $\boldsymbol{S_0}, \boldsymbol{\mu}, \boldsymbol{\sigma}$, using a different correlation matrix P and so on.

1.3.4 Profiling and Optimisations

The biggest improvements to optimising the serial implementation's performance were driven by use of the Eigen library and the Ziggurat algorithm. Whilst a set of benchmark tests weren't conducted for the use of multiple nested loops compared to the Eigen library due to time constraints, it is guaranteed that the Eigen library would always outperform given the number of years its creators and contributors have spent into optimising all its functions and classes. On the contrary, the Ziggurat algorithm was tested against the $default\_random\_engine$ and $normal\_distribution$ class combination for standard normal RNG made available by the $< random >$ header introduced in the C++11 standard. In almost every compiler-based optimisation tested (-$O0$, -$O1$, -$O2$, -$O3$ and -$O3$ -$ffast\text{-}math$), runtime was halved with $D = 8, N = 100, M = 100000$ used as benchmark settings. This approves the decision to switch to a faster, higher quality and scalable RNG algorithm.

Other manual optimisations implemented to improve runtime include:

- Advanced initialisation of the matrix Z_n at each time sub-interval t_n with standard normal random numbers.
    - Replaced lines 7 to 12 in Algorithm 1 with a more efficient one-liner.
- Advanced initialisation of the matrix $\hat{S}_0$ by turning the vector S_0 into a diagonal matrix $\in \mathbb{R}^{D\times D}$ and multiplying $\mathbf{1} \in \mathbb{R}^{M\times D}$ by this new diagonal matrix
- Avoided recalculating constants appearing within loops by moving them outside.
    - $mu\_dt = \mu \times dt,\ sigma\_sqrt\_dt = \sigma \times \sqrt{dt}.$
- Removed unnecessary type conversions from $double$ to $float$.

Profiling was conducted using the $gprof$ utility that uses both instrumentation and sampling. This was done by manually setting the $-pg$ flag combined with the -$O0$ flag explicitly informing the compiler for compiler optimisations to be switched off. Both the flat profile and call graph produced by $gprof$ were very complicated to interpret since almost all of the function names appearing in both were mostly dominated by methods and classes belonging to the Eigen library.

In stating this however, one key observation was made regarding the flat profile. The $r4\_nor()$ and $shr3\_seeded()$ functions in the Ziggurat algorithm represent roughly 0.19 and 0.13 seconds out of a total test runtime of 2.18 seconds. This corresponds to percentages of 5.96% and 4.13% of the total runtime. These functions are important because they were the second and third highest time-consuming functions appearing in the flat profile. This potentially suggests that if the methods available for standard normal RNG from the $< random >$ header (which have been tested to be slower) were to be used instead, these percentages would likely be much higher along with worse runtime. In the flat profile, the largest runtime could potentially be occupied by the RNG-related function calls instead.

1.4 Performance Results

Using the finalised revision of the serial implementation, several benchmark tests were performed to analyse the effect of problem scaling through increasing input size. The $< chrono >$ header mentioned in section 1.3.2 was used for recording high precision runtime in milliseconds (ms). Furthermore, five different settings of compiler optimisations ranging from -$O1$, -$O1$, -$O2$, -$O3$ and -$O3$ -$ffast\text{-}math$ were used to perform these benchmark tests. The results that appear in Figures 3 and 4 below are based on the equally weighted average of 20 runs for a given combination of N and M on each of these compiler optimisations. These testing schemes are further explained below.

For the serial implementation, the problem can be scaled by changing either or all of D, N or M. However, D was kept fixed at the base setting $D = 8$ assets for all benchmarks. The testing scheme used to evaluate performance with varying N was to increase N from $N = 100$ sub-intervals to $N = 1000$ sub-intervals in increments of 25 whilst keep M fixed at the base setting of $M = 10000$ samples. Similarly, to evaluate performance with varying M, the testing scheme used was to increase M from $M = 10000$ samples to $M = 100000$ samples in increments of 2500 whilst keeping N fixed at the base setting of $N = 100$ sub-intervals. Changing both N and M could have been done as a third testing scheme to evaluate the performance effects of varying both but this wasn't able to be done given the time constraints. Unlike Figure 3, it is anticipated that such a scheme would likely result in quadratic curves for each of the compiler optimisations since increasing one whilst holding the other produces linear curves. However, further testing is required to prove this hypothesis.

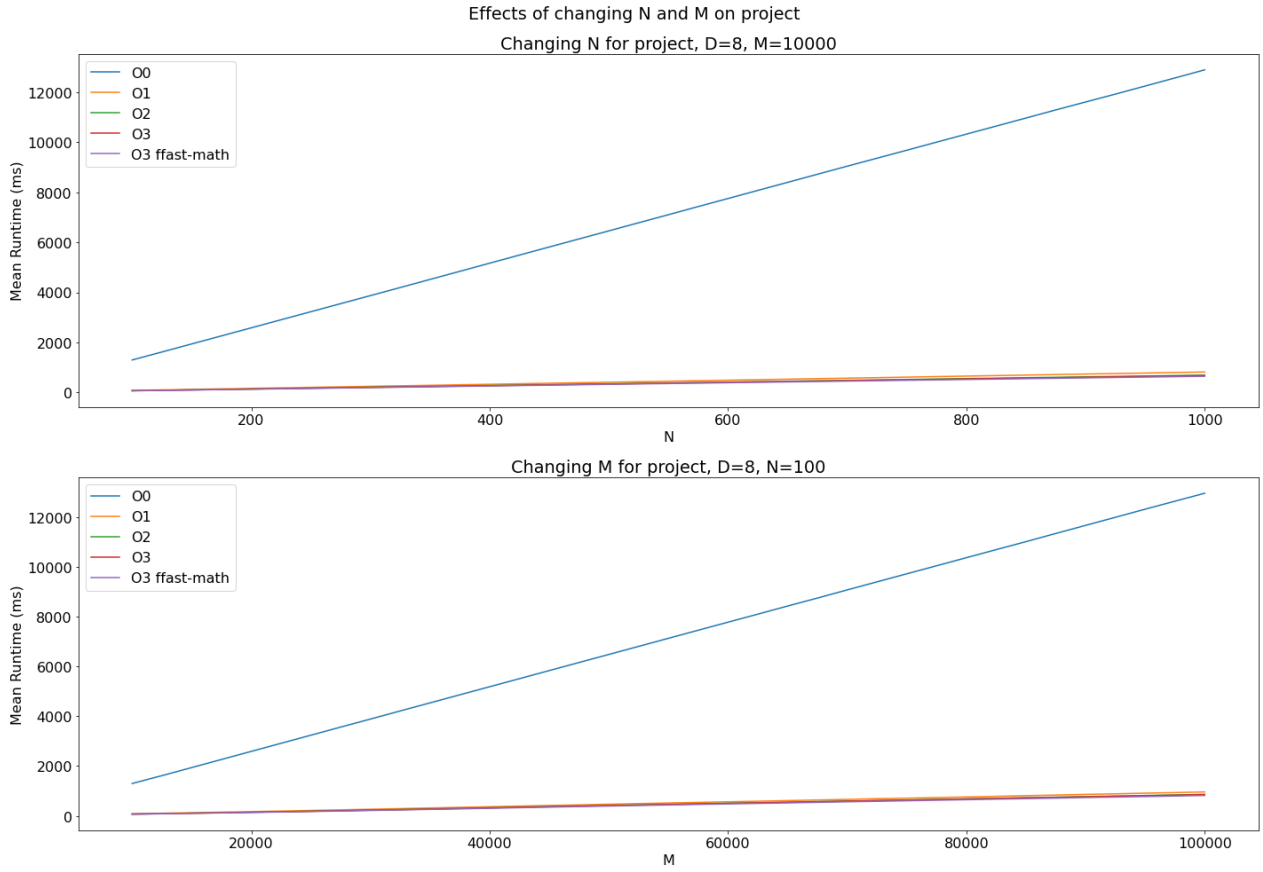

Figure 3: Performance scaling results with $-O0$ included and no scale used on y-axis.

