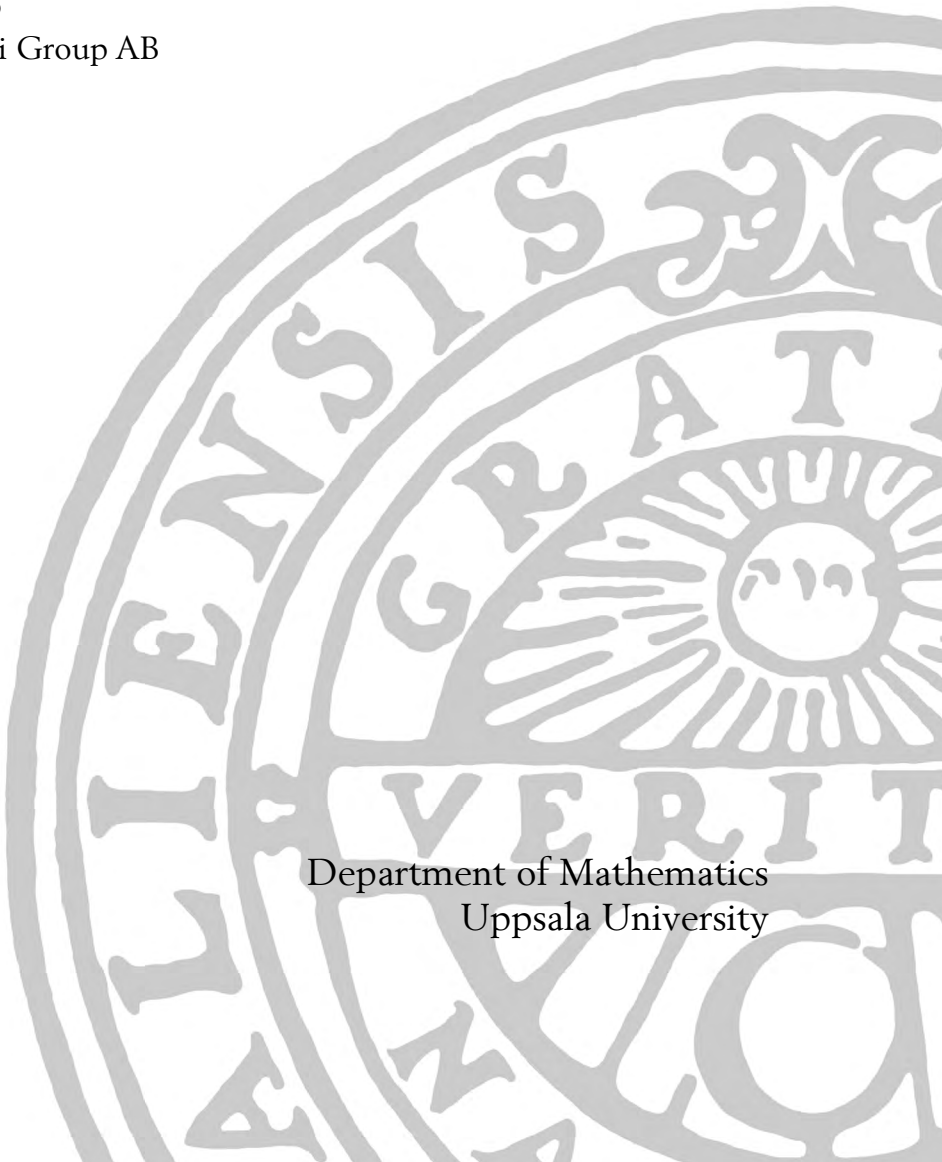
UPPSALA
UNIVERSITET

U.U.D.M. Project Report 2017:12

Asian option pricing using graphics processing units

Anna Shchekina

Examensarbete i matematik, 30 hp
Handledare: Jacob Lundgren, Itiviti Group AB
Ämnesgranskare: Erik Ekström
Examinator: Denis Gaidashev
Juni 2017



Department of Mathematics
Uppsala University

Asian option pricing using graphics processing units

Anna Shchekina
Uppsala University

April 2017

Abstract

The current paper explores a possibility of incorporating graphics processing units to pricing discrete Asian options. Theoretical foundations for mathematical and architectural parts of the project are presented. Ways of attaining instruction independence in the algorithm and limiting factors for parallelism are investigated. Results are compared with a sequential central processing unit implementation.

Contents

1. Introduction

Asian options are particularly challenging for those who strive to understand their value and behavior. Characterized by payoff that is dependent on the average price of the underlying asset over a pre-set timespan, they present themselves as a more stable alternative to the regular plain vanilla options. While most options are susceptible to temporary shocks – intentional or otherwise – in the vicinity of their maturity date, the smoothing characteristic of averages instead yields resilience to such movements. Unfortunately, for the common notion of average it is impossible to use the procedure by which analytical solutions for option prices are usually obtained, and the practitioner must turn to numerical methods.

Finite difference methods are very popular in computational finance, and are readily applicable in the solution of Asian option pricing problems. Due to the dependency of the option payoff on the average, however, the partial differential equation which yields the finite difference method is extended into an additional dimension. This causes a severe increase in the computational complexity of the algorithm, referred to as the curse of dimensionality. In order to arrive at results with both reasonable speed and accuracy, then, it is imperative that the solution algorithm is designed and implemented with performance in mind.

Advances in computer hardware architecture have brought the field to a state where performance for algorithms relying on sequential execution is plateauing, and focus is shifted to exposing algorithm parallelism. A clear sign of this development is the growth of parallel accelerator hardware architectures such as graphics processing units (GPUs), not only in the high performance computing sector but also in more general business areas. Making correct use of such massively parallel architectures requires paying special attention to algorithm dependency structures, and is certainly not a task to be taken lightly. In return, significantly increased computational resources and a relative adaptability to likely future improvements in computer hardware architecture are obtained.

Section 2 presents a brief overview of the option pricing theory and opens up the project with describing the underlying financial model and the examined instrument. The partial differential equation resulting from these considerations is further handled by section 3 which offers a numerical approach to solving the problem by means of finite differences. Architectural and programming aspects of the task are covered by section 4. Finally, results of a number of tests are presented and analysed in section 5.

2. Discrete Asian option

Financial derivatives are an essential tool in today's financial markets, with the derivative market amounting tremendous money turnover. A derivative is a contract defined in terms of a number of underlying assets. Such contracts serve a purpose of both protecting against the risk, termed hedging, and speculating. Since in the general run of things a higher possible return comes with a higher risk, investors who aim at maximizing their expected profit face a challenging problem of finding the correct balance on the risk versus return scale.

Theory of mathematical finance addresses the issue of what can be considered a "fair" price of a derivative instrument. It operates with the principle of no arbitrage which states that a strategy that requires no initial investment and carries no risk of incurring debt, yet has a positive probability of yielding profits in the future, cannot exist on the market. This seems intuitively compelling as such an opportunity if existed would immediately be exploited and investors would aim on taking a maximally long position in it. An arbitrage free market is always supplied with a risk-neutral or martingale probability measure. One can obtain the fair price of a derivative as a mathematical expectation under this measure.

The outline of the current section assumes that the reader is familiar with the basic terms and concepts in mathematical finance. More detailed information on the topic and the omitted proofs are presented in [1]. Rigorous theoretical foundations of derivative pricing including stochastic integration and calculus can be found in [2].

2.1. Model

Consider a market consisting of a risk-free asset, i.e. a bond, and a stock price following the stochastic differential equation (SDE) below:

$$\begin{cases} dS_t = \mu(t,\, S_t)S_t dt + \sigma(t,\, S_t)S_t dW_t, \\ dB_t = r(t,\, S_t)B_t dt \end{cases}$$

where $t \geq 0$ and W_t is a Brownian motion.

The *drift rate*

$$\mu(t,\, S_t) = r(t,\, S_t) - \delta(t,\, S_t)$$

is a combination of a continuously compounded *interest rate* and a *convenience yield* which can be used to model costs of carry or a *continuous dividend yield*.

In case the drift rate and the volatility functions are constants the stock dynamics

$$dS_t = \mu S_t dt + \sigma X_t dW_t$$

are said to be following a *geometric Brownian motion*.

Moreover, the stock pays *discrete dividends* of size D at times $\widehat{t}_i,\ i \in 1 : N_\mathrm{d}$.

2.2. Instrument

A *European call (put) option* with *strike* K and *maturity* T written on the underlying stock S is a contract giving its owner the right, but not the obligation, to buy (sell) the stock at time T at price K. Hence, at time T the payoff is:

$$\chi = (S_T - K)^+ \quad \text{or} \quad (K - S_T)^+$$

for a call or put option, respectively, where x^+ denotes $\max(x,\, 0)$.

Analogously, taking an average stock value on the whole interval $[0;\, T]$ instead of $S(T)$, one is left with a *continuous* or *discrete Asian option*. Letting the payoff value depend on a stock value during the whole contract interval instead of just picking the last moment secures the option holder against possible rapid changes in the stock price right before maturity, thus making the contract more stable.

The payoff of such contract in the discrete case becomes

$$\chi = (A_T - K)^+ \quad \text{or} \quad (K - A_T)^+$$

where A_t is an average of all stock values at fixing times between the the time of the first observation t_0 and the current time t:

$$A_t = \frac{1}{N_\mathrm{f}^{t_0,t}} \sum_{k=1}^{N_\mathrm{f}^{t_0,t}} S_{\widetilde{t}_k{}^-}\ , \tag{1}$$
$$\widetilde{t}_k \text{ — } \textit{fixing date}, \quad \widetilde{t}_k \in [t_0;\, t], \quad k \in 1 : N_\mathrm{f}^{t_0,t}$$

Here $N_\mathrm{f}^{t_0,t}$ is the number of fixing times on the interval $[t_0;\, t]$.

