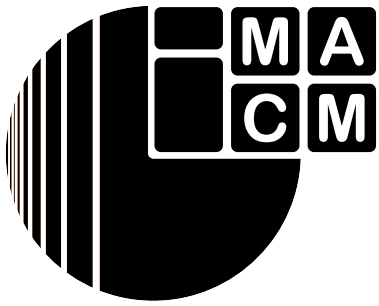


Bergische Universität Wuppertal

Fachbereich Mathematik und Naturwissenschaften

Institute of Mathematical Modelling, Analysis and Computational Mathematics (IMACM)

Preprint BUW-IMACM 19/20

Lorenc Kapllani, Long Teng and Matthias Ehrhardt

A Multistep Scheme to solve Backward Stochastic Differential Equations for Option Pricing on GPUs

June 2019

http://www.math.uni-wuppertal.de

A Multistep Scheme to solve Backward Stochastic Differential Equations for Option Pricing on GPUs

Lorenc Kapllani, Long Teng and Matthias Ehrhardt

Abstract The goal of this work is to parallelize the multistep method for the numerical approximation of the Backward Stochastic Differential Equations (BSDEs) in order to achieve both, a high accuracy and a reduction of the computation time as well. In the multistep scheme the computations at each grid point are independent and this fact motivates us to select massively parallel GPU computing using CUDA. In our investigations we identify performance bottlenecks and apply appropriate optimization techniques for reducing the computation time, using a uniform domain. Finally, a Black-Scholes BSDE example is provided to demonstrate the achieved acceleration on GPUs.

1 Introduction

In this work we parallelize the multistep scheme developed in [19] to approximate numerically the solution of the following (decoupled) *forward backward stochastic differential equation (FBSDE)*:

$$\begin{cases} dX_t &= a(t,X_t)dt + b(t,X_t)dW_t, \quad X_0 = x_0, \\ -dy_t &= f(t,X_t,y_t,z_t)dt - z_t\,dW_t, \\ y_T &= \xi = g(X_t), \end{cases} \tag{1}$$

Lorenc Kapllani
University of Wuppertal, Applied Mathematics and Numerical Analysis, Gaussstrasse 20, D-42119 Wuppertal, Germany, e-mail: kapllani@math.uni-wuppertal.de

Long Teng
University of Wuppertal, Applied Mathematics and Numerical Analysis, Gaussstrasse 20, D-42119 Wuppertal, Germany, e-mail: teng@math.uni-wuppertal.de

Matthias Ehrhardt
University of Wuppertal, Applied Mathematics and Numerical Analysis, Gaussstrasse 20, D-42119 Wuppertal, Germany, e-mail: ehrhardt@math.uni-wuppertal.de

where $X_t, a \in \mathbb{R}^n$, b is a $n \times d$ matrix, W_t is a d-dimensional Brownian motion, $f(t, X_t, y_t, z_t) : [0,T] \times \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{m \times d} \to \mathbb{R}^m$ is the driver function and ξ is the terminal condition. We see that the terminal condition y_T depends on the final value of a forward stochastic differential equation (SDE). For $a = 0$ and $b = 1$, namely $X_t = W_t$, one obtains a *backward stochastic dierential equation (BSDE)* of the form

$$\begin{cases} -dy_t &= f(t, y_t, z_t)dt - z_t\, dW_t, \\ y_T &= \xi = g(W_T), \end{cases} \tag{2}$$

where $y_t \in \mathbb{R}^m$ and $f(t, y_t, z_t) : [0,T] \times \mathbb{R}^m \times \mathbb{R}^{m \times d} \to \mathbb{R}^m$. In the sequel of this work, we investigate the acceleration of numerical scheme developed in [19] for solving (2). Note that the developed schemes can be applied also for solving (1), where the general Markovian diffusion X_t can be approximated, e.g., by using the Euler-Scheme.

The existence and uniqueness of the solution of (2) were proven by Pardoux and Peng [12]. Peng [13] obtained a direct relation between forward-backward stochastic differential equations (FBSDEs) and partial differential equations (PDEs). Based on this relationship, many numerical methods are proposed, e.g. probabilistic based methods in [3, 4, 8, 11, 20], tree-based methods in [5, 16] etc. El Karoui, Peng and Quenez [7] showed that the solution of a linear BSDE is in fact the pricing and hedging strategy of an option derivative. This was the first claim of application of BSDEs in finance.

In general the solution of BSDEs cannot be established in a closed form. Therefore, a numerical method is mandatory. There are two main classes of numerical methods for approximating the solution of BSDEs. The first class is related with the PDE equivalent based on the Feynman-Kac formula and the second is based on the BSDE. Many methods have been developed, but one of the most interesting (due to the ability to achieve very high accuracy) is developed by Zhao, Zhang and Ju [19]. They used Lagrange interpolating polynomials to approximate the integrals, given the values of integrands at multiple time levels. One of the drawbacks of their method is the computation time. However, the method is highly parallel. Hence, due to simple and intense calculations, the best computing environment is the one offered by GPUs rather than CPUs.

Many acceleration strategies have been developed to solve option pricing problems on the GPU with different mathematical models. However, little work is based on BSDEs. Dai, Peng and Dong [6] solved a linear BSDE on the GPU with the theta-scheme method. They analyzed the effects of the thread number per block to increase the speedup. The parallelized program using CUDA achieved high speedups and showed that the GPU architecture is well suited for solving the BSDEs in parallel. Later in 2011, they developed acceleration strategies for option pricing with nonlinear BSDEs using a binomial lattice based method [14]. To increase the speedup, they reduce the global memory access frequency by avoiding the kernel invocation on each time step. Also, due to the load imbalance produced by the binomial grid, they provided load-balanced strategies and showed that the acceleration algorithms

exhibit very high speedup over the sequential CPU implementation and therefore suitable for real-time application.

