# A Replication: Parallel Solution of American Option Derivatives on GPU Clusters [5]

**Karim Layoun and Mathew Shaw**

**Georgia Institute of Technology**

## ABSTRACT

We aim to implement a parallel asynchronous method adapted to the high-dimensional American option pricing problem.

Keywords: Parallel Asynchronous Algorithms, GPU Clusters, Parallel Computing, Computational Finance, High-Dimensional Options

# Introduction

Partial Differential Equations, PDEs, are ubiquitous in multiple scientific fields. They are the mathematical foundation of the modern understanding of sound, heat, statics, dynamics, general relativity, and quantum mechanics. PDEs are also used to describe the price of financial derivatives. Quantitative Finance, a relatively young science, has taken to adapting existing solutions to problems in the developing field. Indeed, the PDE associated with any asset beyond the simplest derivative, Vanilla European Options, suffers from the same difficulties characteristic of meaningful problems in the aforementioned fields: they do not have closed-form solutions.

Numerical methods have been developed as solutions to multiple classes of such PDEs, in the context of different fields, which are then adapted to problems in other disciplines. Indeed, the citation tree of such papers span the general scientific literature, apparent in the in the reference section of the replicated paper. The American option derivative problem is akin to the parabolic convection-diffusion free-boundary value problem defined on an unbounded domain, a subset of the three-dimensional space. A general solution approach to such problems starts with the discretization of the PDE, resulting in a series of large-scale algebraic systems. Consequently, parallel iterative methods, synchronous and asynchronous, are well adapted. Moreover, GPU networks are a preferable alternative to CPU networks for these applications, illustrated in the replicated paper's results [5] and other works [10] [1] [7].

Multiple authors have explored the high-dimensional option pricing problem. For example, the replicated paper references [1], where the authors attempt to use Krylov subspace methods on a sparse matrix representation of the dicretized PDE. Subsequent papers have attempted different solution methods to the problem, as in [8] where Pagès et al. have used the Least-Square Monte Carlo method, in [2] where Ghosh et al. have developed a parallel and numerically accurate pricing of two-asset American options under the Merton jump-diffusion model, and in [6] where Mahony et al. have developed a parallel and pipeline implementation of a pascal-simplex based multi-asset option pricer.

Before proceeding to formally introducing the problem, we must accurately depict the current

state of the modern financial mathematics literature, contextualizing option pricing. Traditionally, Quantitative Finance practitioners were mainly focused on pricing financial derivatives, dealing with financial products as opposed to general financial market. However, the discipline has drastically evolved after the 2008 financial crisis revealed the downfall of the field's oversight. Furthermore, the trading environment rapidly changed with the computerization of exchanges and new trading regulations. Therefore, new topics of study have emerged, namely microstructure, optimal execution, and market-making. While derivative pricing no longer holds center stage, it remains a foundation of quantitative finance, and many financial institutions have great incentive to efficiently and accurately price evermore complex derivatives. In addition, we believe that the introduction of GPU computing to the field must follow the progression of concepts and problems, from the traditional to the contemporary.

In the spirit of "PDE numerical solutions" literature, we hope to provide the reader with a guide to general models of asynchronous methods, illustrated by the financial mathematics application of high-dimensional option pricing. Khodja et al. have implemented the parallel subdomain method without overlapping/projected block relaxation method and the parallel projected Richardson algorithm, detailing how problem form informs solution methods. The authors have found that the projected block relaxation method is 3 times slower than the projected block relaxation method even if it converges 4 times faster than the latter. And so, we will implement the faster projected Richardson method, nevertheless explaining how the structure of the slower algorithm hinders its performance.

We have written the report for the uninitiated yet motivated reader, starting with the easily visualized one dimensional problem and the associated sequential versions of the aforementioned solution methods. We then describe the framework parallelizing iterative methods, expand the problem definition to three dimensions, develop the parallel numerical methods, and present the results of our implementation.

# Simple American Option Model

## "VANILLA" FINANCIAL OPTIONS

In the interest of completeness, and for the report to be mostly self-contained, we will briefly introduce options, the simplest financial derivative and the subject of the replicated paper. The most basic financial option, the European call option, is a contract giving the holder of the option the right to purchase an underlying asset at a prescribed time in the future, the expiry date, for a prescribed amount, the exercise or strike price. Naturally, market participants attempt to value or price the rights afforded by such contracts. Clearly, at expiry, the price of the option is $max(S_T - K, 0)$, where:

- $S_\tau$ is the price of the underlying at time $\tau$
- $K$ is the strike price

Prior to expiry, pricing options is a more involved exercise, as asset prices move randomly and follow the lognormal random walk $df = \sigma dX + (r - \frac{1}{2}\sigma^2)d\tau$, where:

- $f = log(S)$
- $dX$ is a Wiener process
- $r$ is the drift rate under the risk neutral measure $\mathbf{Q}$, or the interest rate
- $\sigma$, the volatility, or standard deviation of asset returns

Thus, the value $\mathbf{C}$ of the described option is a function of the current value of the underlying asset $\mathbf{S}$, and time $\tau$. The value of the option also depends on the option parameters $(K, T)$, random walk parameters $(r, \sigma)$, and $\tau$. A seminal result in quantitative finance states that any derivative security ($\mathbf{U}$) whose price depends only on the current value of $S$ and on $\tau$, and which is paid for up-front, must satisfy the Black-Scholes equation, a partial differential equation; more specifically, a second order linear parabolic equation:

$$\frac{\partial U}{\partial \tau} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} + rS\frac{\partial U}{\partial S} - rU = 0$$

The equation is backward parabolic, which we typically solve backwards in time (starting at expiry) posing two boundary conditions on $S$ and one on $\tau$ [9]. The equation is akin to the classical diffusion equation, and has an explicit solution.