2.3. Price of the contract

Let us show that the price of a discrete Asian option with discrete dividends can be sequentially calculated by solving partial differential equations and applying the *non-arbitrage* condition and Definition 1 in order to find terminal conditions.

Recall from section 2.1 that on the intervals $(\widetilde{t}_i;\, \widehat{t}_j)$ between fixings and dividend payment times the stock follows

$$dS_t = \mu(t,\, S_t)S_t dt + \sigma(t,\, S_t)S_t dW_t$$

It can be shown that obtaining the solution to such SDE problem with a boundary condition at the end of the time interval is equivalent to solving a deterministic partial differential equation, which is widely known as a Black-Scholes equation.

Without loss of generality, let the last fixing time precede the last dividend time, i.e. $\widetilde{t}_{N_\mathrm{f}} < \widehat{t}_{N_\mathrm{d}}$. Consider the period $(\,\widehat{t}_{N_\mathrm{d}}\,;\,T)$. Then the price of the option can be found as the solution to the Black-Scholes equation:

$$\begin{cases} F_t(t,s,A_T) + \dfrac{1}{2}\,\sigma^2(t,s)s^2F_{ss}(t,s,A_T) + \big(r(t,s) - & (t,s) \in (\,\widehat{t}_{N_\mathrm{d}}\,;\,T) \times \mathbb{R}_0^+ \\ \delta(t,s)\big)s\,F_s(t,s,A_T)\; - r(t,s)F(t,s,A_T) = 0, & \\ F(T,s,A_T) = \phi(A_T) = \chi, & s \in \mathbb{R}_0^+ \end{cases}$$

It is important to acknowledge that the average A_T is a fixed parameter during the period considered, thus the number of variables in the equation above is identical to the one in the standard form of the Black-Scholes equation.

Let us examine what happens to the price at dividend payment times $\widehat{t}_{N_d}$. Since no arbitrage exists on the market:

$$S_{\widehat{t}^{\,+}_{N_\mathrm{d}}} = S_{\widehat{t}^{\,-}_{N_\mathrm{d}}} - D$$

Indeed, with $S_{\widehat{t}^{\,+}_{N_\mathrm{d}}} \neq S_{\widehat{t}^{\,-}_{N_\mathrm{d}}} - D$ one could buy the stock right before $\widehat{t}_{N_\mathrm{d}}$, receive the dividend and sell it for the same price right after $\widehat{t}_{N_\mathrm{d}}$ thus getting a risk-free profit.

With this observation and the continuity of the option price in time:

$$F\left(\widehat{t}^{\,-}_{N_\mathrm{d}},\, S_{\widehat{t}^{\,-}_{N_\mathrm{d}}},\, A_T\right) = F\left(\widehat{t}^{\,+}_{N_\mathrm{d}},\, S_{\widehat{t}^{\,+}_{N_\mathrm{d}}},\, A_T\right) = F\left(\widehat{t}^{\,+}_{N_\mathrm{d}},\, S_{\widehat{t}^{\,-}_{N_\mathrm{d}}} - D,\, A_T\right) \tag{2}$$

Without the continuity assumption an arbitrage possibility would be created whilist one could trade with the option itself.

Now consider the (i+1)-th fixing time. According to (1):

$$A_{\widetilde{t}_i{}^+} = \frac{1}{i+1}\sum_{k=0}^{i} S_{\widetilde{t}_k{}^-} \quad \text{and} \quad A_{\widetilde{t}_i{}^-} = \frac{1}{i}\sum_{k=0}^{i-1} S_{\widetilde{t}_k{}^-}$$

$$\Rightarrow\; A_{\widetilde{t}_i{}^+} = \frac{A_{\widetilde{t}_i{}^-}\, i + S_{\widetilde{t}_i{}^-}}{i+1}$$

Consequently, by continuity:

$$F\left(\widetilde{t}_i{}^-,\, S_{\widetilde{t}_i{}^-},\, A_{\widetilde{t}_i{}^-}\right) = F\left(\widetilde{t}_i{}^+,\, S_{\widetilde{t}_i{}^-},\, \frac{A_{\widetilde{t}_i{}^-}\, i + S_{\widetilde{t}_i{}^-}}{i+1}\right) \tag{3}$$

Analogously, for any period $(\widetilde{t}_i; \widehat{t}_j)$ without dividends or fixings, and all other interval combinations, one can find terminal conditions $\phi(a)$ for the Black-Sholes PDE.

This process can be seen as calculating a sequence of solutions $F^n(t, s, a)$ defined on the whole interval $t \in [\widetilde{t}_i; \widehat{t}_j]$ with

$$F^n(t,s,a) = \begin{cases} F(\widetilde{t}_i{}^+, s, a), & t = \widetilde{t}_i \\ F(t, s, a), & \widetilde{t}_i < t < \widehat{t}_j \\ F(\widehat{t}_j{}^-, s, a), & t = \widehat{t}_j \end{cases}$$

$$n = N_\mathrm{f} + N_\mathrm{d} - i - j + 2$$

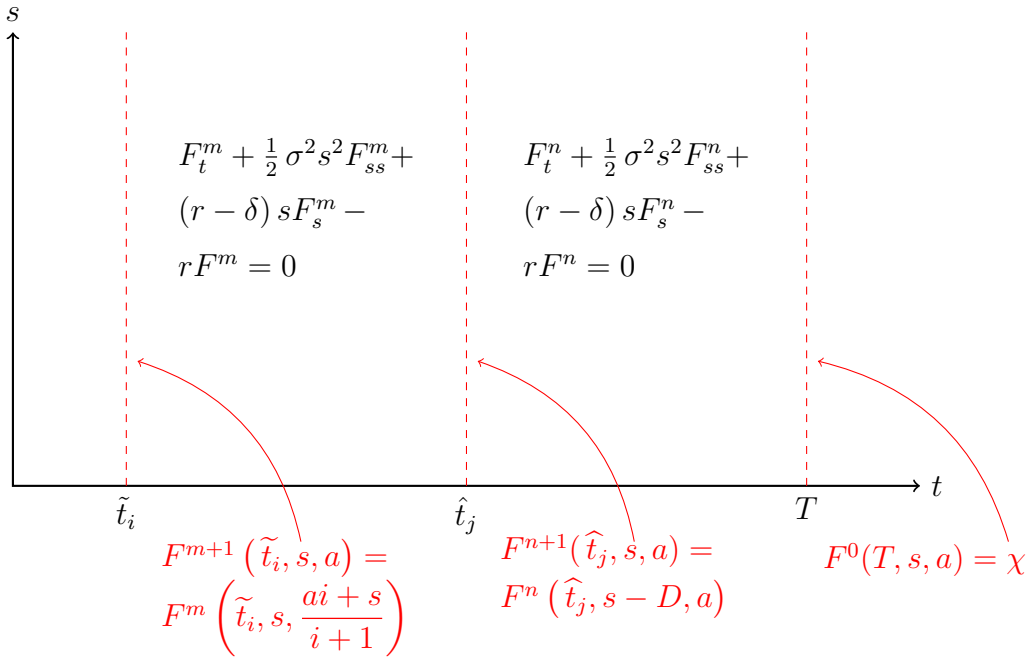


Fig. 1. Scheme for calculating the price

Note that if it happens that $\widetilde{t}_i = \widehat{t}_j$ for some i, j then the two conditions must be sequentially applied one after another in a predefined order. However, it is not a very feasible scenario, so we will not investigate this special case in further detail.

The above procedure is summarized by figure 1 where we look at the (i+1)-th fixing and the (j+1)-th dividend payment. There it is assumed that $m > n$ and arguments s and a stand for stock and average values at the current time point.

3. Numerical solution

It has been shown in the previous section that the procedure of obtaining the price of a discrete Asian option consists of successively solving boundary value problems for a partial differential equation and applying the jump conditions at dividend and fixing times.

There exist a number of ways to numerically solve the equation, among which probably the two most common are the *finite difference method* (FDM) and the *finite elements method* (FEM). Both supply the problem space with a discretization and result in a system of algebraic equations. The main idea of the FDM is approximating the differential equation with difference equations, in which finite differences approximate the derivatives. In turn, the FEM subdivides the problem into simpler parts that are called finite elements and uses variational methods to approximate a solution by minimizing an assembled error function.

Another alternative is utilizing the fact that the solution to the Black-Scholes equation is equivalent to a conditional expectation of a variable following a certain distribution. Thus, it can be approximated by simulating a distribution sample and calculating the expectation for this sample, which is widely known as a *Monte-Carlo method*.

For the current implementation it has been decided to only consider the finite difference method, in particular Crank-Nicolson and backward Euler schemes, as a solver for the PDEs. Futher in this section, the mentioned finite different methods are thoroughly examined and the systems of linear equations for them are derived. A method of solving the resulting systems then is described and it is shown what conditions guarantee its numerical stability. The section ends with a discussion on how to handle the jump conditions on an introduced discretization and estimating the overall accuracy of the numerical approximation. For a thorough analysis of many of the issues handled in this section, see e.g. [3].

3.1. Finite difference method

For solving the Black-Shcoles PDE obtained in section 2.3 we are going to use the *Crank-Nicolson* finite difference method. This type of finite difference scheme is accurate, unconditionally stable, but only for the terminal conditions that belong to $\mathcal{C}^2$ [4].

Since at $s = K$ there is a singularity point in the derivative of the put/call payoff function, Crank-Nicolson cannot guarantee stability. An alternative to this method is a less accurate but unconditionally stable for a wider class of functions *backward Euler* method. In order to avoid the instability issue it is a common practice to perform a few iterations of backward Euler just before $t = T$ [5]. Typically, 4 steps are taken.