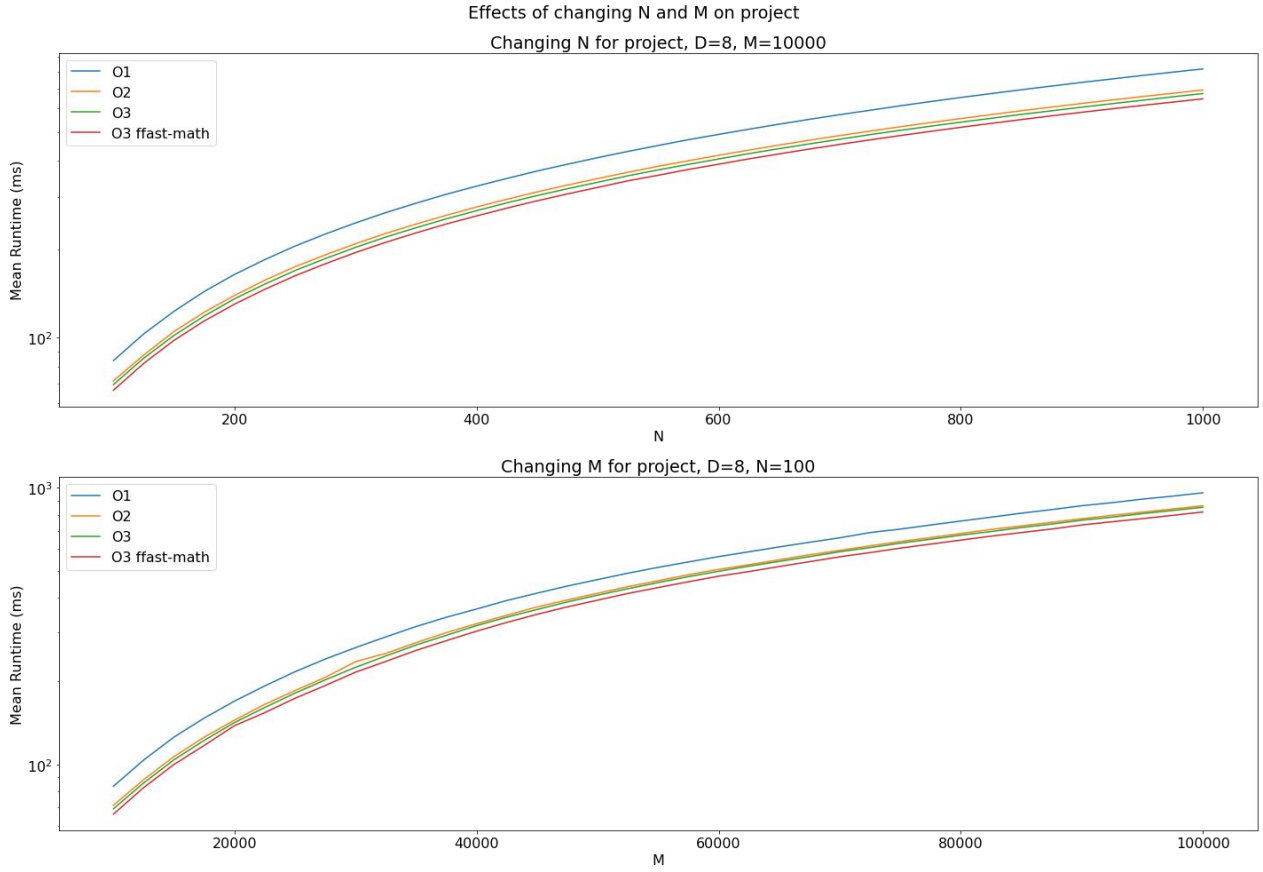

Figure 4: Performance results without $-O0$ included and log-scaled y-axis.

With compiler optimisations turned off i.e. using the -$O0$ flag explicitly, the runtime increases gradually from a mean of 1298.15 ms when $(N, M) = (100,10000)$ to a mean of 12891.65 ms when $(N, M) = (1000,10000)$ and 12950.15 ms when $(N, M) = (100,100000)$. Switching to the -$O1$ flag, the same $(N, M) = (100,10000)$ test yields a mean of 83.65 ms rising to 817.3 ms and 959.2 ms when $(N, M) = (1000,10000)$ and $(N, M) = (100,100000)$ respectively. This is quite an extreme performance improvement as switching flags from -$O0$ to -$O1$ alone results in an approximate 15 times performance increase through reduced runtime. Observe that these results propose an interesting relationship that suggests one-to-one correspondence between the test parameters and runtime. In other words, for a given compiler optimisation level, scaling the base value of N or M by a factor of 10 whilst holding the other fixed leads to the serial implementation's base runtime to also scale approximately by a factor of 10.

The differences between -$O1$, -$O1$, -$O2$, -$O3$ and -$O3$ -$ffast\text{-}math$ are quite hard to distinguish using Figure 3 alone. Hence, Figure 4 was created with this in mind by initially removing all the -$O0$ test data and setting the y-axis to be logarithmic thus leading to a semi-log plot. The outcome is astonishing because bearing in mind that although 20 samples were generated prior to taking the mean, the logarithmic curves seen in Figure 4 are extremely smooth when increasing either N or M. Furthermore, each higher optimisation level ever so slightly reduces the mean runtime relative to the previous one. Although, these aren't significant enough to draw attention towards.

Overall, it appears that sufficient attention to program design together with some manually implemented optimisations and compiler optimisations turned on results in good performance scaling for the serial implementation to the European basket options pricing problem.

Chapter 2
Parallel Implementation

2.1 Introduction

Following from the serial implementation in chapter 1 of this report, the parallel implementation for the European basket options pricing problem will be introduced and discussed thoroughly in this chapter. A brief overview of the two main parallelisation strategies that were realised over the course of COSC7502 to parallelise the problem are initially provided. Next, the chosen parallelisation strategy is discussed in depth. Commencing with the specific parallel programming model and platform chosen, it is then highlighted how the parallel approach specifically builds on the serial implementation followed by the new algorithm in pseudocode. Specific details about program design are then brought to attention followed by a graphical illustration of the verification procedure from chapter 1 reused here to ensure validity and correctness of both implementations. Experimentation plans to thoroughly assess the parallel approach against the serial approach are mentioned before displaying the results of said tests along with the problem scaling ability of both implementations. Finally, the report is concluded with a reflection over the entire project along with the difficulties encountered and the possible extensions to further parallelising the problem.

2.2 Parallelisation Strategies for European Basket Options Pricing

Financial mathematics is a relatively new, small branch of applied mathematics known by many to have emerged as a true discipline only after the BSM model was first released in 1973 (Black et al., 1973). Despite this, there is great interest to date in finding novel and better solutions to quite difficult high-dimensional financial mathematics problems such as the European basket options pricing problem. Attempts to research and find existing parallel approaches to this specific problem over the internet were futile since there weren't any relevant results available for this specific problem that didn't involve completely revising the underlying simulation algorithm used to generate MC samples. However, over the course of COSC7502 this semester, two possible approaches to parallelisation were self-realised.

The biggest barrier to parallelising the problem was satisfying the dependency to consistently generate correlated standard numbers random numbers LZ_n at each timestep n during Euler timestepping. In other words, if each asset did have a constant non-zero correlation set initially with all other assets, it would be impossible to simulate each asset's price path completely independently from every other assets' price paths because of the inherent lack of independence between the price generating processes. Note that allowing for non-zero correlations in both implementations allows solving more realistic problems because asset dependencies on other assets is almost always true in the real world.

The first parallel approach involves generating and storing all the correlated standard numbers random numbers LZ_n for each timestep $0 \leq n \leq N-1$ in advance prior to performing Euler timestepping on each asset. More specifically, during Euler timestepping, each asset's sample's price at the next timestep $\hat{S}_{n+1}^{(d,m)}$ would need to access $(LZ_n)^{(d,m)}$ from an array of matrices $[LZ_0, LZ_1, \dots, LZ_n, \dots, LZ_{N-1}]$ that was already generated in advance. This approach handles the correlation-dependency aspect of the problem but evidently, being a necessity to store all the LZ_n is extremely memory inefficient. For example, using single-precision 32-bit floats, with $D = 8$ assets, $N = 100$ timesteps and $M = 10000$ samples, such an array of matrices requires at least 32 MB of memory during runtime not accounting for the other variables. In contrast, with $D = 8$ assets, $N = 100$ timesteps and $M = 100000$ samples, this is now equivalent to at least 320 MB of memory. It is quite clear that such an approach would be unable to scale well in terms of N and M without requiring an unreasonable strain on computational resources.

The second parallel approach is quite straightforward as it aims to parallelise the serial implementation exactly as it is. This means in a single timestep n, the correlated standard numbers random numbers LZ_n are generated in parallel. Some synchronisation/barrier is put in place for this computation to sync up followed by performing Euler timestepping on each asset using the generated LZ_n. There is no need to store the previous LZ_n at the next timestep $n+1$. For the larger problem size of $D = 8$ assets, $N = 100$ timesteps and $M = 100000$ samples as above, using single-precision 32-bit floats, just 3.2 MB of memory is consumed overall to generate each LZ_n during runtime not accounting for the other variables. Observe that this minimum memory required remains fixed with some given D, M no matter how large N becomes since each LZ_n is no longer generated in advance. The downside of this approach is a lack of full program independence – the beforementioned barrier is required to prevent race conditions from occurring i.e. some $\hat{S}_{n+1}^{(d,m)}$ accessing memory that might be currently being written to by $(LZ_n)^{(d,m)}$. Given that the efficiency benefits far outweigh the costs via much greater problem scaling, this was the parallelisation approach selected and implemented in the remainder of this chapter.

2.3 Implementation Details

2.3.1 Parallel Programming Model

The second approach mentioned, involving parallelising the serial implementation as is, is quite easy to parallelise and can be done by all parallel programming models learnt in COSC7502. However, due to deep personal interest, the CUDA parallel programming platform and model developed by NVIDIA Corporation was used solely to create the parallel implementation. To the uninitiated, CUDA permits developers to harness the power of NVIDIA GPUs to parallelise the parallelisable computation within their programs which can often result in respectable speedups.

CUDA can be used in the second approach by parallelising all computation involving vectors and matrices appearing in the serial implementation via the CUDA concepts of 1D and 2D CUDA thread grids respectively. This means parallelising any vector/matrix operations as well as element-wise operations that appear in Algorithm 1. Specifically, this involves parallelising all parts of the code involving the variables $\boldsymbol{S_0}, \boldsymbol{\mu}, \boldsymbol{\sigma}, \mathrm{P}, L, Z_n, LZ_n, \hat{S}_n$ and $\boldsymbol{Y}$.

Short explanations of how CUDA thread grids can be used for this problem now follow. Starting with simple 1D arrays, consider the vector $\boldsymbol{Y}$. After the final matrix of simulated asset prices $\hat{S}_N$ has been generated, each element of $\boldsymbol{Y}$ can be computed serially as per the Euler scheme previously specified in Algorithm 1. The diagram in figure 5 below can be used to visualise how this would work in parallel. The diagram uses 256 GPU threads per thread block and about 4096 thread blocks in total, although these specific values aren't used in the actual implementation (illustration purposes only). Using the figure, $Y^{(516)}$ can be computed by a single GPU thread on the device and stored in GPU memory at location $threadIdx(3)$ within $blockIdx(2)$ ($1 \leq m \leq M$ but indexing starts from 0 in C++ so need to use -1 offsets).