In 2014, Peng, Liu, Yang and Gong [15] considered solving high dimensional BSDEs on GPUs with application in high dimensional American option pricing. A *Least Square Monte-Carlo (LSMC)* method based numerical algorithm was studied, and summarised in four phases. Multiple factors which affect the performance (task allocation, data store/access strategies and the thread synchronisation) were taken into consideration. Results showed much better performance than the CPU version. In 2015, Gobet, Salas, Turkedjiev and Vasquez [9] designed a new algorithm for solving BSDEs based on LSMC. Due to stratification, the algorithm is very efficient especially for large scale simulations. They showed big speedups even in high dimensions.

Next we introduce some preliminary elements which are needed to understand the multistep scheme. We start with the relation of BSDEs and PDEs. Let $(\Omega, \mathscr{F}, \mathbb{P}, \{\mathscr{F}_t\}_{0\leq t\leq T})$ be a complete, filtered probability space. In this space a standard d-dimensional Brownian motion W_t is defined, such that the filtration $\{\mathscr{F}_t\}_{0\leq t\leq T}$ is generated. We define $\|\cdot\|$ as the standard Euclidean norm in the Euclidean space $\mathbb{R}^m$ or $\mathbb{R}^{m\times d}$ and $L^2 = L^2_F(0,T;\mathbb{R}^d)$ the set of all $\{\mathscr{F}_t\}$-adapted and square integrable processes valued in $\mathbb{R}^d$. A pair of processes $(y_t, z_t) : [0,T]\times\Omega \to \mathbb{R}^m \times \mathbb{R}^{m\times d}$ is the solution of BSDE (2) if it is $\{\mathscr{F}_t\}$-adapted, square integrable, and satisfies (2) in the sense of

$$y_t = \xi + \int_t^T f(s, y_s, z_s)\, ds - \int_t^T z_s\, dW_s, \quad t \in [0,T), \tag{3}$$

where $f(t, y_t, z_t) : [0,T]\times\mathbb{R}^m\times\mathbb{R}^{m\times d} \to \mathbb{R}^m$ is $\{\mathscr{F}_t\}$-adapted and the third term on the right-hand side is an Itô-type integral. This solution exist under "reasonable" regularity conditions [12]. Let us consider the following:

$$y_t = u(t, W_t), \quad z_t = \nabla u(t, W_t) \quad \forall t \in [0,T), \tag{4}$$

where ∇u denotes the derivative of $u(t,x)$ with respect to the spatial variable x and $u(t,x)$ is the solution of the following (backward in time) parabolic PDE:

$$\frac{\partial u}{\partial t} + \frac{1}{2}\sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2} + f(t, u, \nabla u) = 0, \tag{5}$$

with the terminal condition $u(T,x) = \phi(x)$. Under "reasonable" conditions, the PDE (5) possess a unique solution $u(t,x)$. Therefore, for $\xi = \phi(W_T)$, the pair (y_t, z_t) is the unique solution of BSDE (3).

Due to conditional expectations that compound the numerical method, the following notations will be used. Let $\mathscr{F}_s^{t,x}$ for $t \leq s \leq T$ be a σ-field generated by the Brownian motion $\{x + W_r - W_t, t \leq r \leq s\}$ starting from the time-space point (t,x). We define $E_s^{t,x}[X]$ as the conditional expectation of the random variable X under the filtration $\mathscr{F}_s^{t,x}$, i.e. $E_s^{t,x}[X] = E[X|\mathscr{F}_s^{t,x}]$.

This work is organized as follows. In Section 2 we introduce the multistep scheme. Next, in Section 3 our algorithmic framework for using GPU is presented. In Section 4 we illustrate our findings with the Black-Scholes example.

2 The Multistep scheme

In this section we briefly present the multistep scheme. This is done in two steps, the first corresponds to the derivation of the stable semi-discrete scheme, as only the time domain is discretized. Furthermore, the space is discretized and the fully stable multistep scheme is achieved. Note that the scheme will be presented for the one-dimensional case (but recall that in principle it can be generalized for the d-dimensional case).

2.1 The stable semi-discrete scheme

Let N be a positive integer and $\Delta t = T/N$ the step size that partitions uniformly the time interval $[0,T]$: $0 = t_0 < t_1 < \cdots < t_{N-1} < t_N = T$, where $t_i = t_0 + i\Delta t$, $i = 0,1,\ldots,N$. Let k and K_y be two positive integers such that $1 \leq k \leq K_y \leq N$. The BSDE (3) can be expressed as

$$y_{t_n} = y_{t_{n+k}} + \int_{t_n}^{t_{n+k}} f(s,y_s,z_s)\,ds - \int_{t_n}^{t_{n+k}} z_s\,dW_s. \tag{6}$$

In order to approximate y_{t_n} based on the later information $[t_n, t_{n+k}]$, we need to adapt it to the filtration (that is already generated, since we are solving it backwards). Therefore, taking the conditional expectation $E^x_{t_n}[\cdot]$ in (6), we have

$$y_{t_n} = E^x_{t_n}\left[y_{t_{n+k}}\right] + \int_{t_n}^{t_{n+k}} E^x_{t_n}\left[f(s,y_s,z_s)\right] ds \tag{7}$$

where the third term of (6) is disappeared as it is an Itô-type integral. In order to approximate the integral part of (7), Zhao [19] considered the Lagrange interpolating method, since the $E^x_{t_n}\left[f(s,y_s,z_s)\right]$ is a deterministic function of s. Given the values of $\left(t_{n+i}, E^x_{t_n}\left[f(t_{n+i},y_{t_{n+i}},z_{t_{n+i}})\right]\right)$ and using Lagrange interpolating polynomial, (7) becomes

$$y_{t_n} = E^x_{t_n}\left[y_{t_{n+k}}\right] + k\Delta t \sum_{i=0}^{K_y} b^k_{K_y,i} E^x_{t_n}\left[f(t_{n+i},y_{t_{n+i}},z_{t_{n+i}})\right] + R^n_y, \tag{8}$$

where $b^k_{K_y,i}$ are the coefficients derived from the integration of Lagrange interpolating polynomial [19] and R^n_y is the error due to the former.