After the time transform $\tau = T - t$ Black-Scholes equation becomes

$$F_\tau = \frac{1}{2}\sigma^2 s^2 F_{ss} + (r-\delta)\, sF_s - rF =: V(\tau, s, F, F_s, F_{ss}) \tag{4}$$
$$F(0,s,a) = F^a(0,s) = \phi(a),\ F_\tau(\tau, 0, a) = -rF$$

The boundary condition is readily obtained by substituting $s = 0$ into (4).

Limit the stock space from $\mathbb{R}_0^+$ to $[0;\, S_{\max}]$. According to [5] a reasonable choice of $S_{\max}$ and $A_{\max}$ is setting them to 4 strikes. A new boundary at $s = S_{\max}$ calls for proposing an additional boundary condition. This can be acquired considering an economically motivated observation of linear dependence of the price on the stock at large stock values:

$$F_{ss}(\tau, s, a) \xrightarrow[s\to\infty]{} 0 \quad \Rightarrow \quad F_{ss}(\tau, S_{\max}, a) \approx 0$$

Introduce time, stock and average space discretizations:

$$\tau_n = n\,\Delta\tau,\ n \in 0 : N_t - 1,\ \Delta\tau = \frac{T}{N_t - 1},$$
$$s_i = i\,\Delta s,\ i \in 0 : N_s - 1,\ \Delta s = \frac{S_{\max}}{N_s - 1}$$
$$a_j = j\,\Delta a,\ j \in 0 : N_a - 1,\ \Delta a = \frac{A_{\max}}{N_a - 1}$$

Define

$$F_i^n = F(\tau_n,\, s_i)$$

The considered methods are described by the following schemes:

$$\frac{F_i^{n+1} - F_i^n}{\Delta\tau} = V_i^{n+1} \quad \text{backward Euler, implicit}$$
$$\frac{F_i^{n+1} - F_i^n}{\Delta\tau} = \frac{V_i^{n+1} + V_i^n}{2} \quad \text{Crank-Nicolson, explicit}$$

Let us reduce the backward Euler case to a system of linear equations. After approximating the stock space derivatives:

$$\frac{F_i^{n+1} - F_i^n}{\Delta\tau} = \frac{1}{2}\sigma^2 (i\Delta s)^2 \frac{F_{i+1}^{n+1} - 2F_i^{n+1} + F_{i-1}^{n+1}}{(\Delta s)^2} + (r-\delta)\, i\Delta s \frac{F_{i+1}^{n+1} - F_{i-1}^{n+1}}{2\Delta s} - rF_i^{n+1} \tag{5}$$

The initial and boundary conditions become

$$F_i^0 = \phi(a), \quad i \in 0 : N_s$$

$$F_0^n = (1 + r\Delta\tau)\, F_0^{n+1}, \quad n \in 1 : N_t \tag{6}$$

$$F_{N_s}^n = 2F_{N_s-1}^n - F_{N_s-2}^n, \quad n \in 0 : N_t \tag{7}$$

Reorganise (5) into

$$F_i^{n+1} - \frac{\Delta\tau}{2}\big(\underbrace{\left[\sigma^2 i^2 - (r-\delta)\, i\right]}_{\gamma_i} F_{i-1}^{n+1} + \underbrace{\left[-2\sigma^2 i^2 - 2r\right]}_{\alpha_i} F_i^{n+1} +$$

$$\underbrace{\left[\sigma^2 i^2 + (r-\delta)\, i\right]}_{\beta_i} F_{i+1}^{n+1}\big) = F_i^n, \quad i \in 1 : N_s - 2$$

For $i = 0$ using (6) define

$$\xi = -2r$$

For $i = N_s - 1$ using (7):

$$F_{N_s-1}^{n+1} - \frac{\Delta\tau}{2}\big(\underbrace{\left[\gamma_{N_s-1} - \beta_{N_s-1}\right]}_{\theta} F_{N_s-2}^{n+1} + \underbrace{\left[\alpha_{N_s-1} + 2\beta_{N_s-1}\right]}_{\omega} F_{N_s-1}^{n+1}\big) = F_{N_s-1}^n$$

Finally, the above can be rewritten as

$$(I - B_{n+1})\, F^{n+1} = F^n \tag{8}$$

$$B_n = \frac{\Delta\tau}{2}\begin{bmatrix} \xi & 0 & & \\ \gamma_1 & \alpha_1 & \beta_1 & \\ \ddots & \ddots & \ddots & 0 \\ & \gamma_{N_s-2} & \alpha_{N_s-2} & \beta_{N_s-2} \\ 0 & & \theta & \omega \end{bmatrix}, \quad F^n = \begin{bmatrix} F_0^n \\ \vdots \\ F_{N_s-1}^n \end{bmatrix}$$

Coefficients $r(\cdot, i\Delta s),\ \sigma(\cdot, i\Delta s),\ i \in 0 : N_s - 1$ for B_n are calculated at $n\Delta\tau$.
Analogously, for Crank-Nicolson scheme:

$$\underbrace{(I - A_{n+1})}_{A_{\text{LHS}}} F^{n+1} = \underbrace{(I + A_n)}_{A_{\text{RHS}}} F^n \tag{9}$$

where

$$A_n = \frac{B_n}{2} + \left(\frac{r\Delta\tau}{2} - \frac{r}{2 + r\Delta\tau} \right) J^{00} \tag{10}$$

Note that for the Black-Scholes model, where the market functions are fixed, i.e.:

$$\sigma(t,s) \equiv \sigma,\ r(t,s) \equiv r,\ q(t,s) \equiv q$$

it is found that

$$A_{\text{LHS}} = -A_{\text{RHS}} + 2I$$

This observation is particularly useful for implementation and performance. As for the situation of non-constant functions, if the number of nodes is big enough and the function is smooth, it can still be considered that $A_{n+1} \approx A_n$. Supposing constant market parameters σ, r and q matrices A_{LHS} and A_{RHS} are unchanging during the whole process of obtaining a solution. If one of the functions is dependent on time or stock, at each time step the matrices get recalculated taking that the change is significant.

Substituting the exact solution in 5, taking a difference between A_{LHS} and A_{RHS} and Taylor expanding one gets the *truncation error* of the methods. For Crank-Nicolson scheme the truncation error is $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta s^2)$ plus cross terms. For Implicit Euler accuracy in time is smaller and is $\mathcal{O}(\Delta t)$.

3.2. Thomas algorithm

Since (8) and (9) are tridiagonal systems of equations, one can apply the popular tridiagonal matrix algorithm, also known as the Thomas algorithm, to solve them. A more detailed description of the method can be found, for example, in [6].

Consider a system of linear equations:

$$\begin{bmatrix} b_0 & c_0 & & \\ a_1 & b_1 & c_1 & \\ \ddots & \ddots & \ddots & \\ & \ddots & a_{N_s-1} & b_{N_s-1} \end{bmatrix} \begin{bmatrix} F_0^{n+1} \\ F_1^{n+1} \\ \vdots \\ F_{N_s-1}^{n+1} \end{bmatrix} = \begin{bmatrix} d_0^n \\ d_1^n \\ \vdots \\ d_{N_s-1}^n \end{bmatrix} \tag{11}$$

The procedure to be applied to (11) constitutes a special case of the standard Gaussian elimination algorithm for solving systems of linear equations. The main idea is to perform a so called forward sweep which will eliminate the subdiagonal of the matrix and thus transform

the matrix into an upper triangular one:

$$\begin{bmatrix} 1 & \widetilde{c}_0 & & \\ 0 & 1 & \widetilde{c}_1 & \\ \ddots & \ddots & \ddots & 0 \\ & \ddots & \ddots & \widetilde{c}_{N_s-2} \\ 0 & & 0 & 1 \end{bmatrix} \begin{bmatrix} F_0^{n+1} \\ F_1^{n+1} \\ \vdots \\ F_{N_s-1}^{n+1} \end{bmatrix} = \begin{bmatrix} \widetilde{d}_0^n \\ \widetilde{d}_1^n \\ \vdots \\ \widetilde{d}_{N_s-1}^n \end{bmatrix} \tag{12}$$

by setting

$$\widetilde{c}_i = \begin{cases} \dfrac{c_i}{b_i}, & i = 0 \\ \dfrac{c_i}{b_i - a_i \widetilde{c}_{i-1}}, & i \in 1 : N_s - 2 \end{cases} \qquad \widetilde{d}_i = \begin{cases} \dfrac{d_i}{b_i}, & i = 0 \\ \dfrac{d_i - a_i \widetilde{d}_{i-1}}{b_i - a_i \widetilde{c}_{i-1}}, & i \in 1 : N_s - 1 \end{cases} \tag{13}$$

so the solution of (12) is obtained by performing a backward substitution.

3.2.1. Stability

If some of the denominators in (13) are zero or numerically zero the method becomes unstable. A condition that is sufficient but not necessary for stability of tridiagonal algorithm is the left hand side matrix in (9) being diagonally dominant [7]. For Crank-Nicolson and Implicit Euler this is equivalent to two conditions:

$$|\frac{4}{\Delta\tau} + 2\sigma^2 i^2 + 2r| \geq |(r-\delta)i - \sigma^2 i^2| + |-(r-\delta)i - \sigma^2 i^2|, \quad i \in 1 : N_s - 2 \tag{14}$$

$$\frac{2}{\Delta\tau}|r-\delta| \geq |r - \delta - \frac{r}{N_s - 1}| \tag{15}$$

The first condition represents 2 possibilities.