Similarly, a small subsection of a complete 2D CUDA thread grid is provided in figure 6 below to help illustrate the next example. $Z_n^{(1,1)}$ can be generated $\sim N(0,1)$ by some normal RNG algorithm on the GPU device by a single GPU thread and then subsequently stored at $threadIdx(0,0)$ within $blockIdx(0,0)$ within GPU memory. Similarly, $\hat{S}_{n+1}^{(3,7)}$ is computed via the Euler scheme from Algorithm 1 using old data $\hat{S}_n^{(3,7)}$ and then reassigned to the same location in GPU memory by a single GPU thread. The corresponding location would be at $threadIdx(0,2)$ within $blockIdx(2,0)$ using -1 offsets.

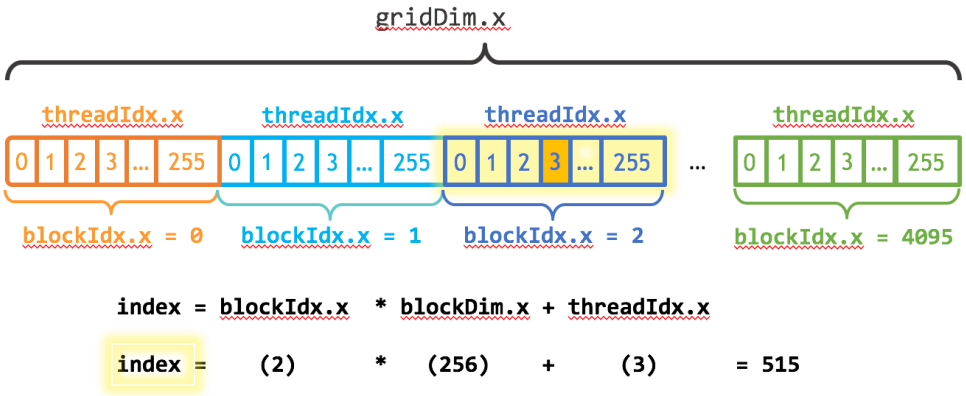

Figure 5: Diagram illustrating the structure of 1D CUDA thread grid (Harris, 2017).

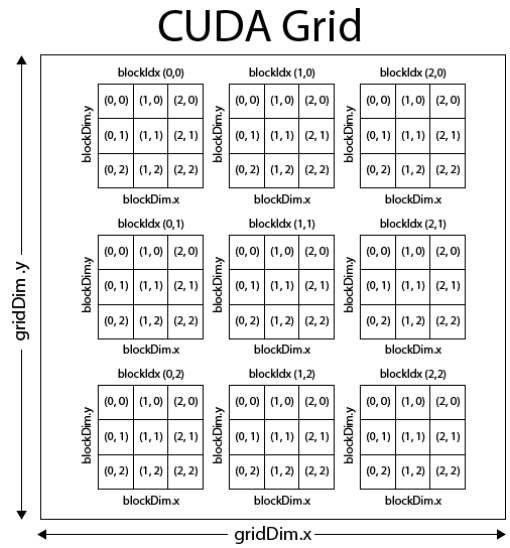

Figure 6: Diagram illustrating the structure of 2D CUDA thread grids (McKennon, 2013).

Finally, computing the mean and standard deviation of $\boldsymbol{Y}$, stored in $\hat{C}_M$ and $\hat{\sigma}_M$ respectively, can also be done in parallel via the concept of parallel reduction. This will be explained in much greater detail towards the end of section 2.3.3 below but for now, observe that both statistics involve computing a summation – $\hat{C}_M = \frac{1}{M}\sum_{m=1}^{M} Y^{(m)}$ and $\hat{\sigma}_M = \sqrt{\frac{1}{M-1}\sum_{m=1}^{M}\left(Y^{(m)} - \hat{C}_M\right)^2}$ from Algorithm 1. Figure 7 below provides insight as to how the simple sum of a vector can be computed in parallel through parallel reduction.

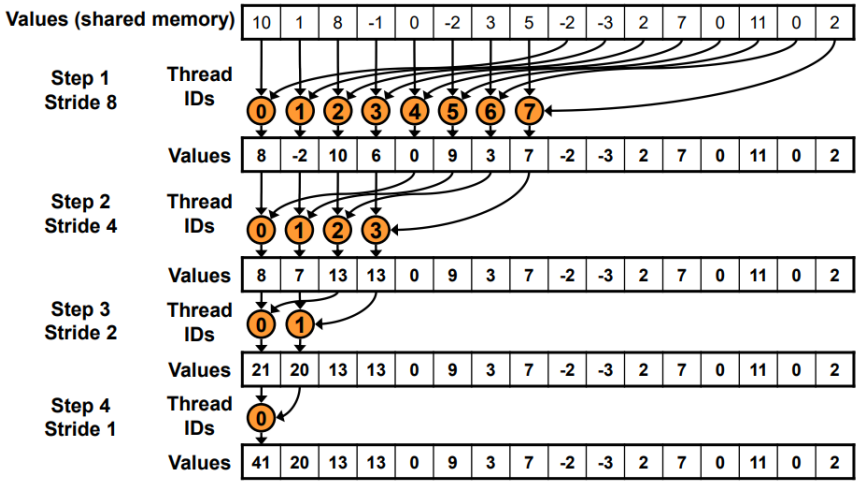

Figure 7: Computing the sum of an array with $N = 16$ elements in parallel via sequential addressing (Harris).

In every iteration, the loop stride is set to equal the number of elements being considered divided by 2. Hence, with $N = 16$ array elements, in the first iteration, the initial $stride = \frac{N}{2} = 8$. As in figure 7, elements at indices (0, 8) are summed and placed at index 0, elements at indices (1, 9) are summed and placed at index 1, etc. In the next iteration, only the first half of the original array is considered because this is the only relevant part of the array storing the sums from the previous iteration. The stride length is reduced to $stride = \frac{N}{4} = 4$ and elements at indices (0, 4) are summed at placed at index 0, elements at indices (1, 5) are summed at placed at index 1, etc. This process repeats until the stride length $= 1$ i.e. roughly $\frac{N}{2^i} = 1 \rightarrow i = \log_2(N)$ iterations. At the end, the final solution for the sum of the entire array will be stored at index 0 from summing the elements at indices (0, 1). Note that this same methodology gives rise to several other similar parallel reduction algorithms for operations such as $\min\{\}$, $\max\{\}$, etc.

2.3.2 Algorithm

The pseudocode for the parallel implementation is provided in Table 2 on the next page. After the short discussion on 1D and 2D CUDA thread grids and parallel reduction algorithms above, Algorithm 2 follows logically. Since the parallel approach does not make any changes to the internal mechanics of the simulation process but instead just parallelises the serial implementation as is, Algorithm 2 appears very similar to Algorithm 1 except, where possible, most of the computation is done on the GPU device instead. This is done in standard CUDA semantics – first allocating some memory on the device for computation, then copying any required memory from the host to the device, then performing the actual computation on the device, then finally copying back the required results back to the host and printing any desired output.

The main simulation loop occurs from lines 8 to 23 with the Euler scheme appearing on line 18. For simplicity, in the code $\hat{S}_0 = [[100, \dots, 100]^T, \dots, [100, \dots, 100]^T] \in \mathbb{R}^{D \times M}$ and $\boldsymbol{\mu} = [0.02, \dots, 0.02]^T, \boldsymbol{\sigma} = [0.3, \dots, 0.3]^T$ for $1 \leq d \leq D$ i.e. all assets' samples share the same initial asset price, constant drift and diffusion. $D = 8$ assets, $N = 100$ timesteps and $M = 10000$ samples are the base simulation settings. Furthermore, $K = 100, r = 0.02, T = 1$ and $\mathrm{P} = \mathbb{I}$ (the identity matrix) to represent fully uncorrelated assets.

Since both Algorithms 1 and 2 are quite similar, explanation for individual segments of the pseudocode below is futile as the reader can simply revisit the thorough description of Algorithm 1 provided in section 1.3.1 for any clarifications. However, there is one subtlety specific to Algorithm 2. As shown on lines 3 to 4, it is indeed necessary to allocate GPU memory and initialise variables such as $\sqrt{\Delta t}$ and $\exp(-rT)$ on the device. This is because basic math functions such as square root $sqrt()$ or exponential $exp()$ made available via the standard C++ math library $< cmath >$ are only computable on the host but not the device. This is logical since the code that implements these functions is written for compiling and execution via CPUs, not GPUs. The easiest solution to this issue is to pre-compute $\sqrt{\Delta t}$ and $\exp(-rT)$ initially on the host and then copy the resulting value onto new variables stored on the device, as is indicated by the pseudocode.