Next, we derive a semi-discretized form for the z_t process. Let $\Delta W_s = W_s - W_{t_n}$ for $s \geq t_n$. Then ΔW_s is a standard Brownian motion with mean 0 and standard deviation $\sqrt{s-t_n}$. Let l and K_z be two positive integers such that $1 \leq l \leq K_z \leq N$. Using l instead of k in (6), multiplying both sides by $\Delta W_{t_{n+l}}$, taking the conditional expectation $E^x_{t_n}[\cdot]$ and using the Itô isometry we obtain

$$0 = E^x_{t_n}\left[y_{t_{n+l}} \Delta W_{t_{n+l}}\right] + \int_{t_n}^{t_{n+l}} E^x_{t_n}\left[f(s, y_s, z_s)\Delta W_s\right] ds - \int_{t_n}^{t_{n+l}} E^x_{t_n}\left[z_s\right] ds. \tag{9}$$

Using again the Lagrange interpolation method to approximate the two integrals in (9), we have

$$\begin{aligned} 0 = E^x_{t_n}\left[y_{t_{n+l}} \Delta W_{t_{n+l}}\right] + l\Delta t \sum_{i=0}^{K_z} b^l_{K_z,i} E^x_{t_n}\left[f(t_{n+i}, y_{t_{n+i}}, z_{t_{n+i}})\Delta W_{t_{n+i}}\right] + R^n_{z1} \\ - l\Delta t \sum_{i=0}^{K_z} b^l_{K_z,i} E^x_{t_n}\left[z_{t_{n+i}}\right] - R^n_{z2}, \end{aligned} \tag{10}$$

where $b^l_{K_z,i}$ are the coefficients derived from the integration of the Lagrange interpolating polynomial and (R^n_{z1}, R^n_{z2}) are the errors for the first and second integrals in (9).

Consider (y^n, z^n) as an approximation of (y_t, z_t), the semi-discrete scheme is defined as follows: Given random variables (y^{N-i}, z^{N-i}), $i = 0, 1, \ldots, K-1$ with $K = \max\{K_y, K_z\}$, find the random variables (y^n, z^n), $n = N-K, \ldots, 0$ such that

$$\begin{aligned} y^n &= E^x_{t_n}\left[y^{n+k}\right] + k\Delta t \sum_{i=0}^{K_y} b^k_{K_y,i} E^x_{t_n}\left[f(t_{n+i}, y^{n+i}, z^{n+i})\right] \\ 0 &= E^x_{t_n}\left[z^{n+l}\right] + \sum_{i=1}^{K_z} b^l_{K_z,i} E^x_{t_n}\left[f(t_{n+i}, y^{n+i}, z^{n+i})\Delta W_{t_{n+i}}\right] - \sum_{i=0}^{K_z} b^l_{K_z,i} E^x_{t_n}\left[z^{n+i}\right]. \end{aligned} \tag{11}$$

Zhao [19] showed that in order to have a stable semi-discrete scheme, the following should hold:

$$\begin{aligned} &k = K_y, \qquad \text{with} \quad K_y = 1, 2, \ldots, 7 \quad \text{and} \quad K_y = 9 \\ &l = 1, \quad \text{with} \quad K_z = 1, 2, 3. \end{aligned} \tag{12}$$

The coefficients are presented in the following Table 1 and 2.

2.2 The stable fully discrete scheme

Let $\mathbb{R}_{\Delta x}$ denote a partition of the real axis, i.e. $\mathbb{R}_{\Delta x} = \left\{x_i | x_i \in \mathbb{R}, i \in \mathbb{Z}, x_i < x_{i+1}, \lim_{i\to+\infty} x_i = +\infty, \lim_{i\to-\infty} x_i = -\infty\right\}$. The fully discrete scheme is defined as (cf. [19]): Given random variables (y^{N-l}_i, z^{N-l}_i), $l = 0, 1, \ldots, K-1$ with $K =$

Table 1: The coefficients $\{b_{K_y,i}^{K_y}\}_{i=0}^{K_y}$ until $K_y = 3$.

K_y	$b_{K_y,i}^{K_y}$			
	$i=0$	$i=1$	$i=2$	$i=3$
1	$\frac{1}{2}$	$\frac{1}{2}$		
2	$\frac{1}{6}$	$\frac{4}{6}$	$\frac{1}{6}$	
3	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$

Table 2: The coefficients $\{b_{K_z,i}^{1}\}_{i=0}^{K_z}$ for $K_z = 1,2,3$

K_z	$b_{K_z,i}^{1}$			
	$i=0$	$i=1$	$i=2$	$i=3$
1	$\frac{1}{2}$	$\frac{1}{2}$		
2	$\frac{5}{12}$	$\frac{8}{12}$	$-\frac{1}{12}$	
3	$\frac{9}{24}$	$\frac{19}{24}$	$-\frac{5}{24}$	$\frac{1}{24}$

$\max\{K_y, K_z\}$, find the random variables (y_i^n, z_i^n), $n = N-K,\dots,0$ such that

$$
\begin{aligned}
y_i^n &= \hat{E}_{t_n}^{x_i}\left[\hat{y}^{n+K_y}\right] + K_y \Delta t \sum_{j=1}^{K_y} b_{K_y,j}^{K_y} \hat{E}_{t_n}^{x_i}\left[f(t_{n+j}, \hat{y}^{n+j}, \hat{z}^{n+j})\right] + K_y \Delta t\, b_{K_y,0}^{K_y} f(t_n, y_i^n, z_i^n) \\
0 &= \hat{E}_{t_n}^{x_i}\left[\hat{z}^{n+1}\right] + \sum_{j=1}^{K_z} b_{K_z,j}^{1} \hat{E}_{t_n}^{x_i}\left[f(t_{n+j}, \hat{y}^{n+j}, \hat{z}^{n+j})\, \Delta W_{t_{n+j}}\right] \\
&\quad - \sum_{j=1}^{K_z} b_{K_z,j}^{1} \hat{E}_{t_n}^{x_i}\left[\hat{z}^{n+j}\right] - b_{K_z,0}^{1} z_i^n,
\end{aligned}
\tag{13}
$$