Either $\sigma^2 \geq |r-\delta|$ which implies $\sigma^2 i^2 \geq |r-\delta| i$ for all i and then

$$\frac{2}{\Delta\tau} + \sigma^2 i^2 + r \geq \sigma^2 i^2$$

which always holds. Otherwise, with $\sigma^2 < |r-\delta|$

$$\frac{2}{\Delta\tau} + \sigma^2 i^2 + r \geq |r-\delta| i \tag{16}$$

Consider a real valued function:

$$f(x) = -\sigma^2 x^2 + |r - \delta| x - r$$

Its maximum is attained at

$$x^* = \frac{|r - \delta|}{2\sigma^2}$$

with

$$f(x^*) = \frac{|r - \delta|^2}{4\sigma^2} - r$$

Hence, taking

$$\frac{2}{\Delta\tau} \geq f(x^*) \quad \Leftrightarrow \quad \Delta\tau \leq \frac{2}{\frac{|r-\delta|^2}{4\sigma^2} - r} \tag{17}$$

assures fulfilment of (16).

To sum up, for (14) to hold either volatility must be large enough or time discretization must be refined in time such that it satisfies (17).

As for condition (15), one can use the triangle inequality to claim that it follows from

$$\frac{2}{\Delta\tau}|r - \delta| \geq |r - \delta| + \frac{r}{N_s - 1}$$

which is equivalent to

$$\Delta\tau \leq \frac{2\,|r - \delta|\,(N_s - 1)}{|r - \delta|\,(N_s - 1) + r} \tag{18}$$

Since normally N_s is big enough to guarantee $r << |r - \delta|\,(N_s - 1)$ condition (18) is automatically fulfilled. Otherwise, the time step must be set according to (18).

Thus, it has been shown that it is always possible to choose such Δt that stability for the Thomas algorithm is ensured.

3.3. Jumps

After solving the SLE (9) on time intervals free from dividends and fixings, one must transfer the information about the solution from the previous finite difference step to the terminal condition of the current step. There are two directions of the data transfer — between the stock levels for each finite difference solution path with a fixed average; and between different averaging levels.

The transfer happens through equations (2) and (3) derived in section. Observe that the points from Figure 1 with stock and average being equal to

$$s = s_i - D, \quad i \in N_{\mathbf{f}} \quad \text{and} \quad a = \frac{a_j\, n + s_i}{n+1}, \quad j \in N_{\mathbf{d}}$$

do not necessarily lie on the initially established grid. To handle this one must interpolate the price between the values for which the solution was calculated after one of the finite difference method steps. One can, for example, choose a linear interpolation or a more accurate four-point Lagrange interpolation.

These transfer requirements can be viewed as a hindering factor for independence between the averaging levels. Consequently, they create an obstacle for parallelizing an otherwise fully independent procedure of solving for the option pcice between averaging levels. Thus, it induces a need for synchronization at jump points which refer to average recalculation. As for the other type of jumps, they do not make things more complicated as the sequentiality inside each average level is inevitable meaning that there is no way to solve the partial differential equation at a given step without obtaining the initial conditions for the system from the previous step.

3.4. Accuracy

The overall error of the scheme consists of the following components:

- finite difference method error
- interpolation error
- error incurred by discretizing average dimension
- error incurred by truncating stock space
- cross terms

Finite difference method error was discussed in section 3.1. Since less accurate in time Implicit Euler method is used at the time step just before maturity, the accuracy can potentially fall to $\mathcal{O}(\Delta t)$. If one wants to keep the overall order of accuracy for the sequence of finite difference instances equal to 2 one must compensate for the higher error by using smaller time steps. The last interval must be split into Δt parts, then Implicit Euler will have the same accuracy as Crank-Nicolson. When used for practical purposes, the program is normally run for a rather small number of time steps. Thus, a factor 4 is commonly employed instead of Δt and the resulting reduction in accuracy is seen as minor.

Interpolation error for linear interpolation that takes place at the ends of stock intervals has the order of accuracy 2 which is big enough to not increase the overall error. 4 point

Lagrange interpolation algorithm with order of accuracy 4 is used for inner points.

Experiments show that for a regular range of volatilities and the considered problem the optimal truncation level is around 4 strike values. This includes accounting for trade-off resulting from loosing accuracy for boundary conditions and increasing it by reduced stock step. We are expecting this choice of boundary to result in a relatively small error.

Bringing up rear, the expected errors of price in stock and time are of order 2. Average dimension error is something that is harder to make theoretical predictions about so we will base our knowledge of its order on practical results. The errors in all dimensions are further explored in the convergence section.

4. GPU

Historically, occurrence of the graphics processing units in their current state on the market was solely dictated by an ever increasing demand for hardware that had the ability to handle sophisticated computer graphics. Correspondingly, GPUs are designed in accordance with the inherently massively parallel nature of graphics calculations as their main target. However, is has been eventually discovered that in many occasions it appears to be highly beneficial to use the GPU for purposes other than graphics and image processing.

The principal factor distinguishing the GPU approach from the more standard central processing unit approach is the focus on *throughput* rather than *latency*. Sacrificing faster clock speed for each parallel processor in the GPU is compensated by the enormous amount of these simple processors. GPUs thus excel at tasks that do not call for mathematically complex operations and at the same time can offer a massive amount of parallel instructions.

Before CUDA was first introduced in 2006 as a C/C++ extension it was only possible to utilize GPUs doing the native shader programing which was laborious and not adjusted for the needs of *general purpose GPU* programming (GPGPU). Since the GPGPU area is rather new, there are not many developed platform models. Among others, that include OpenCL, CUDA is the most mature and well-supported. Taking this into consideration, it was chosen to implement the GPU parts of the code using NVIDIA CUDA language.

In this section some relevant architecture details of the graphics processing units are covered. Considerations on the applicability of the GPU to the Asian option pricing algorithm are presented. An example of a GPU kernel is provided. An interested reader can find a more thorough description of the GPU specifics and best programming practices in [8].

4.1. Motivation

4.1.1. General merits

The GPU endows a reasonable trade-off between speed and power increase in comparison with the CPU. Speaking of performance, the GPU provides 11 times speedup, which is measured in peak GFLOPS, i.e. billions of floating-point operations per second. Slower clock speed, characteristic for GPUs, means less power consumption for the same amount of calculation. Thus, the GPU consumes as little as two times more power, which makes it overall $11/2 \approx 5$ times more efficient than the CPU. However, for double precision GPU performance is not as impressive and constitutes only up to 7 times speedup factor. That being said, the numbers mentioned above solely apply to a graphics oriented program; for a GPGPU program an expected result would normally be a 2–5 times better performance.

Another advantageous feature of the GPU is its *scalability*. Days have passed when achieving an enhanced performance by increasing clock speed or developing smarter controls was a painless affair. Nowadays, adding more cores to a processor represents a much cheaper alternative, which corresponds fittingly with the concept of a multiprocessor.

It is easy to see that taking its vast number of cores combined with their relative weakness the GPU excels at regular math-intensive work. It is important to keep in mind that to achieve the highest possible efficiency, however, a program must combine interleaving usage of both types of processing units. This is often the case for *heterogeneous* systems which use specialised hardware to adjust for the characteristics of a specific task.

4.1.2. Suitability for the algorithm

Algorithms that can benefit from a GPU implementation are those providing a massive amount of parallel instructions at a time point. If a program does not have enough parallelism to fill up the large number of GPU processor units then usually a proper CPU application will overbid all the attempts to adjust the algorithm for utilization of the GPU. Costs incurred by slow memory accesses and absence of comprehensive control unit as well as a higher programming complexity will in this instance outweigh the parallelization advantages.

At first glance the considered algorithm seem to be reasonably amenable to parallelization. Since average is fixed as a parameter between fixing times running finite difference methods for each average in the discretization is entirely independent from all other averaging levels. This holds everywhere but at fixing correction times when the information transfer from one level to another occurs, recall section 3.3. Moreover, the same left hand side matrices for the system of linear equations are used between average levels which could be employed for data reuse between threads. This possibility is explored in section 4.5.

It is harder to find instruction level parallelism in stock and time dimensions, however, as the procedure of solving the Black-Scholes equation includes a matrix inversion which is inherently sequential. As a compromise, one could find independent instructions in matrix-vector multiplications involved in calculating the right hand side of the equation.

In addition to that, the procedure of handling jumps which consists of linear and Lagrange interpolation is easily parallelizable between averages just as is the initialization of the solver.

When a trader is interested in a price of a particular option it is often the case that he or she wants to know the price for a number of different strikes. Since the algorithm described is completely independent for different strike prices it instantly follows that adding a strike dimension to the implementation could easily contribute to maximizing utilization of the GPU. However, one must keep in mind that for different levels of K not only the terminal condition will change but also step sizes in stock and average dimensions. This is due to

the fact that the step sizes being calculated as the assumed maximum stock boundary $4K$ divided by the number of stock and average nodes. Having varying step sizes is limiting because finite difference matrices cannot be reused between strike levels.

Conclusively, it does not take long until the number of averaging levels combined with strike levels becomes large enough to pile up the GPU and at the same time make it harder for the CPU to compete with it. Possible adjustments to the algorithm that can potentially increase the amount of parallel parts in the code are further discussed in section 5.3.

4.2. Design

In order to get hold of efficient GPU programming, an understanding of how hardware is built is crucial. A number of design considerations will be handled in the current section. For a more detailed information one might want to consult NVIDIA guides [9, 10], specifically, documentation references for GeForce GTX 1070 used for the project.

The fundamental unit in the GPU is a CUDA core. A set of such cores are combined to build a *streaming multiprocessor* (SM) which is analogous to a core in the CPU. GeForce GTX 1070, for example, is supplied with 15 streaming multiprocessors consisting of 128 processors each which makes a total of 1920 CUDA cores.

In a heterogeneous computer that has both the CPU and the GPU they are commonly denoted as *host* and *device*. If the GPU is embedded on the same chip where the CPU resides they share the same memory and do not experience the problem of high host-device traffic latency. In the case of a discrete device, it is supplied with its own *device memory*.