Algorithm 2: *Parallel Implementation for MC Simulation for European Basket Options Pricing*

declare D, N, M, K, r, T **on host**;
initialise D, N, M, K, r, T **on host**;
allocate GPU memory $D, M, K, r, \Delta t \leftarrow \frac{T}{N}, \sqrt{\Delta t}, \exp(-rT)$ **on device**;
initialise $D, M, K, r, \Delta t \leftarrow \frac{T}{N}, \sqrt{\Delta t}, \exp(-rT)$ **on device**;
allocate GPU memory $\hat{S}_0 \in \mathbb{R}^{D\times M}, \boldsymbol{\mu}, \boldsymbol{\sigma}, \mathrm{P}, L, \mathrm{P}, Z_0, LZ_0, \boldsymbol{Y}$ **on device**;
initialise $\hat{S}_0, \boldsymbol{\mu}, \boldsymbol{\sigma}, \mathrm{P}, \boldsymbol{Y} \leftarrow \mathbf{0}$ **on device**;
compute $L \leftarrow cholesky\_decomposition(\mathrm{P})$ **on device**;
for $0 \leq n \leq N-1$ **do**
    for $1 \leq d \leq D$ **do in parallel**
        for $1 \leq m \leq M$ **do in parallel**
            // Store Z_n in same memory location as Z_{n-1} (i.e. overwrite old data)
            generate $Z_n^{(d,m)} \leftarrow \sim^{iid} N(0,1)$ **on device**;
        end
    end
    compute $LZ_n \leftarrow L \times Z_n$ **on device**;
    for $1 \leq d \leq D$ **do in parallel**
        for $1 \leq m \leq M$ **do in parallel**
            // Store $\hat{S}_{n+1}$ in same memory location as $\hat{S}_n$ (i.e. overwrite old data)
            compute $\hat{S}_{n+1}{}^{(d,m)} \leftarrow \hat{S}_n{}^{(d,m)} + \mu_d \hat{S}_n{}^{(d,m)} \Delta t + \sigma_d \hat{S}_n{}^{(d,m)} (LZ_n)^{(d,m)} \sqrt{\Delta t}$ **on device**;
            set $\hat{S}_{n+1}{}^{(d,m)} \leftarrow \max\left(\hat{S}_{n+1}{}^{(d,m)}, 0\right)$ **on device**;
        end
    end
end
for $1 \leq m \leq M$ **do in parallel**
    $Y^{(m)} \leftarrow \exp(-rT) \times \max\left(\frac{1}{D}\sum_{d=1}^{D} \hat{S}_N^{(d,m)} - K, 0\right)$;
end
compute $\hat{C}_M \leftarrow \frac{1}{M}\sum_{m=1}^{M} Y^{(m)}$ **via parallel reduction and store on host**;
compute $\hat{\sigma}_M \leftarrow \sqrt{\frac{1}{M-1}\sum_{m=1}^{M}\left(Y^{(m)} - \hat{C}_M\right)^2}$ **via parallel reduction and store on host**;
set $z \leftarrow 1.96$ **on host**;
compute $CI\_left \leftarrow \hat{C}_M - z\frac{\hat{\sigma}_M}{\sqrt{M}}$ **on host**;
compute $CI\_right \leftarrow \hat{C}_M + z\frac{\hat{\sigma}_M}{\sqrt{M}}$ **on host**;
compute $radius \leftarrow z\frac{\hat{\sigma}_M}{\sqrt{M}}$ **on host**;
print $D, N, M, \hat{C}_M, [CI\_left, CI\_right], radius$ **on host**;

Table 2: Pseudocode for the parallel implementation.

2.3.3 Program Design

The specifics of the parallel implementation program design shall now be discussed in this section. NVIDIA provides a free collection of libraries, tools and technologies known as CUDA-X to developers which is made available through both their CUDA Toolkit and their HPC SDK. The advantages of using libraries from CUDA-X are logically similar to using any other library (Cohen, 2014):

- Much higher confidence in computation results given dedicated team validating correctness across every supported and upcoming hardware platform.
- Highly optimised functions and methods tested thoroughly via performance regressions across same wide range of platforms.
- Enables developers to save time on writing their own GPU libraries from scratch, debugging.

- Enables developers to instead spend time on other important aspects such as logic, features, deployment to production, etc.
- Wide support and help made available via NVIDIA developer forums.

Hence, whilst it is certainly possible to implement the parallel approach entirely using 1D and 2D CUDA thread grids as demonstrated in section 2.3.1, there is no need to do so.

Specifics about different library usage in the parallel implementation shall now be clarified. From CUDA-X, the $cuRAND, cuSOLVER$ and $cuBLAS$ math libraries and $thrust$ parallel algorithm library were used to solve the European basket options pricing problem. Detailed descriptions of the functions and respective arguments can be obtained from the CUDA Toolkit Documentation available through the NVIDIA Developer website (NVIDIA, 2021). First, within the $cuRAND$ library, the $curandCreateGenerator(), curandSetPseudoRandomGeneratorSeed()$ and $curandGenerateNormal()$ functions were used to create a RNG, set the seed for RNG and generate $\sim N(0,1)$ standard normal random numbers for each Z_n in parallel respectively. Note that the last function performs lines 9 to 14 from Algorithm 2 in parallel as required. Next, within the $cuSOLVER$ library, the $cusolverDnCreate(), cusolverDnSpotrf\_bufferSize()$ and $cusolverDnSpotrf()$ functions were used to initialise the $cuSolverDN$ library and create a handle on the $cuSolverDN$ context, calculate the necessary size of work buffers required to Cholesky factorise P and compute the actual Cholesky factorisation of P respectively. Within the $cuBLAS$ library, the $cublasSgemm()$ and $cublasSgeam()$ functions were used to multiply matrices $L \times Z_n$ and compute matrix transpose $Z_n^T L^T = (LZ_n)^T$ respectively. Note that the matrix transpose $Z_n^T L^T$ at each timestep n is not necessary which is why it was excluded from Algorithms 1 and 2. However, it was used in the parallel code since the serial code required this transpose at each timestep n in order for the program to run without errors. As explained earlier, the original intention with the parallel implementation was to keep it as similar as possible to the serial implementation in order to test the raw parallelisation power of GPUs. It would not be helpful but unfair if significant speedups were achieved in the parallel implementation by skipping transposing LZ_n at each timestep for $0 \leq n \leq N-1$ which was required for the serial code to work properly. Note that this means $\hat{S}_n \in \mathbb{R}^{M \times D}$ in both implementations rather than $\in \mathbb{R}^{D \times M}$. Finally, within the $thrust$ library, the $reduce()$ function was used to compute $\hat{C}_M$ and $\hat{\sigma}_M$. Following from the description at the end of section 2.3.1 on how parallel reduction algorithms work, $\hat{C}_M$ is computed simply by summing $\boldsymbol{Y}$ and dividing by M to obtain the sample mean. Then, a new vector is declared and initialised storing the squared differences between each element of $\boldsymbol{Y}$ and $\hat{C}_M$. This is done via a self-created function using a 1D CUDA thread grid (new vector $\in \mathbb{R}^{M \times 1}$). Finally, $\hat{\sigma}_M$ is computed by calling the $reduce()$ method once again to compute the sum of this new vector, then dividing by $(M-1)$ and square rooting to obtain the sample standard deviation. Both are used to again create a 95% confidence interval on the simulated European basket option price.

2.3.4 Verification Procedure

In order to verify the correctness of the parallel implementation, the same verification procedure from section 1.3.3 is reused here. However, instead of only being able to verify with the true solution provided by the MATLAB implementation, the serial implementation can be used now to ensure that the parallel program is producing stochastic but expected results.

Regarding the incremental construction of the parallel program, arriving at the final working version was very time consuming as is expected when developing with CUDA. This is because functions from CUDA libraries can produce silent errors that can easily go undetected unless manual error checking is put in place. Hence, for each of functions belonging to the $cuRAND, cuSOLVER$ and $cuBLAS$ libraries that were used in any computation performed on the device, a separate helper function was created and used to check for any errors raised by these functions. The $thrust$ library automatically raises runtime errors, so a separate helper function was not required whenever the $reduce()$ method was used. More specifically, the following functions were created to help in debugging at program runtime:

- $check\_general\_errors()$ for any general CUDA errors that occur at runtime e.g. failed memory copy, etc.
- $check\_cuRAND\_errors()$ for any errors raised by functions belonging specifically to the $cuRAND$ library.

- $check\_cuSOLVER\_errors()$ for any errors raised by functions belonging specifically to the $cuSOLVER$ library.
- $check\_cholesky\_success()$ for any errors raised specifically by the Cholesky factorisation of P.
    - Can happen if P is not initialised with asset correlation values that make it a Hermitian positive-definite matrix.
- $check\_cuBLAS\_errors()$ for any errors raised by functions belonging specifically to the $cuBLAS$ library.

The special case of setting the BMs' correlation matrix $\mathrm{P} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix}$ i.e. matrix of ones to verify correctness of both implementations can be reused here. Recall this means every asset is perfectly positively correlated with every other asset. Hence, the price dynamics for one asset perfectly replicates every other asset's price dynamics. If the reader no longer remembers, please refer back to section 1.3.3 for a complete mathematical reasoning of why pricing such a basket of assets is then equivalent to just pricing a single asset. Note that this also requires the values of S_0, μ, σ to be shared between all assets for this to work.

Following feedback from the serial implementation submission, figure 8 below was produced to provide a graphical illustration verifying the correctness of both implementations. The exact price plot line, equivalent to $C_0 \approx 12.8216$, was generated via the B-S formula implemented in MATLAB's Financial toolbox's $blsprice()$ using $S_0 = 100, K = 100, \mu = 0.02, T = 1, \sigma = 0.3$. It is evident from the figure that using this special case P along with common input for all assets, both implementations work as intended. There is high confidence that the programs will yield similar results to each other for all other types of valid input e.g. varying $\hat{S}_0, \boldsymbol{\mu}, \boldsymbol{\sigma}$, using a different correlation matrix P and so on. Exact results cannot be expected due to the stochastic nature of both programs but note that both implementations should get closer and closer as M gets larger since more samples will be used in computing $\hat{C}_M$. This might not look like the case in figure 8 below but that is potentially due to M not being large enough (might need to set to at least > 100000). The fact that $\hat{C}_M$ for both implementations is sometimes above and other times below the exact price provides reassurance that the RNG mechanism used to compute Z_n in both programs does not seem to contain any bias.