where (y_i^n, z_i^n) denotes the approximation of $\big(y(t_n,x_i), z(t_n,x_i)\big)$, $\hat{E}_{t_n}^{x_i}[\cdot]$ is the approximation of $E_{t_n}^{x_i}[\cdot]$ and $(\hat{y}^{n+j}, \hat{z}^{n+j})$ are the interpolating values from (y^{n+j}, z^{n+j}) at the space point $x_i + W_{t_{n+j}} - W_{t_n}$. In order to approximate the conditional expectations, the Gauss-Hermite quadrature rule is used, due to the high accuracy that can be achieved only with a few points. Therefore, the conditional expectation can be expressed as

$$
\hat{E}_{t_n}^{x_i}\left[\hat{y}^{n+k}\right] = \frac{1}{\sqrt{\pi}} \sum_{j=1}^{L} \omega_j \hat{y}^{n+k}\big(x_i + \sqrt{2k\Delta t}\, a_j\big), \tag{14}
$$

where (ω_j, a_j), for $j = 1,\dots,L$ are the weights and roots of the Hermite polynomial of degree L (see [19]). In the same way, one can express the other conditional expectations in (13).

The error of the method exhibits different behaviours due to the time-space discretization. However, the maximal order of convergence is 3 for both processes. For

technical details, we refer to [19]. Due to the high order of convergence, the numerical method can achieve very high accuracy. It can be observed from (13) that the calculations on each point are independent. Therefore parallelization techniques can be easily adapted. In the next Section 3, we discuss the algorithmic framework.

3 The Algorithmic Framework

3.1 The Algorithm

According to Section 2, the whole process for solving (2) is divided into 3 steps.

1. **Construct the time-space discrete domain.**
   We divide the time period $[0,T]$ into N time steps using $\Delta t = T/N$ and get $N+1$ time layers. Moreover, in order to balance the errors in time and space directions, we adjust the space step size Δx and the time step size Δt such that they satisfy the equality $(\Delta x)^r = (\Delta t)^{q+1}$, where $q = \min(K_y+1, K_z)$ and r denotes the global error from the interpolation method used to generate the non-grid points when calculating the conditional expectations.
2. **Calculate K initial solutions with $K = \max\{K_y, K_z\}$.**
   Since only the terminal value is given, one needs to generate the other $K-1$ values. This can be done by running a 1-step scheme for $[t_{N-K+1}, t_{N-1}]$ with a higher number of time points such that the $K-1$ produced initial values will have neglectable error.
3. **Calculate the numerical solution (y_0, z_0) backward using equation** (13)**.**
   Note that the calculation for the y_t process is done implicitly by Picard iteration.

3.2 Preliminary considerations

In the numerical experiments, we have considered the following points:

- The space domain needs to be truncated. Since the space domain represent the Brownian motion discretization, in our test we use $[-16, 16]$.
- When generating the non-grid points for the calculation of conditional expectations, some will be outside of the domain. For such points, the value on the boundaries is considered, as the desired solution will not be affected.
- Due to uniformity of the grid, one does not need to consider $2K$ (K for y_t and K for z_t) interpolations for each new calculation, but only 2. This is due to the following:
  Suppose we are at time layer t_{n-K}. To calculate y_t and z_t values on this time layer, one needs the calculation of conditional expectations for K time layers. The cubic spline interpolation is used to find the non-grid values, and the necessary linear systems are solved. For instance, the coefficients for y_t process are $A_y \in$

$\mathbb{R}^{K\times M}$. All the spline coefficients are stored. When we are at time layer t_{n-K-1}, only the spline interpolation corresponding to the previous calculated values is considered. Then, the columns of matrix A_y are shifted $+1$ to the right in order to delete the last column and enter the current calculated coefficients in the first column. The new A_y is used for the current step. The same procedure is followed until t_0. This reduces the amount of work for the algorithm.

- There is a very important benefit from the uniformity of the grid. When we need to find the position of the non-grid point, a naïve search algorithm is to loop over the grid points. In the worst case, a O(M) work is needed. However, this can be done in O(1), i.e. the for loop is removed. Recall that each new point is generated as $X_j = x_i + \sqrt{2\Delta t k}\, a_j$. This means that taking int$((X_j - x_{min})/\Delta x)$ gives the left boundary of the grid interval that X_j belongs to. This reduces the total computation time substantially, as it will be demonstrated in the numerical experiments.

3.3 The Parallel implementation

In this Section we present the naïve parallelization of the multistep scheme. Nevertheless, we have kept into attention the optimal CUDA execution model, i.e. creating arrays such that the access will be aligned and coalesced, reducing the redundant access to global memory, using registers when needed etc.

The first and second steps of the algorithm are implemented in the host. The third step is fully implemented in the device. Recall from (13) that the following steps are needed to calculate the approximated values on each time layer backward:

- **Generation of non-grid points $X_j = x_i + \sqrt{2\Delta t k}\, a_j$.**
  In the uniform domain, the non-grid points need to be generated only once. To do this, a kernel is created where each thread generates L points.
- **Calculation of the values $\hat{y}$ and $\hat{z}$ at the non-grid points.**
  This is the most time consuming part of the algorithm, since it involves the solution of two linear systems (see third point in Subsection 3.2) arising from the spline interpolation.
  We used the BiCGSTAB iterative method since the matrix is tridiagonal. To apply the method, we considered the cuBLAS and cuSPARSE libraries. For the inner product, second norm and addition of vectors, we use the cuBLAS library.
  For the matrix vector multiplication, we use the cuSPARSE library with the compressed sparse row format, due to the structure of the system matrix. Moreover, we created a kernel to calculate the spline coefficients based on the solved systems under the spline interpolation idea.
  Finally, a kernel to apply the last point in Subsection 3.2 was created to find the values at non-grid points. Note that each thread is assigned to find the values.
- **Calculation of the conditional expectations.**
  For the first conditional expectations in the right hand side of (13), we created one kernel, where each thread calculates one value by using (14). Moreover, we

merged the calculation of three conditional expectation in one kernel, namely

$$\hat{E}^{x_i}_{t_n}\left[\hat{z}^{n+j}\right], \quad \hat{E}^{x_i}_{t_n}\left[f(t_{n+j},\hat{y}^{n+j},\hat{z}^{n+j})\right], \quad \hat{E}^{x_i}_{t_n}\left[f(t_{n+j},\hat{y}^{n+j},\hat{z}^{n+j})\,\Delta W_{t_{n+j}}\right],$$

for $j = 1,2,\ldots,K$. This reduces the accessing of data multiple times from the global memory. Note that one thread calculates three values as in (14).