Figure 2 schematically illustrates the structures of the two processing units. Instead of having a few massive high performance optimised cores with strong *arithmetic logical units* (ALUs) the GPU is supplied with a huge number of weaker processors. Diminished versions of control unit and shared caches are shared between the chunks of SPs. Each core of a CPU has a hierarchy of caches at hand and a comprehensive control unit supporting out-of-order execution and speculation. At the same time an SM consists of 128 ALU-analogous processors which share a small amount of local memory consisting of *shared memory*, L1 data cache and instruction cache. All cores in a SM can only work on the same instruction at a time. Communication and synchronization within a SM is cheap and is done by means of shared memory. Shared memory accesses are characterised by a very small latency, but the memory itself is limited in size, constituting 96KB for the considered device.

Parallelizm on the GPU is accomplished by launching a vast number of threads that execute independent parts of the code. The maximum amount of threads that can be launched per SM is 2048, which means each accounts for as little as 96KB/2048 = 48B of shared

memory. Threads are grouped into blocks and blocks, in turn, are organised in a grid. Each block is automatically assigned to a single SM by the hardware. Despite it might seem that all threads in a block run together, not all of them, in fact, are executed concurrently. Subdivisions of a block that actually run simultaneously are called *warps*. A warp typically consists of 32 treads which are executed in a *single instruction multiple data* (SIMD) fashion. If an instruction follows a conditional statement then the threads which shall not enter the statement make the calculation nonetheless but then do not provide the result back. Warps are handled by the GPU thread scheduler. A large number of active warps in a multiprocessor would serve an indicator that the GPU resources are utilized efficiently.

Different are the ways that the GPU and the CPU handle instruction stalls. CPUs are good at reducing memory latencies with use of integrated tools like branch predictor which are included in its sophisticated control unit. In turn, GPUs try to cover latencies by switching to another thread if waiting for memory for the current thread takes too long. Thus, if there are enough threads to pile up the GPU one gets a good throughput.

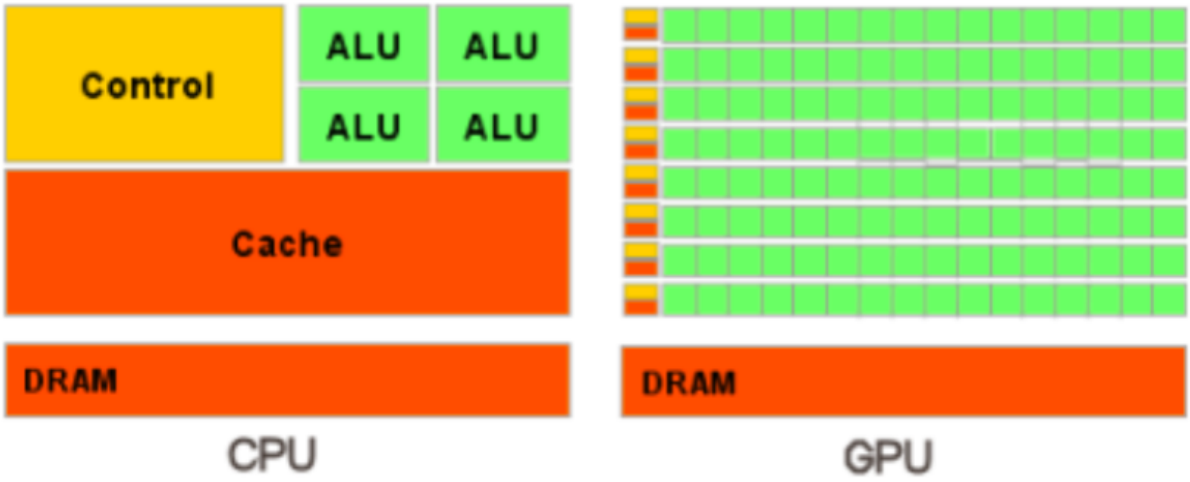


Fig. 2. CPU and GPU architectures

4.3. Downsides

Moving data between the GPU and the CPU is a bottleneck. The GPU has a very good bandwidth with its own memory, which is, in fact, better than that for the CPU. However, data transfer between the two processing units is extremely slow.

Besides, global device memory itself is not as big as the host memory. GDRAM assigned to the GPU constitutes 8GB whereas DRAM connected to CPU can fit around 5 times more data. This observation is limiting for the algorithm as the size of the price vectors for all the different levels in the discretization should be bounded by 8GB.

To take advantage of bandwidth for the GPU one should employ spacial and temporal locality to use data loaded together as a cache line. On the CPU instruction stalls will be mainly eliminated by hardware prefetching which is a part of its sophisticated control unit absent for the GPU. Because of the limited amount of data cache per SM to prevent latency and stalls threads inside a warp should work with data from the same memory read which is called *memory access coalescing*.

Another issue that can significantly limit performance is so-called *divergent execution*. Since the instruction cache only fetches one instruction at a time to be executed by all the threads inside a warp, if it finds conditional statements then taken that some percent of threads do not go inside a branch it will just stall, more precisely, do useless work. Thus, numerous paths of execution will create a huge overload which can have a dramatic effect on performance. A solution might be putting threads which are likely to follow the same path in the same warp combined with memory access coalescing.

4.4. Example of a GPU kernel

In a CUDA program there exist 3 types of functions:

- *kernel* functions that are run on the device and launched from the host to start new threads, identified with `__global__` keyword
- device functions that are run on the device and launched from a kernel or another device function, identified with `__device__`
- host functions that are run on the host and launched from the host, identified with `__host__`

Historically, 1 kernel at a time was allowed. Nowadays there is also a possibility of starting new kernels from the device, which would be denoted as *dynamic parallelism*.

For convenience of the programmer blocks and grids are 3 dimensional. After a kernel has been launched the way to distinguish between the threads is calculating a unique 3

dimensional index `(ix, iy, iz)` for each one based on its tread number `threadIdx`, block number `blockIdx` and size of blocks `blockDim`. Such index in a specific dimension is the thread index in the current dimention plus the offset. If one of the dimensions is to be skipped for a particular algorithm then the corresponding component will be fixed at 1. Figure 3 demonstrates the constitution of a block of threads launched by a GPU kernel.

In this section construction of a simple CUDA kernel is illustrated by a dividend correction procedure imposed by the condition (2) from the section 2.3. Let us first observe this function as it would be regularly implemented on the CPU:

```
__host__ void HandleDividend(
    const double * values,
    double * values_buffer,
    const double dividend,
    const double stock_step,
    const int s_num_nodes,
    const int a_num_nodes,
    const int num_strikes)
{
    for(int s_idx = 0; s_idx < s_num_nodes; s_idx++)
    {
        const double curr_stock = s_idx * stock_step;
        const double new_stock =
            fmax(0., curr_stock - dividend);

       for(int a_idx = 0; a_idx < a_num_nodes; a_idx++)
        {
            for(int k_idx = 0; k_idx < num_strikes; k_idx++)
            {
                const int offset = k_idx * a_num_nodes * s_num_nodes +
                    a_idx * s_num_nodes;
                const double new_value = fmax(0.,
                    InterpolateStock(new_stock, offset, values));

                const int curr_ind = offset + s_idx;
                values_buffer[curr_ind] = new_value;
            }
        } } }
```

All the option prices for different levels of stock, average and strike are aligned in memory pointed by `values`. Here `InterpolateStock` is a function performing 4 point Lagrange interpolation in a point `new_stock` using the corresponding part of `values`.

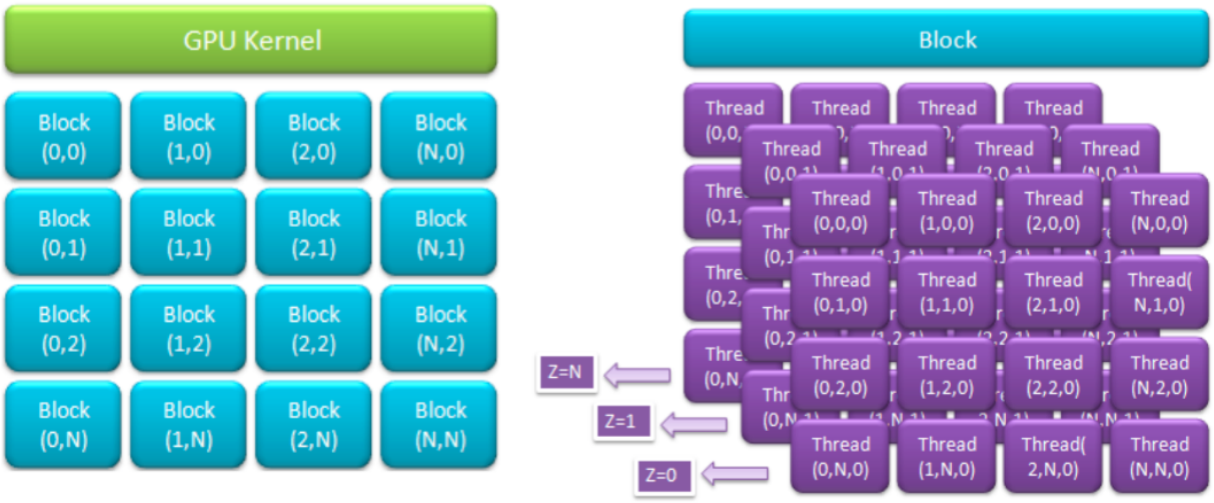


Fig. 3. Thread configuration

Now instead of the 3 loops introduce a 3 dimensional grid of threads and rewrite the function to be suitable for the GPU.

```
__global__ void HandleDividendGPU(
    const double * values,
    double * values_buffer,
    const double dividend,
    const double stock_step,
    const int s_num_nodes,
    const int a_num_nodes,
    const int num_strikes)
{
    const int ix = blockIdx.x * blockDim.x + threadIdx.x, // S
        iy = blockIdx.y * blockDim.y + threadIdx.y, // A
        iz = blockIdx.z * blockDim.z + threadIdx.z; // K

    if (ix < s_num_nodes && iy < a_num_nodes && iz < num_strikes)
    {
        const double curr_stock = ix * stock_step;
```

```
        const double new_stock =
            fmax(0., curr_stock - dividend);

        const int offset = iz * a_num_nodes * s_num_nodes +
                    iy * s_num_nodes;

        const double new_value = fmax(0.,
            InterpolateStock(new_stock, offset, values));

        const int curr_ind = offset + ix;
        values_buffer[curr_ind] = new_value;
    }
}
```

Stock, average and strike dimensions are represented by `X`, `Y` and `Z` planes respectively.