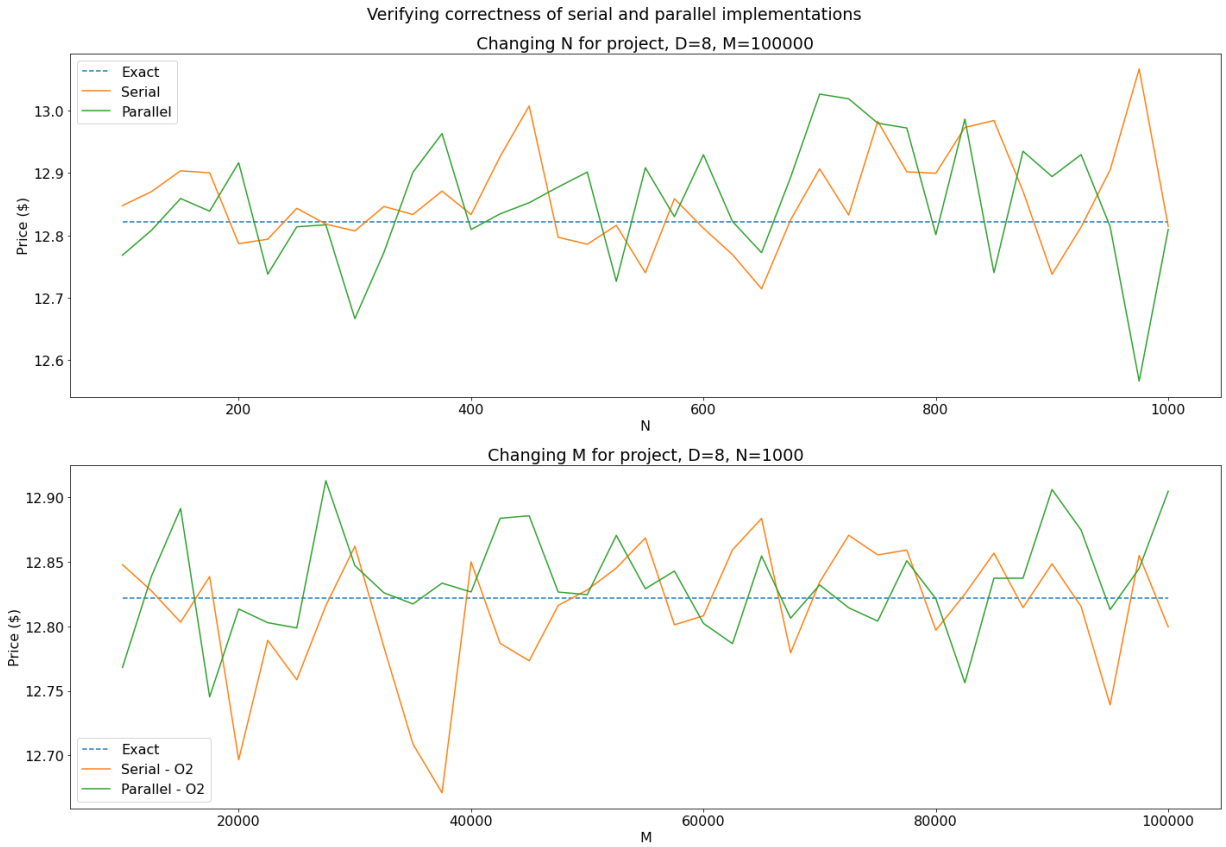

Figure 8: Diagram verifying the correctness of both implementations.

2.4 Experimentation Plan

Up until this point, all of the parallel implementation's construction, compiling, execution and debugging was done on a personal Windows PC using an NVIDIA RTX 3060 Ti GPU. This section aims to briefly mention the thorough experimentation plan to be conducted separately on UQ SMP's Getafix cluster using one of its many available NVIDIA V100 GPUs instead. The specifics of these plans will be detailed in the next section.

The parallel program was first verified to be working as intended via the verification procedure from sections 1.3.3 and 2.3.4 together with comparing both implementations' outputs across a range of random N, M values (e.g. $(100,12500), (300,15000)$, etc.). Important initial parameters $\hat{S}_0, \boldsymbol{\mu}, \boldsymbol{\sigma}, K, r, T, \mathrm{P}$ were left unchanged from the serial implementation in order to be able to compare against figures 3 and 4 later on in the results section. Next, a swift check was done to test whether the parallel program's runtime performance was affected by compiler-based optimisation flags. Using the -$O2$ flag over the -$O0$ flag consistently resulted in significant speedups across different values of D, N, M whilst always satisfying the verification procedure once again. Hence, the -$O2$ flag was always set in all of the experiments when compiling the parallel program. Other optimisation flags such as -$O1$ and -$O3$ were disregarded to avoid creating extremely convoluted diagrams for the performance results section.

Finally, the complete experimentation plan that was arrived at can be divided into three separate parts:

1) Testing the parallel implementation on the same set of D, N, M values that were used to benchmark the serial implementation in section 1.4, in order to obtain a basic comparison of both implementations.
2) Testing both implementations' ability to solve much larger problems i.e. problem scaling. Logically, this is a much larger set of D, N, M values than the ones used in step 1) above.
3) Testing how changing the number of GPU threads used per block to compute $\hat{S}_{n+1}$ affects runtime performance on large D, N, M values.

2.5 Experimentation Methodology

Conducting the first step of the experimentation plan listed above was quite straightforward. With $D = 8$ fixed, the parallel implementation was first tested on values of N increasing from 100 to 1000 timesteps in increments of 25 keeping $M = 10000$ samples fixed, then on values of M increasing from 10000 to 100000 samples in increments of 2500 keeping $N = 100$ timesteps fixed. Bear in mind that these are the same values the serial implementation was benchmarked on to create figures 3 and 4 in section 1.4. Accordingly, 20 runs would be conducted on each test to create averages to be used for the resulting figures, as in the serial implementation. A hypothesis was made prior to carrying out these benchmark tests – it was expected that there would be a significant non-trivial overhead associated with device memory allocation, host to device and device to host memory transfers given the number of times these calls appeared in the parallel code. It was also anticipated that this cost would become easily apparent for small test sets via a longer runtime, most of it attributable to these device memory allocation and memory transfer calls. Indeed, from figure 3, the serial implementation tests often completed in less than 1 second when the optimisation flag was set to anything but -$O0$ raising the possibility the parallel implementation could actually run slower on these small values of N, M.

The second step of the experimentation plan aimed to test how well both implementations scale with very large problem sizes. The first change made was to instantly double D from 8 to 16 i.e. pricing a European basket option on 16 rather than 8 assets to add compute time. As before, the actual values set for $\hat{S}_0, \boldsymbol{\mu}, \boldsymbol{\sigma}, K, r, T, \mathrm{P}$ were left unchanged for simplicity and consistency with all previous tests. Both implementations were then first tested on values of N increasing from 1000 to 10000 timesteps in increments of 1000 keeping $M = 100000$ samples fixed, then on values of M increasing from 100000 to 1000000 samples in increments of 100000 keeping $N = 1000$ timesteps fixed. Comparing the smallest and largest tests then, the latter is a factor of 2000 times larger after:

- Doubling of $D = 8$ to $D = 16$.

- Thousand-fold increase from $N = 100$ to $N = 10000$ timesteps combined with increasing $M = 10000$ to $M = 100000$ samples.
- Or thousand-fold increase from $N = 100$ to $N = 1000$ timesteps combined with increasing $M = 10000$ to $M = 1000000$ samples.

The number of runs done on each benchmark test was reduced from 20 to 5 in anticipation of the serial implementation benchmarks possibly taking too long in this larger test set. Testing problem scaling via larger test sets also has the benefit of possibly dampening the significant overhead attributable to device memory allocation and memory transfers due to requiring more compute time relative to time used by these function calls in producing the final output.

The last step of the experimentation plan was to assess whether varying the number of GPU threads used to compute variables stored as 2D CUDA thread grids had any significant performance costs/benefits. The number of threads per block and number of grid blocks used in computing Z_n, LZ_n and $Z_n^T L^T$ were evaluated automatically at runtime by the respective CUDA library functions used for these variables. Hence, only the underlying layout of $\hat{S}_{n+1}$ could be altered via manually set threads in each dimension since each Euler timestep was performed using a custom kernel. Until this point, this custom kernel was always invoked with D threads for both dimensions i.e. a total of $D \times D$ threads per block. This was set to respect the second dimension of $\hat{S}_n \in \mathbb{R}^{M \times D}$. Having more than D threads for the second dimension would result in a waste as they could never be used without leading to memory access violations (the horizontal index would be out of bounds). However, there could certainly be more than D threads used in the first dimension resulting in rectangular rather than square grid blocks. But, there are two general dependencies that must be considered carefully when setting the number of threads per block and corresponding number of grid blocks. That is, the maximum number of threads per block is usually 1024 and the maximum number of grid blocks is 65535 for most older NVIDIA GPUs. This is important because for example, setting the number of threads per block too low for a test set can cause the corresponding required number of grid blocks to exceed this upper bound resulting in a runtime crash. This would occur when $M = 1000000$ samples and only 8 threads were used for the second dimension resulting in 125000 vertical blocks. Keeping this in mind, the vertical number of GPU threads was changed from D to also test 32, 64 vertical threads per block a total of 512, 1024 threads for block (keeping the number of threads horizontally fixed). Finally, note that these upper bounds are general – different GPUs will have different upper bounds on the maximum number of threads per block and number of grid blocks (the reader can find out for their specific NVIDIA GPU via $deviceQuery()$).

2.6 Performance Results

2.6.1 Performance Comparison on Small Problem Sets

Following the extensive discussion on the experimentation methodology conducted for each of the steps listed in the experimentation plan in section 2.4, the first set of benchmark tests comparing performance on small problem sets were conducted to produce figures 9 and 10 located below. Note that figures 3 and 4 have been reused here.

Figure 9 instantly verifies the hypothesis made earlier – there are clear significant non-trivial overheads most likely caused by needing to allocate GPU memory and conduct memory transfers back and forth resulting in the parallel implementation being slower than the serial implementation in all benchmark tests. The fact that, over 20 runs, parallel runtime actually seems to be ever so slightly decreasing with increasing N, M is proof that there only a small portion of each test's runtime attributable towards actual compute time to solve the problem.