- **Calculation of the z_t values.**
  The second equation in (13) is used and each thread calculates one value.
- **Calculation of the y_t values.**
  The first equation in (13) is used and each thread calculates one value, using the Picard iterative process.

4 Numerical Results

We implement the parallel algorithm using CUDA C programming. The parallel computing times are compared with the serial ones on a CPU. Furthermore, the speedups are calculated. The CPU is Intel(R) Core(TM) i5-4670 3.40Ghz with 4 cores. The GPU is a NVIDIA GeForce 1070 Ti with a total 8GB GDDR5 memory.

In the following we consider an option pricing example, the Black-Scholes model. Consider a security market that contains one bond with price p_t and one stock with price S_t. Therefore, their dynamics are described by:

$$\begin{cases} dp_t &= r_t p_t\,dt, \quad t \geq 0, \\ p_0 &= p, \end{cases} \tag{15}$$

$$\begin{cases} dS_t &= \mu_t S_t dt + \sigma_t S_t\,dW_t, \quad t \geq 0, \\ S_0 &= x, \end{cases} \tag{16}$$

where r_t denotes the interest rate of the bond, p is its current value, μ_t is the expected return on the stock S_t, σ_t is the volatility of the stock, x is its current value and W_t denotes the Brownian motion.

An European Option is a contract that gives the owner the right, but not the obligation, to buy or sell the underlying security at a specific price, known as the strike price K, on the option's expiration date T. A European call option gives the owner the right to purchase the underlying security, while a European put option gives the owner the right to sell the underlying security. Let us take the European call option as an example. The decision of the holder will depend on the stock price at maturity T. If the value of the stock $S_T < K$, then the holder would discard the option; whereas if $S_T > K$, the holder would use the option and make a profit of $S_T - K$. Therefore, the payoff of a call option is $(S_T - K)^+$ and for a put option $(K - S_T)^+$, where $(f)^+ = \max(0, f)$. The option pricing problem of the writer (or seller) is to determine a premium for this contract at present time t_0. Note that the payoff function is an $\{\mathscr{F}_T\}$-measurable random variable.

Suppose that an agent sells the option at price y_t and then invests it in the market. Denote his wealth on each time by y_t. Assume that at each time the agent invests a portion of his wealth in an amount given by π_t into the stock, and the rest $(y_t - \pi_t)$ into the bond. Now the agent has a portfolio based on the stock and the bond. Considering a stock that pays a dividend $\delta(t,S_t)$, the dynamics of the wealth process y_t are described by

$$\begin{aligned} dy_t &= \frac{\pi_t}{S_t}\,dS_t + \frac{y_t-\pi_t}{p_t}\,dp_t + \pi_t\delta(t,S_t)\,dt \\ &= \frac{\pi_t}{S_t}(\mu_t S_t\,dt + \sigma_t S_t\,dW_t) + \frac{y_t-\pi_t}{p_t}(r_t p_t\,dt) + \pi_t\delta(t,S_t)\,dt \\ &= \big(r_t y_t + \pi_t(\mu_t - r_t + \delta(t,S_t))\big)dt + \pi_t\sigma_t\,dW_t. \end{aligned} \tag{17}$$

Let $z_t = \pi_t\sigma_t$, then

$$-dy_t = -\Big(r_t y_t + \big(\mu_t - r_t + \delta(t,S_t)\big)\frac{z_t}{\sigma_t}\Big)\,dt + z_t\,dW_t. \tag{18}$$

For a call option, one needs to solve a Forward Backward Stochastic Differential Equation (FBSDE), where the forward part is given from the SDE modelling of the stock price dynamics.

Example 1. Let us consider the Black-Scholes FBSDE

$$\begin{cases} dS_t &= \mu_t S_t dt + \sigma_t S_t\,dW_t, \quad S_0 = x, \quad t\in[0,T] \\ -dy_t &= -\Big(r_t y_t + \big(\mu_t - r_t + \delta(t,S_t)\big)\frac{z_t}{\sigma_t}\Big)\,dt + z_t\,dW_t, \quad t\in[0,T) \\ y_T &= (S_T-K)^+. \end{cases} \tag{19}$$

For constant parameters (i.e. $r_t = r$, $\mu_t = \mu$, $\sigma_t = \sigma$, $\delta_t = \delta$), the analytic solution is

$$\begin{cases} y_t &= V(t,S_t) = S_t\exp\big(-\delta(T-t)\big)N(d_1) - K\exp\big(-r(T-t)\big)N(d_2), \\ z_t &= \frac{\partial V}{\partial S}\sigma = S_t\exp\big(-\delta(T-t)\big)N(d_1)\sigma, \\ d_{1/2} &= \frac{\ln\left(\frac{S_t}{K}\right) + \left(r \pm \frac{\sigma^2}{2}\right)(T-t)}{\sigma\sqrt{T-t}}, \end{cases} \tag{20}$$

where $N(\cdot)$ is the cumulative standard normal distribution function. In this example, we consider $T = 0.33$, $K = S_0 = 100$, $r = 0.03$, $\mu = 0.05$, $\delta = 0.04$, $\sigma = 0.2$, with the solution at (t_0,S_0) being $(y_0,z_0) \doteq (4.3671, 10.0950)$.