Somewhere in the host code kernel `HandleDividendGPU` must be invoked as:

```
HandleDividendGPU<<<dimGrid, dimBlock>>>(args);
```

`dimGrid` and `dimBlock` are configuration parameters represented by a `dim3` structure.

```
const dim3 dimBlock(TILE_WIDTH_3D, TILE_WIDTH_3D, TILE_WIDTH_3D),
    dimGrid((s_num_nodes - 1) / TILE_WIDTH_3D + 1,
    (a_num_nodes - 1) / TILE_WIDTH_3D + 1,
    (num_strikes - 1) / TILE_WIDTH_3D + 1);
```

The maximum number of threads per block is 1024, and the optimal block size is either 512 or 256. The constant `TILE_WIDTH_3D` is equal to 8 so that the block size becomes 512. Then grid size is set to fit all the nodes in the 3 dimensions. Constructed this way, the last block in the grid will contain extra nodes which need to be sorted out inside the kernel using conditional statements.

`InterpolateStock` function can remain unchanged taken that `__device__` is added before the declaration. It is also possible that the function can be still used as previously if `__host__ __device__` keyword precedes the name.

4.5. Shared memory

One can recall from section 4.2 that there is room for threads which belong to one block to load and store data to a special piece of memory that is assigned to this block and is shared between all the threads in it. Doing so allows to reduce access times significantly compared with accessing the global device memory. In order to make an efficient use of shared memory one must utilize data locality of the algorithm in a smart way.

Since there is a vast amount of data reuse between both average and strike levels, namely the left hand side matrices for the system (9) do not depend on any grid parameter other than those for time grid and so are identical for all the levels, that seems natural to only calculate them once and then let the solvers for each particular level have access to this precalculated value. Hence, matrices could be placed in the shared memory.

The only case when equivalence of the system between levels is violated is when market parameters r, σ and q are dependent on the stock value. Then each line of the matrix will be calculated using a new node on the stock grid which differs between the strike levels. As a result, one will only be able to reuse data between the average but not the strike levels.

Motivating factors for these kind of optimization are:

- lower latency when accessing data from the shared memory
- a rather high cost of constructing these matrices
- avoiding wasteful usage of the global GPU memory, where the replicated matrix is stored $N_K * N_a$ times

However, there are a few downsides of this approach. Since the number of threads in a block is bounded by 512 just as the size of the shared memory assigned for a block is bounded by 64KB, there exists a limit on a size of thread groups for which one can take advantage of precalculating the common data.

The matrix to be stored is a 3-diagonal matrix of doubles, which means that the real amount of data that needs to be stored has order of $3 * N_s$. Thus, taken that the shared memory can store 8192 doubles, one can conclude that the size of matrix that fits there can be no bigger than $N_s = 2730$ which is a size that can grant a good accuracy of the result.

All things considered, shared memory utilization will not be included in the current implementation of the algorithm taken that the number of nodes used for testing will be as high as $N_s = 4000$. Still, this can potentially be beneficial to performance especially if one comes up with more advanced optimization techniques.

5. Results

In the current section results of the implemented program will be presented. Furthermore, detailed descriptions of the test performed for it will be given.

At first, the results generated by the code are compared with the results of a CPU implementation of the algorithm which used Curran approximation to prove correctness.

The system used for the implementation and testing has the following characteristics:

- Intel Xeon CPU W3520 @ 2.67GHz
- GPU GeForce GTX 1070
- Microsoft Windows 7 Enterprise
- Microsoft Visual Studio 2013 compiler

Market framework specified for the tests:

- $K = 100$
- $T = 1$
- $\sigma = 0.4$
- $r = 0.05$
- $\delta = 0.02$

Observe that these parameters satisfy $\sigma^2 \geq |r - \delta|$ and $r << |r - \delta|\,(N_s - 1)$ which guarantees stability for the Thomas algorithm (recall section 3.2.1).

5.1. Convergence tests

The aim of the *convergence* test performed for the algorithm implementation is to verify that with an increased density of the discretization the difference in the results it produces tends to zero. The discretization mentioned is the one introduced in section 3.1. Formally speaking:

$$F_{N_t,\,N_s,\,N_k} \to F^{\mathbf{theoretical}} \quad \text{with} \quad N_t,\, N_s,\, N_k \to \infty \tag{19}$$

$$\Leftrightarrow \quad \text{with} \quad \Delta t,\, \Delta s,\, \Delta k \to 0$$

One way of verifying (19) is to check that the condition holds for each of the three dimensions of the discretization separately while keeping the number of nodes in the other two dimensions fixed at a certain level which ideally should be an infinity. Since one cannot acquire a theoretical reference solution for the considered problem, it can be a good idea to choose a set of sufficiently big $N_t^{\mathbf{ref}}$, $N_s^{\mathbf{ref}}$, $N_k^{\mathbf{ref}}$ and thus use a value $F_{N_t^{\mathbf{ref}},\,N_s^{\mathbf{ref}},\,N_k^{\mathbf{ref}}}$ as

a self-reference for all the three convergence tests. To produce convergence graphs (5)–(6) number of nodes for reference was chosen to be $N_s^{\mathbf{ref}} = 8001$, $N_t^{\mathbf{ref}} = 2000$, $N_k^{\mathbf{ref}} = 4001$.

To provide a higher plausibility instead of just looking at one initial stock value pick a set of 40 equidistant stock nodes in the interval $[\frac{3K}{4}; \frac{5K}{4}]$ and calculate the corresponding option prices. The resulting vector will be used as a reference and stock nodes with the highest deviation will be considered the errors.

Each of the three tests the intervals $[100; N_i^{\mathbf{ref}}]$, $i \in \{t, s, k\}$ are divided into 20 equal parts to assure the smoothness of the resulting graphs to be of the same order. Then for each of the 21 resulting nodes new average or stock and time grid settings are constructed and a solver is run. Finally, a relative error to the reference solution is being produced. Number of nodes in the two dimensions that remain constant are taken equal to the corresponding reference numbers. Below 100 nodes the accuracy of the method is not good enough, so results below that point are not displayed on the graphs.

Note that the grid settings vary between strike levels as well because strike value defines the upper limit of stock and average intervals. Moreover, for multiple strikes only the maximum relative error is tracked for each number of levels.

Figures (5)–(6) are produced in logarithmic scale so that they would naturally show the rate of convergence.