Further evidence is found in figure 10 as even on a log-scaled y-axis, the parallel implementation's performance does not seem to worsen even when $N = 1000$ timesteps and $M = 100000$ samples are 10 times larger than their initial values of $N = 100$ timesteps and $M = 10000$ samples respectively. Judging by the rate at which the serial implementation's runtime is picking up with these small tests of N, M pairs, it appears likely the parallel implementation could potentially have better performance with bigger problem sizes via high problem scaling.

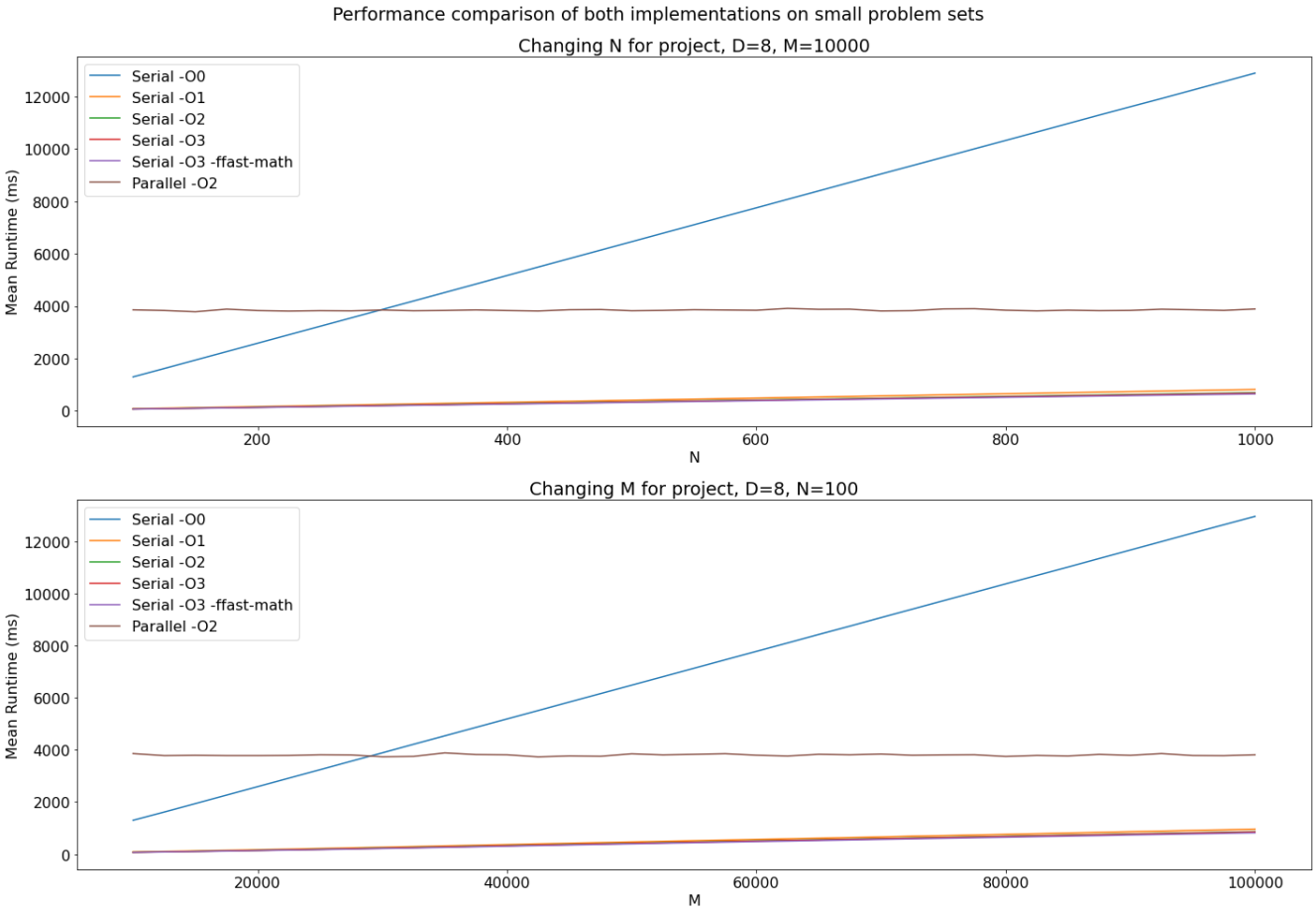

Figure 9: Performance results of both implementations on small problem sets with Serial -$O0$ included and no scale used on y-axis.

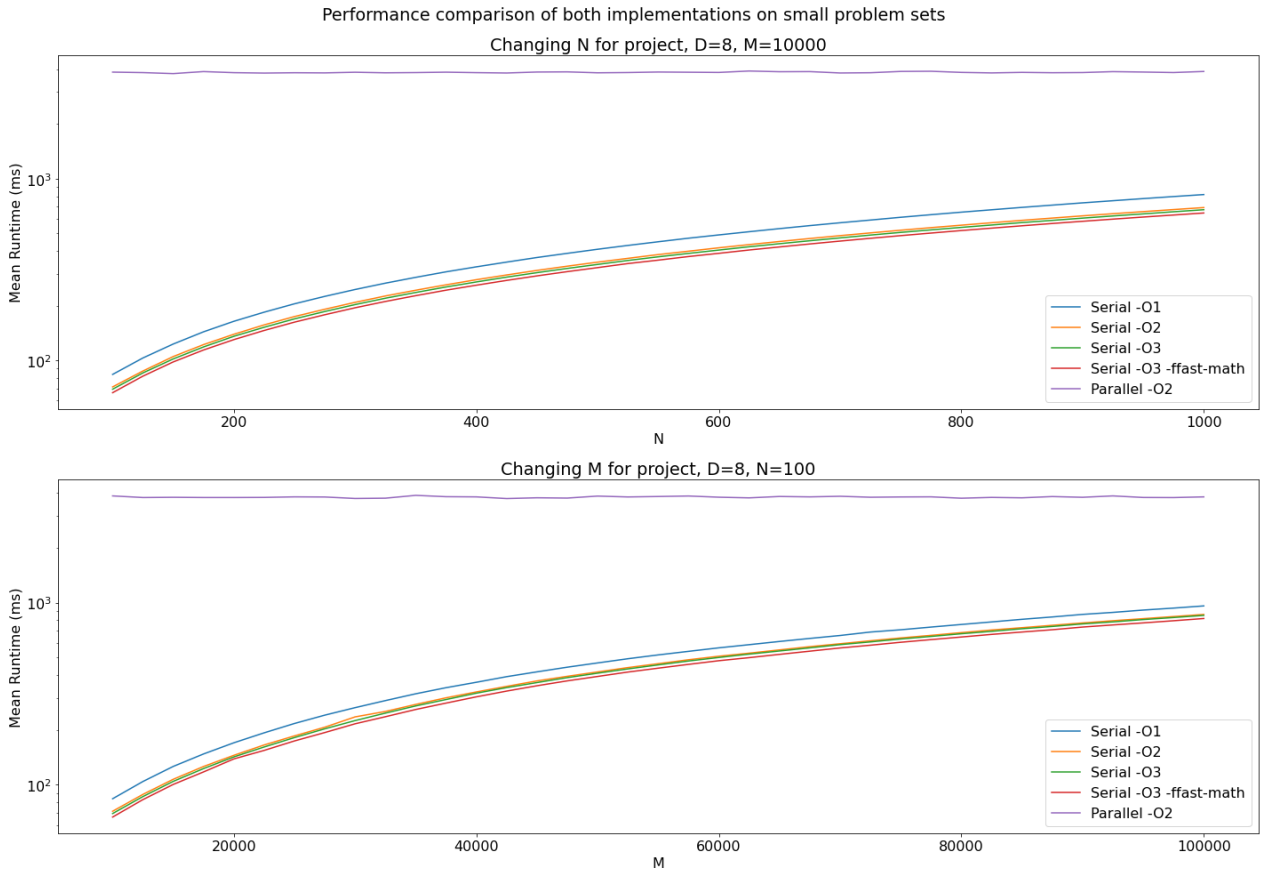

Figure 10: Performance results of both implementations on small problem sets with Serial -$O0$ removed and log-scaled y-axis.