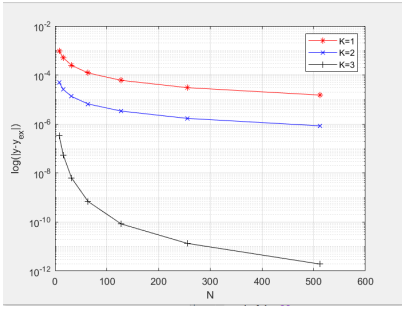
Note that the terminal condition has a non-smooth problem for the z_t process. Therefore, for discrete points near the strike price (also called at the money region), the initial value for the z_t process will cause large errors on the next time layers. To overcome this non-smoothness problem, we considered smoothing the initial conditions, cf. the approach of Hendricks [10]. For the forward part of (20), we have the analytic solution

$$S_t = S_0\exp\Big(\big(\mu - \frac{\sigma^2}{2}\big)t + \sigma W_t\Big). \tag{21}$$

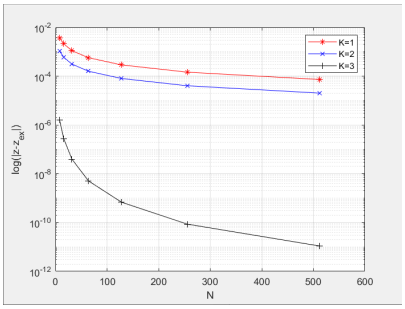
Discretizing (21), the exponential term will lead to a non-uniform grid. Therefore, instead of working in the stock price domain, we work in the log stock price domain. If we denote $X_t = \ln S_t$, then the analytic solution of X_t reads

$$X_t = X_0 + \left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W_t. \tag{22}$$

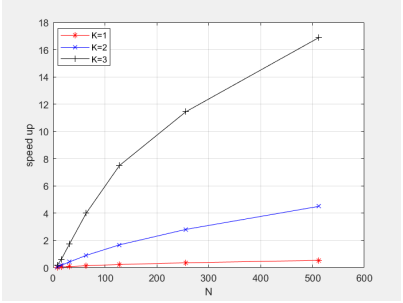
The backward part is the same as the (19). In Table 3 we show the importance of using the log stock price. Note that the speedup is relative to the serial case using a for loop.



(a) Error for the y process.



(b) Error for the z process.



(c) Speedup

N	K	M	CPU time (sec)	GPU time (sec)	Speed up
8	1	158	0.01	0.35	0.02
16	1	224	0.02	0.39	0.05
32	1	316	0.05	0.60	0.09
64	1	446	0.15	1.01	0.15
128	1	632	0.43	1.81	0.24
256	1	892	1.23	3.40	0.36
512	1	1262	3.64	6.70	0.54
8	2	350	0.02	0.30	0.07
16	2	588	0.07	0.41	0.18
32	2	990	0.26	0.61	0.43
64	2	1664	0.95	1.04	0.91
128	2	2798	3.27	1.95	1.68
256	2	4704	10.76	3.82	2.81
512	2	7912	36.31	8.07	4.50
8	3	776	0.05	0.30	0.17
16	3	1552	0.25	0.41	0.62
32	3	3104	1.12	0.64	1.76
64	3	6208	4.62	1.16	4.00
128	3	12414	19.15	2.55	7.50
256	3	24826	76.94	6.72	11.44
512	3	49650	307.09	18.20	16.87

(d) Speedup results in table format.

Fig. 1: Results of naïve parallelization for the Black-Scholes example.

The naïve results using 256 threads per block are presented in Figure 1 (note that $K_y = K_z = K$). It can be easily observed that the higher accuracy can be achieved when considering a 3-step scheme. Since we have more time layers to consider, more work can be assigned to the GPU and therefore increasing the speedup of the application. The highest speedup that we obtained for the Black-Scholes example is $17\times$.

Table 3: Comparison due to uniformity of the domain under log stock price transformation for the Black-Scholes model for $N = 256$, $K = K_y = K_z = 3$ and $M = 24826$.

Type	Time	Speedup
Serial (with for)	36825.27	
Serial (without for)	76.94	478.62
Parallel (with for)	237.57	155.01
Parallel (without for)	6.72	5476.19

```
==22320== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   58.95%  7.24543s     85124  85.116us  15.520us  207.81us  void nrm2_kernel<double, dou
                    7.88%  968.66ms      1527  634.36us  595.40us  716.39us  sp_inter_non_grid_d_no_for(d
                    7.39%  908.40ms      1527  594.89us  520.64us  763.43us  gen_non_grid(double*, double
                    7.12%  875.58ms     42562  20.571us  19.040us  237.67us  void csrMv_kernel<double, do
                    5.45%  670.20ms      1527  438.90us  388.77us  494.69us  calc_f_and_c_exp_d(double*,
                    3.55%  436.25ms     85124  5.1240us  3.5840us  8.8000us  void dot_kernel<double, doub
                    1.62%  199.67ms     42562  4.6910us  3.4560us  8.0000us  void axpy_kernel_val<double,
                    1.62%  199.27ms     85124  2.3400us  1.8880us  7.9040us  void reduce_1Block_kernel<do
                    1.29%  159.04ms       509  312.45us  308.07us  333.60us  calc_c_exp_d(double*, double
                    1.19%  146.80ms     21569  6.8060us  4.4160us  8.7360us  step_3(double*, double*, dou
                    1.03%  126.59ms     42571  2.9730us     544ns  34.647ms  [CUDA memcpy HtoD]
                    0.68%  83.450ms    127687     653ns     608ns  7.6160us  [CUDA memcpy DtoH]
                    0.67%  82.697ms     21569  3.8340us  3.2960us  7.7440us  step_8(double*, double*, dou
                    0.42%  51.357ms     12192  4.2120us  3.4560us  6.2720us  copy_d(double*, double*, int
                    0.39%  48.323ms       509  94.936us  91.937us  103.94us  calc_y(double*, double*, dou
                    0.37%  45.056ms     20993  2.1460us  1.8880us  5.3760us  step_13(double*, double*, do
                    0.12%  15.323ms      6144  2.4930us  1.5360us  5.8880us  void copy_kernel<double, int
                    0.12%  14.548ms       512  28.414us  26.560us  35.040us  spline_coeff(double*, double
                    0.07%  8.9152ms       509  17.515us  16.512us  19.680us  calc_z(double*, double*, dou
ouble*, int, double)
                    0.00%  15.168us         1  15.168us  15.168us  15.168us  CSR(double*, int*, int*, int
                    0.00%  4.0640us         1  4.0640us  4.0640us  4.0640us  csinterp_d(double*, double*,
                    0.00%     736ns         1     736ns     736ns     736ns  [CUDA memset]
```