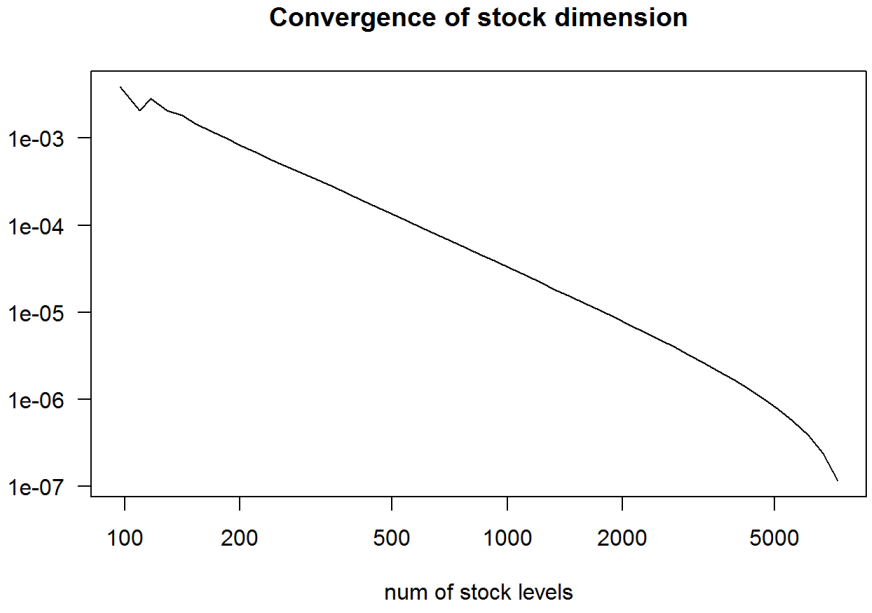


Fig. 4. Stock convergence rate

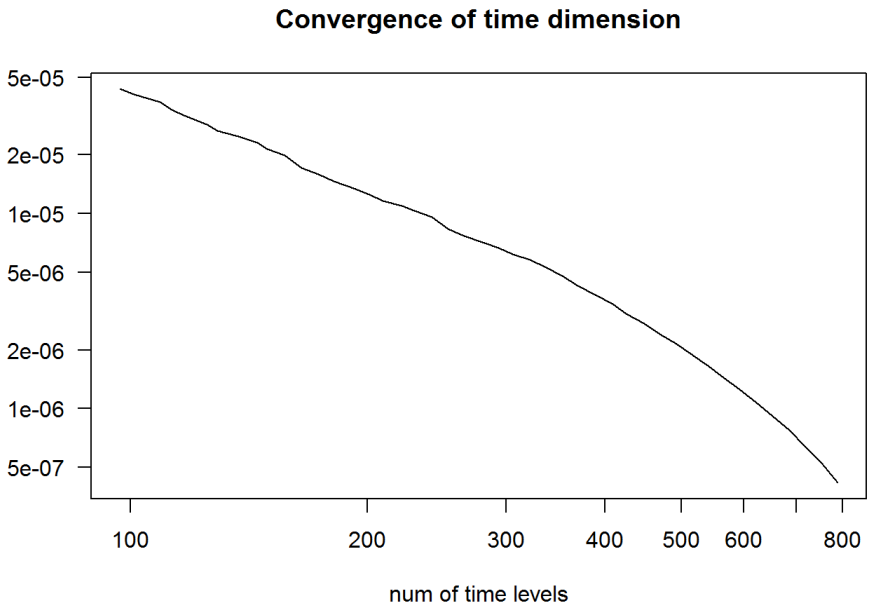


Fig. 5. Time convergence rate

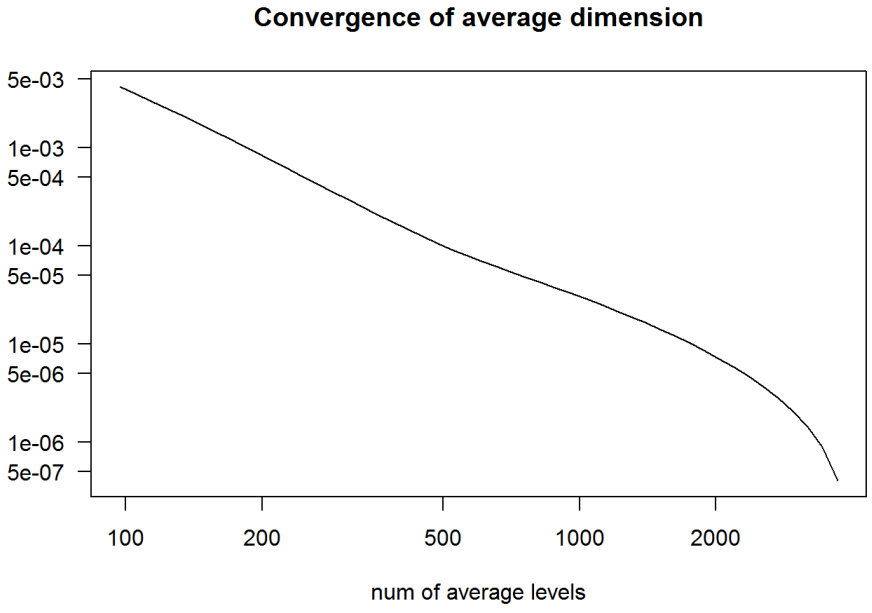


Fig. 6. Average convergence rate

When the program is run for a group of options with various strikes, a sequence of strike values with a step $\Delta K = 2$ starting at $K_0 = 0$ is chosen. There is a support for non-constant volatility, interest rate and convenience rate which are implemented as piecewise linear functions of time. Furthermore, tools for calculating an average value for fixed time intervals as an integral are provided, which is required due to the discretization.

After calculating the decrease rates from the graphs one can conclude that the order of accuracy in each dimension is 2. This means that the total error is equal to

$$\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta s^2) + \mathcal{O}(\Delta a^2)$$

plus the cross term values. The orders for cross terms can not be obtained so straightforwardly with convergence tests and are not usually a part of an empiric error approximation.

When error reaches the order of 1e-06 a truncation error which is inevitable with machine calculations comes into play. Thus, the result no longer represents the true convergence rate. This can explain the increase in convergence observed for higher number of nodes.

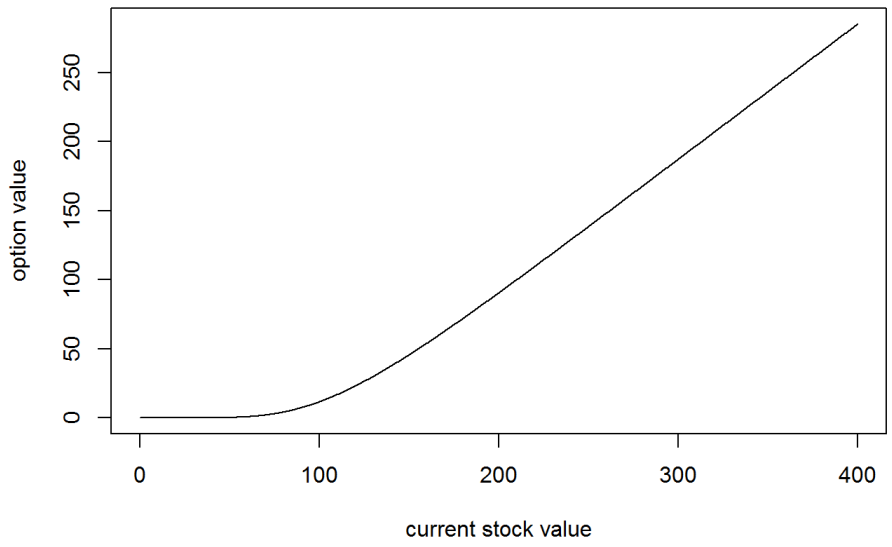


Fig. 7. Option price for various S_0

After having confirmed that the errors fall off as expected, outputs of the program for cases that do not take advantage of the parallelism, namely those with $N_s = 0$, were compared with the results of a non-parallel implementation of the same algorithm. Since there is no analytical formula representing the price of an Asian option one must use an approximation

formula as a reference to verify the result. The non-parallel implementation refers to a Curran approximation of an Asian option price to prove its *correctness*.

Figure 7 displays a resulting output for the program — a vector of option prices for all the stock values on the discretization grid at time $t = 0$. The form of the graph reaffirms the correctness of the result as its "hockey stick" shape is what is expected from a European option price. Since the option considered is a call, for stock prices smaller than the strike $K = 100$ the expected option value tends to zero as there is a smaller chance of getting any profit from buying the option. The number of stock nodes on the x axes is $N_s^{\mathbf{ref}} = 8001$.

5.2. Performance test

To check that the GPU application performs better than the CPU one and find out speed-up dynamics for increasing number of strikes a performance test is run.

The test program first defines what number of nodes in each direction is a minimum requirement to reach a certain level of accuracy with the reference solution, then constructs all the structures needed for pricing and tracks the time that the solver occurrence occupies. The performance test assumes that the convergence test has been run before it with the same set of parameters and relative errors in each dimension has been written to a file.

It is important to note that the current version of the test presupposes the worst case scenario where the errors in all the three dimensions sum up with a positive sign to compound the total error. In reality, errors may cancel each other depending on the coefficients that are hidden in the notation of O big and so a smaller number of nodes will be required for a desired accuracy. Consequently, the minimal achievable error for the test is a sum of relative errors from the convergence test for the set of nodes closest to the reference.

First, the lowest considered accuracy is examined. At each step the error is reduced by 7% and we iterate through relative error vectors until the total error, namely the sum, reaches the new level of accuracy. In case of multiple strikes, again, only the maximum error between the strike levels is considered.

For the needs of this test performance reference node levels $N_s^{\mathbf{ref}} = 4001$, $N_t^{\mathbf{ref}} = 1000$ and $N_k^{\mathbf{ref}} = 2001$ were chosen. One may recall from section 4.2 that the device global memory is limited to 8GB. Hence, with these particular node levels the maximum number of strikes such that the price values vector for all different sets of levels (excluding the time dimension) will fit into the memory is of the order 60.

$$2 * 60 * 4001 * 2001 * \texttt{sizeof(double)} = 7685760960 < 8\text{GB}$$

Figure 8 illustrates the results of the performance test for the GPU application with only

1 strike and thus no parallelism in the strike dimension compared with the CPU application that employs no parallelism. One can observe from the graph that already for accuracy 2e-03 the GPU implementation outperforms the CPU one. This means that our suggestion about engagement of the GPU being beneficial to the algorithm is proven true. Function positions are consistent with the idea that the GPU should perform better for a bigger amount of calculation, which gets available for higher accuracies. Maximum speedup in latency that one can see occurs at the highest accuracy and constitutes around a factor of 10.

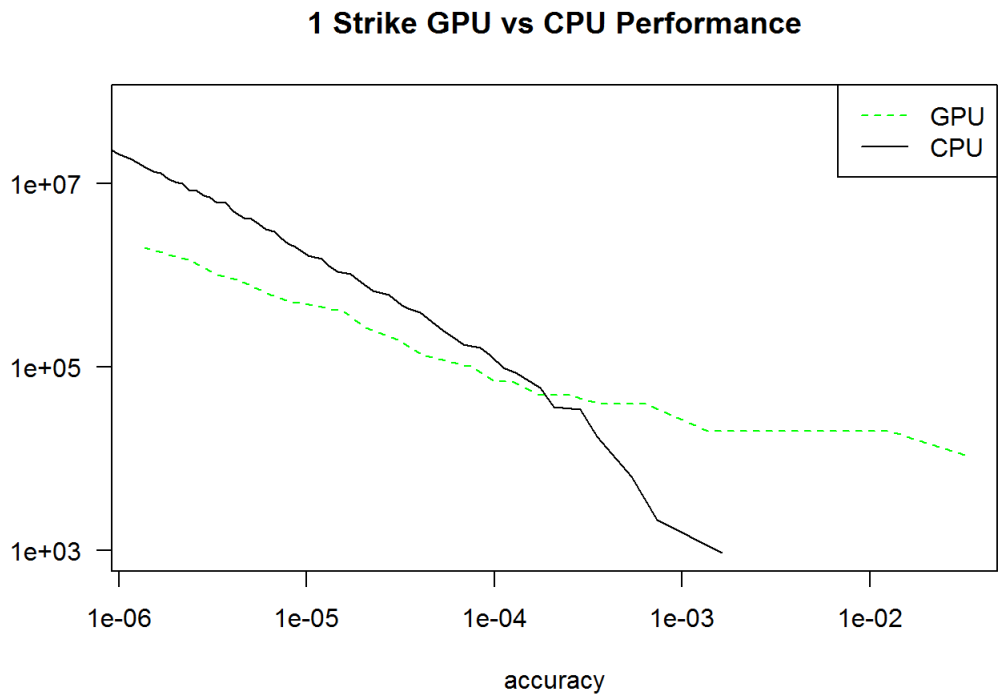


Fig. 8. GPU and CPU performance comparison

Now that it has been confirmed that for 1 strike there is a reasonably small level of accuracy for levels higher than which using the GPU is beneficial for performance, one should also examine how well the algorithm scales with adding nodes in the strike dimension.