2.6.2 Performance Comparison with High Problem Scaling

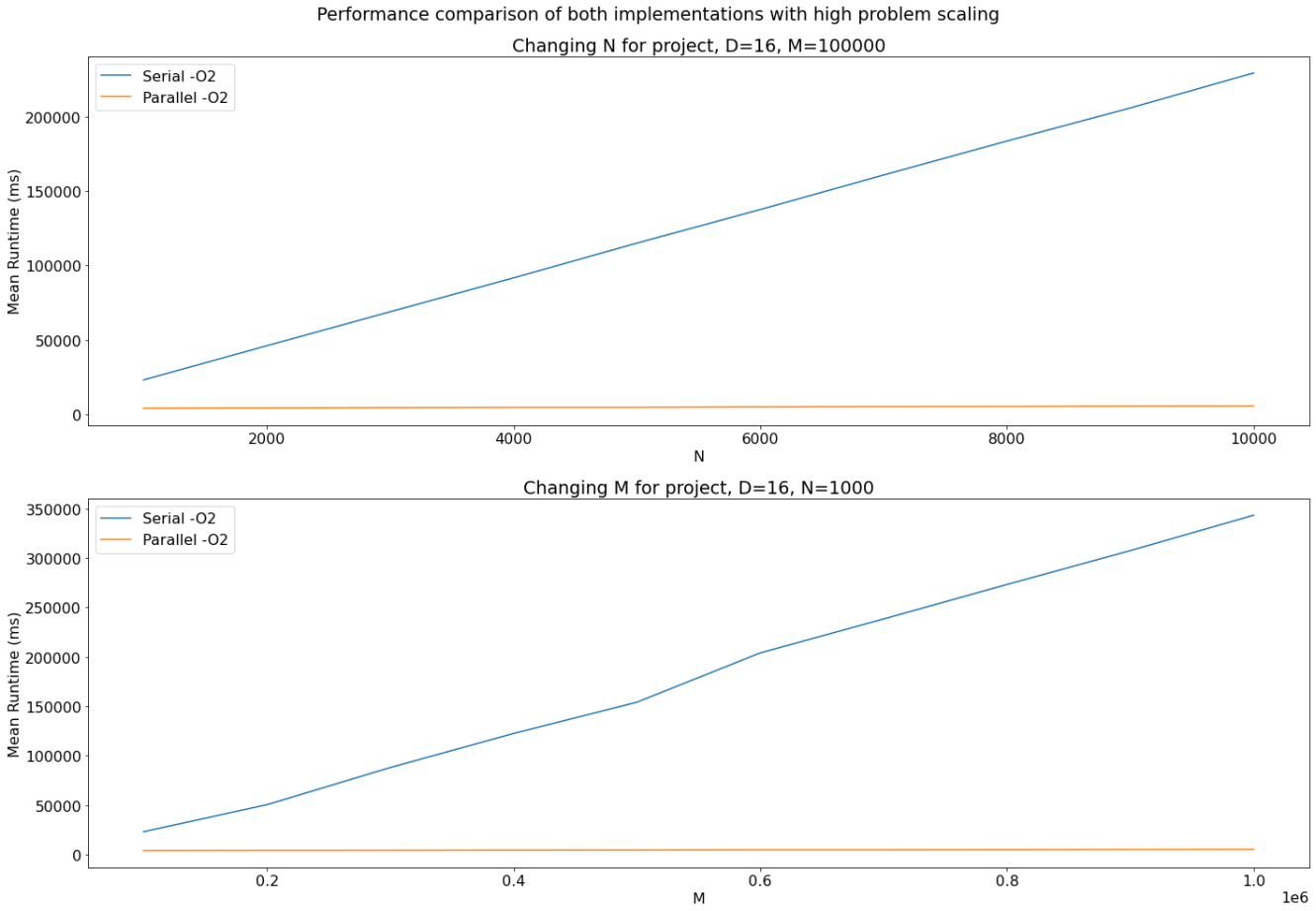

Figure 11: Performance results of both implementations with high problem scaling.

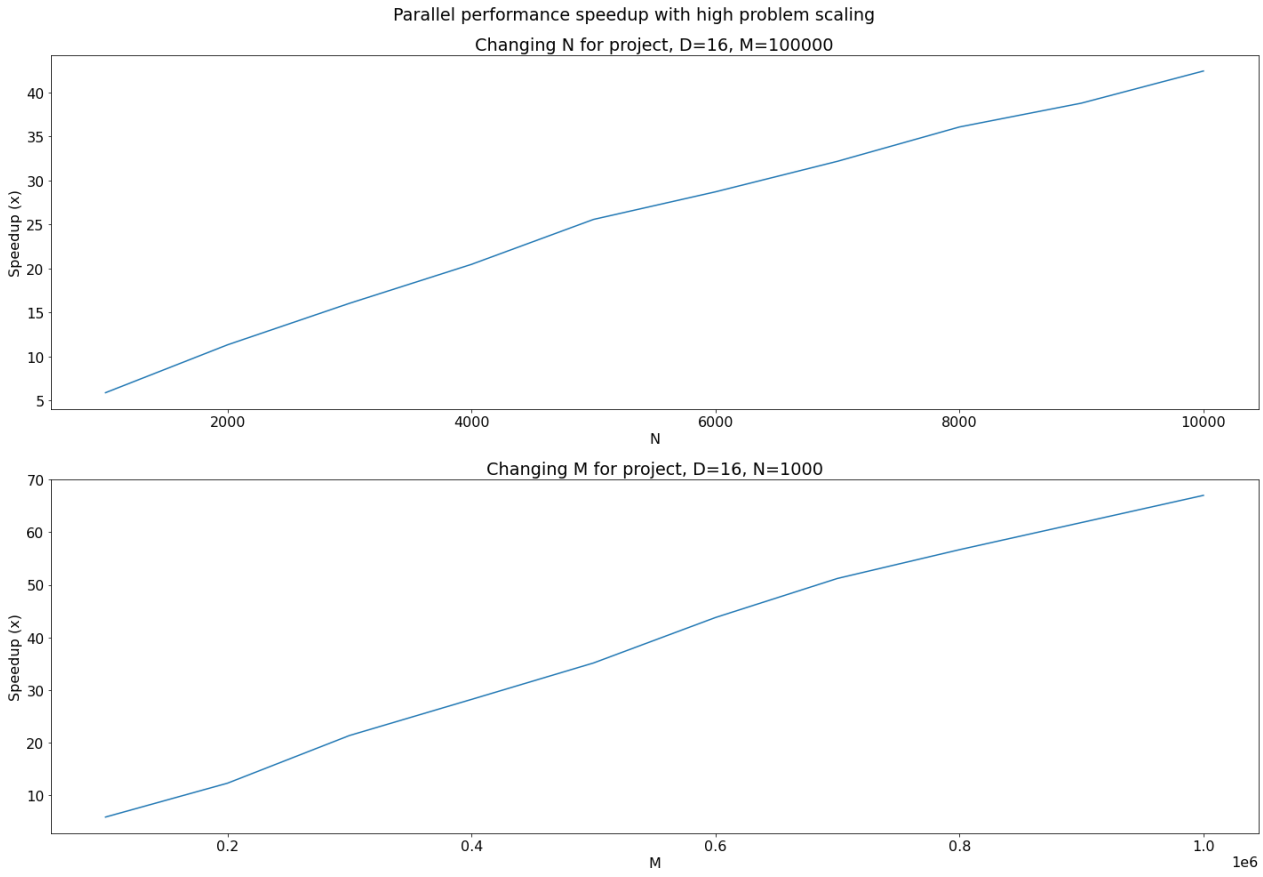

Figure 12: Parallel performance speedup with high problem scaling.

The second set of benchmark tests comparing performance on very large problem sets were conducted to produce figures 11 and 12 above. Despite the parallel implementation's poor performance on small problem sets, it outperformed the serial implementation in figures 11 and 12 when both were benchmarked on much larger problem sets using the methodology described in section 2.5. Based on the average of 5 runs, at $N = 10000$ timesteps and $M = 100000$ samples, the serial implementation took roughly 3 minutes and 52 seconds whereas the parallel implementation took approximately just under 5.4 seconds resulting in an incredible speedup ≈ 42 times faster. On the other spectrum, with $N = 1000$ timesteps and $M = 1000000$ samples, the serial implementation took roughly 5 minutes and 43 seconds whereas the parallel implementation took approximately just under 5.2 seconds resulting in an even more incredible speedup ≈ 67 times faster.

The fact that the parallel implementation took only 2 seconds more to solve problem sizes that were 2000 times larger from $D = 8, N = 100, M = 10000$ to $D = 16, N = 10000, M = 100000$ or $D = 16, N = 1000, M = 1000000$ highlights the exceptional performance scaling of the parallel implementation over the serial implementation for the European basket options pricing problem. Note that whilst it is possible to keep increasing D, N, M (ensuring number of threads per grid block and grid blocks dependencies are satisfied for custom GPU kernels), exact solutions will never be possible to the stochastic nature of MC simulation algorithms.