(a) Performance of naïve kernels

```
==22383== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   18.57%  960.11ms      1527  628.75us  588.39us  899.21us  sp_inter_non_grid_d_no_for(
                   17.69%  914.31ms      1527  598.76us  521.16us  765.06us  gen_non_grid(double*, doubl
                   16.82%  869.32ms     42562  20.424us  18.880us  23.584us  void csrMv_kernel<double, d
                   12.76%  659.30ms      1527  431.76us  387.52us  487.91us  calc_f_and_c_exp_d(double*,
                   11.24%  580.94ms    127686  4.5490us  3.4240us  9.1200us  void dot_kernel<double, dou
                    5.27%  272.61ms    127686  2.1340us  1.8560us  114.79us  void reduce_1Block_kernel<d
                    3.42%  176.80ms     42562  4.1530us  2.9440us  7.7760us  void axpy_kernel_val<double
                    3.08%  159.16ms       509  312.69us  308.83us  334.11us  calc_c_exp_d(double*, doubl
                    2.72%  140.38ms     21569  6.5080us  4.4480us  8.0320us  step_3(double*, double*, do
                    1.86%  95.952ms         9  10.661ms     544ns  34.389ms  [CUDA memcpy HtoD]
                    1.53%  79.093ms    127687     619ns     576ns  7.5840us  [CUDA memcpy DtoH]
                    1.50%  77.376ms     21569  3.5870us  3.1360us  7.4240us  step_8(double*, double*, do
                    0.97%  50.224ms     12192  4.1190us  3.3600us  7.0080us  copy_d(double*, double*, in
                    0.93%  47.838ms       509  93.984us  87.809us  112.77us  calc_y(double*, double*, do
                    0.81%  41.774ms     20993  1.9890us  1.7920us  7.2960us  step_13(double*, double*, d
                    0.29%  15.097ms      6144  2.4570us  1.5040us  7.1040us  void copy_kernel<double, in
                    0.28%  14.292ms       512  27.914us  25.856us  35.840us  spline_coeff(double*, doubl
                    0.17%  8.7412ms       509  17.173us  16.352us  19.136us  calc_z(double*, double*, do
                    0.11%  5.5165ms      1024  5.3870us  4.6720us  6.1760us  rhs(double*, double*, int,
                    0.00%  14.048us         1  14.048us  14.048us  14.048us  CSR(double*, int*, int*, in
                    0.00%  3.9040us         1  3.9040us  3.9040us  3.9040us  csinterp_d(double*, double*
                    0.00%     736ns         1     736ns     736ns     736ns  [CUDA memset]
```

(b) Performance after first optimization iteration

```
==22446== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   24.09%  910.16ms      1527  596.05us  552.26us  777.51us  sp_inter_non_grid_d_no_for(d
                   22.59%  853.73ms     42562  20.058us  18.624us  231.43us  void csrMv_kernel<double, do
                   16.13%  609.41ms    127686  4.7720us  3.5840us  9.0240us  void dot_kernel<double, doub
                    7.92%  299.37ms    127686  2.3440us  1.8560us  8.0320us  void reduce_1Block_kernel<do
                    5.90%  222.81ms      1527  145.91us  133.38us  167.65us  calc_f_and_c_exp_d(double*,
                    4.44%  167.65ms     42562  3.9380us  2.9440us  7.9680us  void axpy_kernel_val<double,
                    3.74%  141.18ms     21569  6.5450us  4.5120us  7.6800us  step_3(double*, double*, dou
                    2.75%  103.89ms      1527  68.035us  61.664us  92.384us  gen_non_grid(double*, double
                    2.55%  96.399ms         9  10.711ms     544ns  34.512ms  [CUDA memcpy HtoD]
                    2.10%  79.491ms    127687     622ns     576ns  7.5520us  [CUDA memcpy DtoH]
                    2.06%  77.984ms     21569  3.6150us  3.1680us  6.9120us  step_8(double*, double*, dou
                    1.29%  48.910ms     12192  4.0110us  2.7200us  7.2320us  copy_d(double*, double*, int
                    1.27%  47.908ms       509  94.122us  87.840us  108.10us  calc_y(double*, double*, dou
                    1.10%  41.495ms     20993  1.9760us  1.7600us  6.8160us  step_13(double*, double*, do
                    0.90%  34.072ms       509  66.938us  65.377us  73.441us  calc_c_exp_d(double*, double
                    0.40%  15.041ms      6144  2.4480us  1.5040us  7.8400us  void copy_kernel<double, int
                    0.38%  14.304ms       512  27.937us  26.016us  35.040us  spline_coeff(double*, double
                    0.24%  8.9084ms       509  17.501us  16.768us  20.032us  calc_z(double*, double*, dou
                    0.16%  6.0294ms      1024  5.8880us  4.6400us  6.7840us  rhs(double*, double*, int, d
                    0.00%  13.856us         1  13.856us  13.856us  13.856us  CSR(double*, int*, int*, int
                    0.00%  4.0000us         1  4.0000us  4.0000us  4.0000us  csinterp_d(double*, double*,
                    0.00%     704ns         1     704ns     704ns     704ns  [CUDA memset]
```

(c) Performance after second optimization iteration

Fig. 2: Results of iterative parallelization for the Black-Scholes example.

Furthermore, we optimize the kernels created for the Black-Scholes BSDE for $N = 512$, $K_y = 3$ and $K_z = 3$. For this, we used the NVIDIA profiling tools (`nvprof` and `nvvp`) to gather information about performance bottlenecks and apply the proper optimization technique.

After applying `nvprof`, the main bottleneck of the application is the second norm kernel that calculates the errors in the BiCGSTAB algorithm as presented in Figure 2a. Note that this kernel is already optimized by the NVIDIA developers. However, it is a kernel that serves for a general purpose. Instead of using second norm kernel, we used the dot kernel and later took the square root. This reduced the computation time from 7.2 *s* to 580.9 *ms* as presented in Figure 2b. We consider it as the first iteration of the optimization process.