Figure 9 shows time measurements for 1, 10, 30 and 50 simultaneously calculated option prices with different strikes. The result is divided by the number of strikes so that it is more convenient for evaluating the benefits of parallelizing between strike levels. Intuitively, one can see it as running `# strikes` occurrences of the program and comparing the result with the parallelized one. Time is measured in seconds, but the plot produced is in log scale.

We expect the dependence of time to number of strikes to be sublinear because that would mean that the set-up cost stays the same while there is still a room for piling the GPU with

more work. We also think it is likely that one will see a slow-down in the speed-up after a certain point which indicates that the processor has been fully charged.

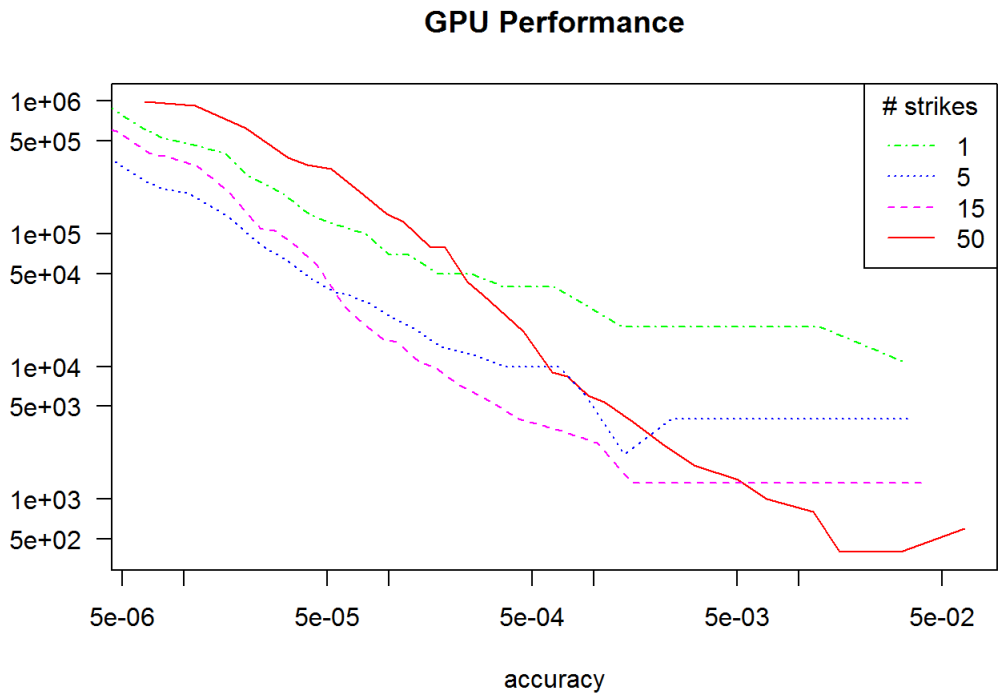


Fig. 9. GPU scaling

Indeed, the graph demonstrates that for a sertain range of accuracies the bigger amount of strike nodes results in a better performance and, respectively, the moments when it makes sense switching to the GPU version from the CPU come earlier. Already for 15 strikes the GPU program performs better than the CPU one for all considered accuracies. However, there is a trade-off between too little parallelism creating GPU stalls and too much parallelism that GPU cannot handle due to the limited amount of processors. But for each level of accuracy one can find a number of strikes for which the run time will be optimized. It cannot be seen on the graph but if we extended the x axes to higher accuracies we would have found out that there would be points where plots for $K = 5$ and $K = 15$ intersect the plot for $K = 1$. Still, up to 50 strikes considered, the effect of overcharging the GPU with work never make the performance worse than that without any parallelization at all. As for the 1 strike result, the best speedup in comparison with the CPU implementation is obtained for accuracy 1e-06 and is equal to a factor of 16.

5.3. Possible improvements

Studying time consumption of the program shows that almost all the run time is spent solving the tridiagonal matrix system of linear equations resulting from the discretisation of the Black-Scholes partial differential equation. Hence, performance could be significantly improved if one found a way to add more parallelism this specific part of the code.

Some of the enhancements of the current algorithm that create more independent parts for obtaining solution to the system of linear equations are:

- Parallel Cyclic Reduction (PCR) algorithm [11, 12, 13]. Repeatedly split the problem thus creating more work and a more complex elimination step. Such splitting of the problem allows parallel computation.
- Parallelised Thomas algorithm [14, 15]. Divide the matrix into blocks while adding border conditions such that the blocks can be proceeded in parallel.
- Iterative methods. An alternative to solving the tridiagonal linear system analytically is using an iterative method approximating the solution. This creates parallelism between the matrix rows at each iteration. A downside is that now intermediate solutions must be synchronised at each iteration. Since there is a number of various iteration methods, only a couple must be chosen for consideration that fits the problem specifics optimally.

5.4. Conclusion

The results obtained illustrate that the initial suggestion about the use of graphics processing units to pricing Asian options being advantageous is proven correct despite all the limiting factors such as need for synchronisation and a seeming lack of parallelism. Achieving 10–15 times speedup does speak in favour of the GPU as an alternative to the CPU.

Taken that the number of cores in the GPU used for the project is 1920 one could surmise that the speedup that one should expect of the same order. However, according to Amdahl's law [16], this is far-fetched for most of the programs. The law states that the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement. Thus, for each proportion of sequential code in the overall code there exists a cap in achievable speedup, and regardless of the number of added cores performance remains a certain level after some point. 10 to 20 times speedup, respectively, corresponds to 90 to 95 parallel portion of the code. Apart from these considerations, one must also take into account the limitations that are inherent in graphics processing units in particular when estimating the maximum expected speedup in latency, that were described in section 4.3.

References

[1] T. Björk. *Arbitrage Theory in Continuous Time*. Oxford Finance Series. Oxford, 2009.

[2] B. Øksendal. *Stochastic Differential Equations: An Introduction with Applications*. Universitext. Springer Berlin Heidelberg, 2013.

[3] J. F. Epperson. *An introduction to numerical methods and analysis*. Wiley, 2002.

[4] H. Windcliff, P. A. Forsyth, and K. R. Vetzal. Analysis of the stability of the linear boundary condition for the Black-Scholes equation. 2003.

[5] M. B. Giles and R. Carter. Convergence analysis of Crank-Nicolson and Rannacher time-marching. Oxford, 2005. Oxford University Computing Laboratory.

[6] C. Hirsch. *Numerical Computation of Internal and External Flows: Fundamentals of Computational Fluid Dynamics*. Elsevier/Butterworth-Heinemann, 2 edition, 2007.

[7] W. T. Lee. Tridiagonal matrices: Thomas algorithm. University of Limerick.

[8] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Applications of GPU Computing Series. Elsevier Science, 2010.

[9] NVIDIA Corp. *CUDA C Best Practices Guide*.

[10] NVIDIA Corp. *CUDA C Programming Guide*.

[11] D. Egloff. High performance finite difference PDE solvers on GPUs. QuantAlea GmbH, Switzerland, 2010.

[12] M. T. Heath. Parallel numerical algorithms. chapter 9. University of Illinois, 2013.

[13] Y. Zhang, J. Cohen, and J. D. Owens. Fast tridiagonal solvers on the GPU. University of California, 2010.

[14] M. Barta. Parallelised Thomas algorithm for solution of TDM systems: Foundations and applications to implicit numerical schemes for integration of PDEs. Astronomical Institute of Czech Academy of Sciences, 2011.

[15] G. E. Karniadakis and R. M. Kirby II. *Parallel Scientific Computing in C++ and MPI: A seamless approach to parallel algorithms and their implementation*. Cambridge University Press, 2013.

[16] D. P. Rodgers. Improvements in multiprocessor system design. In *Proceedings of the 12th annual international symposium on Computer architecture*, volume 13 of *3*, pages 225–231, Boston, Massachusetts, USA, 7 1985. IEEE Computer Society Press Los Alamitos.

[17] P. Wilmott. *Paul Wilmott on Quantitative Finance*. Wiley, 2000.

[18] R. Zvan, P. A. Forsyth, and K. R. Vetzal. Discrete Asian barrier options. Waterloo, Ontario, 1998. University of Waterloo.

[19] R. Zvan, P. A. Forsyth, and K. R. Vetzal. Robust numerical methods for PDE models of Asian options. Waterloo, Ontario, 1997. University of Waterloo.

[20] J. N. Dewynne and P. Wilmott. A note on average rate options with discrete sampling. *SIAM Journal on Applied Mathematics*, 55(1):267–276, 1995.

[21] N. Cai, Y. Song, and S. Kou. A general framework for pricing Asian options under Markov processes. *Operations Research*, 63(3):540–554, 2015.

[22] P. S. Hagan and G. West. Interpolation methods for curve construction. *Applied Mathematical Finance*, 13(2):89–129, 2006.

[23] F. Le Floc'h. Stable interpolation for the yield curve. Paris, 2013. Calypso Technology.

[24] A. K. Parrott and S. Rout. Semi-Lagrange time integration for PDE models of Asian options. London. University of Greenwich.

[25] R. Rannacher. Finite element solution of diffusion problems with irregular data. *Numerische Mathematik*, 43(20):309–328, 1984.

[26] L. C. G. Rogers and Z. Shi. The value of an Asian option. *Journal of Applied Probability*, 32:1077–1088, 1995.