2.6.3 Using More Vertical GPU Threads in Computing $\hat{S}_n$

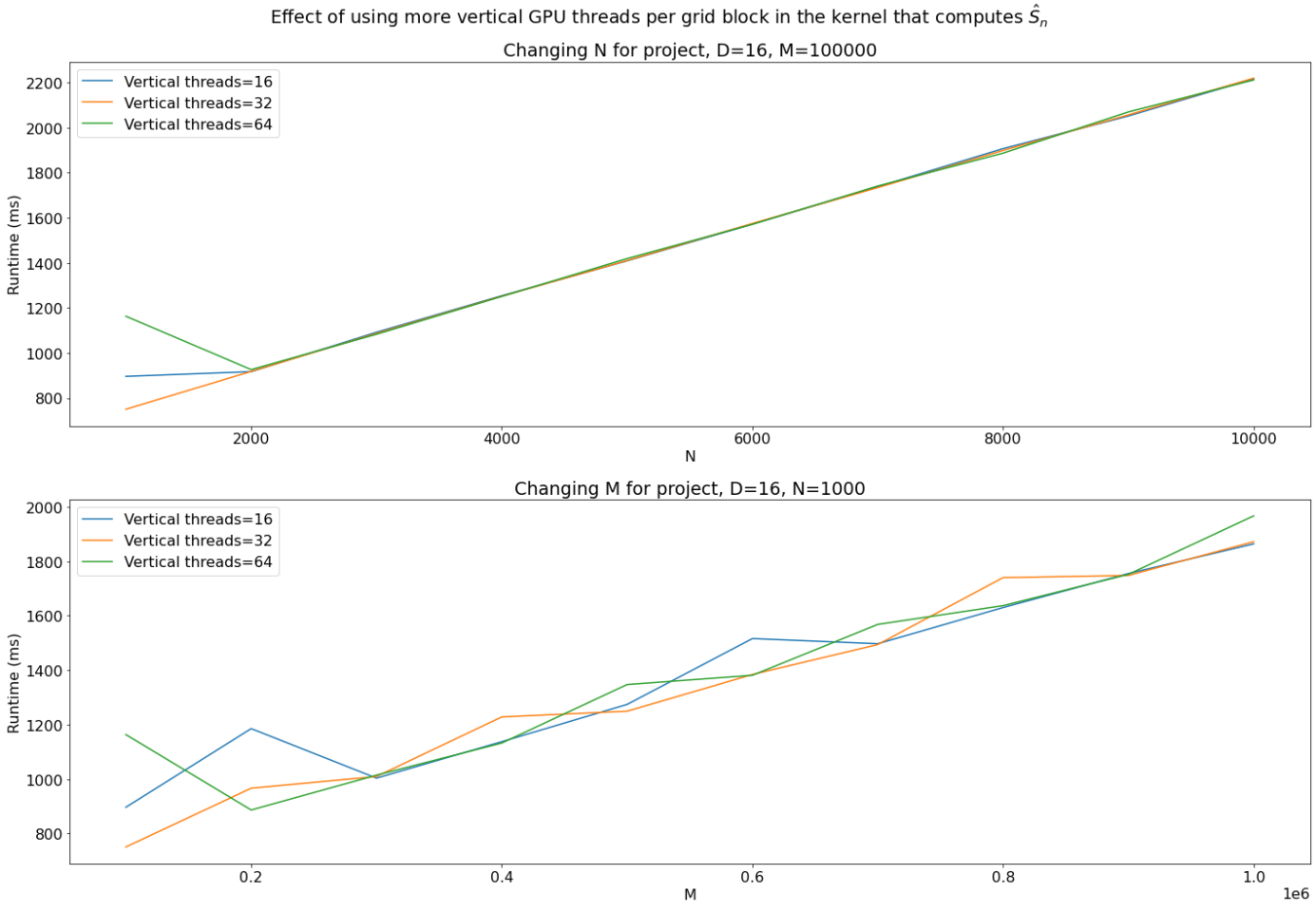

Figure 13: Performance effects of using more vertical threads per grid block in the kernel that computes $\hat{S}_n$.

The results of the last set of benchmark tests comparing the performance effects of changing the number of vertical threads per grid block used by the kernel that computes $\hat{S}_n$ at every time timestep n are displayed in figure 13 above. It is clear that there are no significant performance benefits/costs. This might be the case for this problem specifically because there is a direct trade-off between increasing (decreasing) the number of vertical threads resulting in decreased (increased) number of grid blocks overall (to cover the whole 2D CUDA thread grid) and vice-versa. Regarding the slight difference observable at $N = 1000$ timesteps and $M = 100000$ samples might be coincidental since only one run was performed in obtaining these results. Using more runs to generate averages for plotting in figure 13 would seemingly result in the smooth single plot line behaviour seen in the upper graph for $N = 2000$ to $N = 10000$ timesteps with $M = 100000$ samples.

2.7 Conclusion

In conclusion, this project report served to investigate high-performance computing techniques applied to the mathematical finance problem of European basket options pricing. Initially, an introduction to the context, terminology and underlying sophisticated mathematics governing the underlying problem was provided. Given that no closed-form solution exists, Monte Carlo simulation together with the Euler method was introduced as a simple technique to achieve a numerical solution to an otherwise extremely challenging task.

An algorithm for the serial implementation was introduced and provided in pseudocode before translating this algorithm into the C++ programming language. This implementation made use of the high-performance C++ Eigen library for linear algebra and the Ziggurat algorithm. After conducting the verification procedure, manual optimisations and profiling were conducted thereafter followed by the performance results for minor problem scaling with increasing input size and differing compiler optimisation flag. This was to highlight the performance scaling ability of the serial implementation to the problem.

In the next chapter, NVIDIA CUDA was discussed as to how this chosen parallel programming model could be applied specifically to the problem at hand. Following a basic explanation of how 1D and 2D CUDA thread grids work and parallel reduction algorithms, an algorithm for the parallel implementation was introduced and provided in pseudocode before translating into the CUDA C++ programming language. This implementation made use of several functions from highly optimised CUDA-X libraries such as $cuRAND, cuSOLVER, cuBLAS$ and $thrust$. The same verification procedure used for the serial implementation was reused to ensure everything was working correctly as intended. After arriving at a thorough experimentation plan and conducting it, the performance results were divided into three major sections – performance comparison on small problem sets, performance comparison using high problem scaling and the performance effects of using more vertical GPU threads in a 2D CUDA thread grid. In the end, the parallel implementation showed extremely promising and impressive results and is well suited to the European basket options pricing problem.

The most difficult aspect in both chapters was being able to obtain working versions of both implementations as the debugging process was quite challenging and time consuming in both programs. The Eigen library does not have the most support for clarifications on specific functions apart from the documentation. Whilst working with CUDA, the online developer community would sometimes point specify runtime issues occurring to incorrect function parameter usage, but also often times point the same issue to relate with compute capability of GPU, GPU virtual architecture, GPU driver issues, etc. There were also several memory access violation runtime crashes that were quite tedious to locate and debug.

There are also some noticeable limitations in the parallel implementation. Unfortunately, the $nvprof$ profiler provided by NVIDIA to aid with profiling CUDA programs was only discovered a few days prior to report submission and hence it was impossible to conduct profiling to check for GPU usage/activity as well as any program hotspots due to time constraints. In stating this however, some manual optimisations such as avoiding recalculating constants, removing unnecessary type conversions were still used. Furthermore, implementing in CUDA naturally results in

writing a much longer program (almost 3.5 times larger than serial implementation) that is simultaneously not easily readable due to CUDA's and CUDA-X libraries' strange function and method names.

To close, a natural future extension to the parallel implementation for the underlying problem would be investigating the use of multiple concurrent CUDA Streams or even GPUs to price multiple European basket options on multiple assets (with possibly differing dynamics for realism) simultaneously. This would certainly require using the aforementioned $nvprof$ profiler to check GPU usage to choose between multiple Streams or GPUs. For example, if only 50% of the GPU was used in total to price a single European basket option, then two of them could be priced simultaneously on a single GPU via multiple streams which would be the more efficient alternative in terms of utilised computational resources.

Bibliography

Doherty, B., 2020. *USING PUT OPTION CONTRACTS IN A WEATHER MARKET*. [online] **SuccessfulFarming**. Available at: https://www.agriculture.com/markets/analysis/the-us-crop-condition-percentage-ratings-are-in-the-60s-usda-reports [accessed 5 September 2021].

Hargrave, M., 2003. *Options Contract*. [online] **Investopedia**. Available at: https://www.investopedia.com/terms/o/optionscontract.asp [accessed 5 September 2021].

Ianieri, R., 2006. *Four Advantages of Options*. [online] **Investopedia**. Available at: https://www.investopedia.com/articles/optioninvestor/06/options4advantages.asp [accessed 5 September 2021].

Chen, J., 2007. *Discounting*. [online] **Investopedia**. Available at: https://www.investopedia.com/terms/d/discounting.asp [accessed 5 September 2021].

Downey, L., 2003. *Efficient Market Hypothesis (EMH)*. [online] **Investopedia**. Available at: https://www.investopedia.com/terms/e/efficientmarkethypothesis.asp [accessed 5 September 2021].

Black, F. and Scholes, M., 1973. *The Pricing of Options and Corporate Liabilities*. **The Journal of Political Economy**, [online] 81(3), pp.637-654. Available at: https://www.cs.princeton.edu/courses/archive/fall09/cos323/papers/black_scholes73.pdf [accessed 5 September 2021].

Chen, J., 2003. *Exotic Option*. [online] **Investopedia**. Available at: https://www.investopedia.com/terms/e/exoticoption.asp [accessed 5 September 2021].

Mitchell, C., 2003. *Basket Option*. [online] **Investopedia**. Available at: https://www.investopedia.com/terms/b/basketoption.asp [accessed 5 September 2021].

tgood, 2017. *Geometric Brownian Motion Simulation in Python*. [online] **Stack Overflow**. Available at: https://stackoverflow.com/questions/45021301/geometric-brownian-motion-simulation-in-python [accessed 6 September 2021].

Jacob, B. and Guennebaud, G., 2006. *Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms*. [online] **Eigen**. Available at: https://eigen.tuxfamily.org/index.php?title=Main_Page [accessed 7 September 2021].

Marsaglia, G. and Tsang, W., 2000. *The Ziggurat Method for Generating Random Variables*. **Journal of Statistical Software**, [online] 5(8), pp.1-7. Available at: https://people.sc.fsu.edu/~jburkardt/cpp_src/ziggurat/ziggurat.html [accessed 7 September 2021].

Harris, M., 2017. *An Even Easier Introduction to CUDA*. [online] **NVIDIA Developer**. Available at: https://developer.nvidia.com/blog/even-easier-introduction-cuda/ [accessed 6 November 2021].

McKennon, J., 2013. *CUDA Parallel Thread Management*. [online] **Microway**. Available at: https://www.microway.com/hpc-tech-tips/cuda-parallel-thread-management/ [accessed 6 November 2021].

Harris, M., n.d. *Optimizing Parallel Reduction in CUDA*. [online] **NVIDIA Developer**. Available at: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf [accessed 6 November 2021].

Cohen, J., 2014. *Normal CUDA vs. CuBLAS?* [online] **Stack Overflow**. Available at: https://stackoverflow.com/questions/25836003/normal-cuda-vs-cublas [accessed 7 November 2021].

NVIDIA, 2021. *CUDA Toolkit Documentation - cuRAND*. [online] **NVIDIA Developer**. Available at: https://docs.nvidia.com/cuda/curand/index.html [accessed 7 November 2021].

NVIDIA, 2021. *CUDA Toolkit Documentation - cuSOLVER*. [online] **NVIDIA Developer**. Available at: https://docs.nvidia.com/cuda/cusolver/index.html [accessed 7 November 2021].

NVIDIA, 2021. *CUDA Toolkit Documentation - cuBLAS*. [online] **NVIDIA Developer**. Available at: https://docs.nvidia.com/cuda/cublas/index.html [accessed 7 November 2021].

NVIDIA, 2021. *CUDA Toolkit Documentation - thrust*. [online] **NVIDIA Developer**. Available at: https://docs.nvidia.com/cuda/thrust/index.html [accessed 7 November 2021].