Moreover, we applied again `nvprof` and found that the next performance inhibitor is the kernel which calculates the non-grid values and another kernel that generates the non-grid values. This is due to the inefficient memory accesses. To overcome this problem, we considered loop interchanging and loop unrolling. As presented in Figure 2c, the performance of above kernels is improved, from 960.1 *ms* to 910.1 *ms* and 914.3 *ms* to 103.9 *ms* respectively. Finally, we changed the thread configuration to 128 threads per block in order to increase parallelizm and we were able to achieve a $51\times$ speedup. We present the speedups for each iteration of the optimization process in Table 4.

Table 4: Speedups of Black-Scholes model for $N = 256$, $K = K_y = K_z = 3$ and $M = 24826$.

Type	Time	Speedup
Serial	307.09	
Naïve	18.2	16.87
First iteration	7.45	41.22
Second iteration	6.02	51.01

5 Conclusions and Outlook

In this work we parallelized the multistep method developed in [19] for the numerical approximation of BSDEs on GPU.

Firstly, we presented an optimal operation to find the location of the interpolated values. This was essential for the reduction of the computational time. The numerical results exhibited a high accuracy in very small computation times. Moreover, we optimized the application after finding the performance bottlenecks and applying optimization techniques. Using the *cuBLAS* kernel to calculate the error of the BiCGSTAB iterative method, loop interchange and loop unrolling provided us a $51\times$ speedup for the Black-Scholes example.

Based on our results, the GPU architecture for the multistep scheme is well suited for the acceleration of BSDEs. For future work we will focus on parallelizing d-dimensional problems using more time steps as in [17] for a higher accuracy with financial applications such as multi-asset option pricing.

Acknowledgements The authors were partially supported by the bilateral German-Portuguese Project *FRACTAL – FRActional models and CompuTationAL Finance*, the bilateral German-Hungarian Project *CSITI – Coupled Systems and Innovative Time Integrators* and the bilateral German-Slovakian Project *ENANEFA – Efficient Numerical Approximation of Nonlinear Equations in Financial Applications* all financed by the DAAD.

References

1. Behforooz, G. and Papamichael, N., 1979. End conditions for cubic spline interpolation, J. Inst. Math. Appl., 23, pp. 355-366, 1979.
2. Behforooz, G., 1995. A Comparison of the E(3) and Not-A-Knot Cubic Splines, J. Inst. Math. Appl. Math. Comp., 72, pp. 219-223.
3. Bender, C. and Steiner, J., 2012. Least-squares Monte-Carlo for backward SDEs. In: Numerical methods in finance (pp. 257-289). Springer, Berlin, Heidelberg.
4. Bouchard, B. and Touzi, N., 2004. Discrete-time approximation and Monte-Carlo simulation of backward stochastic differential equations. Stochastic Processes and their applications, 111(2), pp. 175-206.
5. Crisan, D. and Manolarakis, K., 2012. Solving backward stochastic differential equations using the cubature method: application to nonlinear pricing. SIAM J. Fin. Math., 3(1), pp.534-571.
6. Dai, B., Peng, Y. and Gong, B., 2010. Parallel option pricing with BSDE method on GPU. In 2010 Ninth Int. Conference on Grid and Cloud Computing (pp. 191-195). IEEE.
7. El Karoui, N., Peng, S. and Quenez, M.C., 1997. Backward stochastic differential equations in finance. Mathematical Finance, 7(1), pp. 1-71.
8. Gobet, E., Lemor, J.P. and Warin, X., 2005. A regression-based Monte Carlo method to solve backward stochastic differential equations. Annal. Appl. Probab., 15(3), pp. 2172-2202.
9. Gobet, E., López-Salas, J.G., Turkedjiev, P. and Vázquez, C., 2016. Stratified regression Monte-Carlo scheme for semilinear PDEs and BSDEs with large scale parallelization on GPUs. SIAM J. Sci. Comput., 38(6), pp. C652-C677.
10. Hendricks, C., 2017. High-Order Methods for Parabolic Equations in Multiple Space Dimensions for Option Pricing Problems, Dissertation, University of Wuppertal.
11. Lemor, J.P., Gobet, E. and Warin, X., 2006. Rate of convergence of an empirical regression method for solving generalized backward stochastic differential equations. Bernoulli, 12(5), pp. 889-916.
12. Pardoux, E. and Peng, S., 1990. Adapted solution of a backward stochastic differential equation. Systems & Control Letters, 14(1), pp. 55-61.
13. Peng, S., 1991. Probabilistic interpretation for systems of quasilinear parabolic partial differential equations. Stochastics and stochastics reports, 37(1-2), pp. 61-74.
14. Peng, Y., Gong, B., Liu, H. and Dai, B., 2011. Option pricing on the GPU with backward stochastic differential equation. In 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming (pp. 19-23). IEEE.
15. Peng, Y., Liu, H., Yang, S. and Gong, B., 2014. Parallel algorithm for BSDEs based high dimensional American option pricing on the GPU. J. Comp. Inform. Syst., 10(2), pp. 763-771.

16. Teng, L., 2018. A Review of tree-based approaches to solve forward-backward stochastic differential equations. arXiv preprint arXiv:1809.00325.
17. Teng, L., Lapitckii, A. and Gnther, M., 2018. A Multi-step Scheme based on Cubic Spline for solving Backward Stochastic Differential Equations. arXiv preprint arXiv:1809.00324.
18. van der Vorst, H.A., 1992. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems, SIAM J. Sci. Stat. Comput., 13(2), pp. 631-644.
19. Zhao, W., Zhang, G. and Ju, L., 2010. A stable multistep scheme for solving backward stochastic differential equations. SIAM J. Numer. Anal., 48(4), pp. 1369-1394.
20. Zhao, W., Chen, L. and Peng, S., 2006. A new kind of accurate numerical method for backward stochastic differential equations. SIAM J. Sci. Comput., 28(4), pp. 1563-1581.