## AMERICAN OPTIONS

A possible variation to the aforementioned contract gives the holder of the option to exercise it at any point during its lifetime: American options. The problem of pricing these derivatives is more complicated. Indeed, for each $\tau$, we should determine the price of the option and whether to exercise it. And so, pricing American options is a free boundary problem, where at each time $\tau$, there is a value $S$, the optimal exercise price $S_f$, such that to one side of the boundary $S_f$ one should hold the option and to the other one should exercise it. Furthermore, note that the value of the option is always greater than or equal to that of the payoff function. If not so, there would be opportunities for instantaneous risk-free profit, arbitrage, which would be quickly eliminated by astute market participants. Consequently, the option should be exercised when the value of the option is equal to that of the payoff function, and held otherwise. Thus, the price of American options satisfy the following inequality:

$$\frac{\partial U}{\partial \tau} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} + rS\frac{\partial U}{\partial S} - rU \leq 0$$

The right to exercise the option at any time manifests itself in the inequality, as the PDE no longer has a unique value.

Furthermore, the Black-Scholes formulation of the free boundary problem for American options can be reduced to a linear complementarity problem [9]. Hence, the linear complementarity form defined on $[0,T] \times \mathbb{R}$:

$$\begin{aligned}
&\frac{\partial U}{\partial \tau} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} + rS\frac{\partial U}{\partial S} - rU \leq 0 \\
&\left(\frac{\partial U}{\partial \tau} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} + rS\frac{\partial U}{\partial S} - rU\right)(U - \phi) = 0 \\
&U \geq \phi \\
&\phi = U(T,S)
\end{aligned}$$

## TOWARDS A NUMERICAL SOLUTION

The previous boundary value problem is defined on the unbounded domain $\mathbb{R}$. This inconvenience is remedied by setting boundaries on the spatial domain. It can be proven that the solution of the bounded problem converges to the solution of the unbounded problem [4]. Let $\Omega = [0,L] \subset \mathbb{R}$ be the bounded domain, where the parameter $L$ denotes the largest asset value one wishes to consider. Thus, we must define boundary conditions for the problem. Traditionally, the Dirichlet condition ($U$ fixed on $\delta\Omega$) or the Neumann condition (normal derivative of $U$ fixed on $\delta\Omega$). The authors have chosen the Dirichlet condition such that $U(s,\tau) = 0 \; \forall \tau, s \in \delta\Omega$. Also, for ease of notation, the authors have defined the variable change $t = T - \tau$. Consequently, we obtain the bounded linear complementarity problem defined on $[0,T] \times \Omega$:

$$\begin{aligned}
&\frac{\partial U}{\partial t} - \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} - rS\frac{\partial U}{\partial S} + rU \geq 0 \\
&\left(\frac{\partial U}{\partial t} - \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} - rS\frac{\partial U}{\partial S} + rU\right)(U - \phi) = 0 \\
&U \geq \phi \\
&\phi = U(0,S) \\
&U = 0 \text{ on } \delta\Omega
\end{aligned}$$

Free boundary problems do not have explicit solutions, and must be solved numerically. Iterative algorithms are particularly well suited for solving such problems, discretizing the PDE in the spatial and temporal dimensions with finite-difference schemes, and solving backwards (forwards with the variable change of $\tau$) for each point in the resulting discrete grid. Finite difference methods replace partial derivatives by approximations stemming from Taylor series expansions of functions around the point or points of interest. We can use different numerical schemes to discretize the PDE, leading to eventual differences in the numerical properties of solution algorithms:

$$\frac{\partial U}{\partial t} \approx \frac{U(S,t+\delta t)-U(S,t)}{\delta t} + O(\delta t)$$
$$\frac{\partial U}{\partial S} \approx \frac{U(S+\delta S,t)-U(S-\delta S,t)}{2\delta S} + O((\delta S)^2)$$
$$\frac{\partial^2 U}{\partial S^2} \approx \frac{U(S+\delta S,t)-2U(S,t)+U(S-\delta S,t)}{2\delta S} + O((\delta S)^2)$$

Furthermore, each point in the mesh is of the form $(n\delta S, m\delta t)$, where the temporal dimension (t) is divided such that T = $M\delta$t, and we set an adequate finite upper bound L on the spatial dimension such that L = $N\delta$S.

$$U_n^m = U(n\delta S, m\delta t)$$

The authors of the replicated paper use the implicit finite difference method,

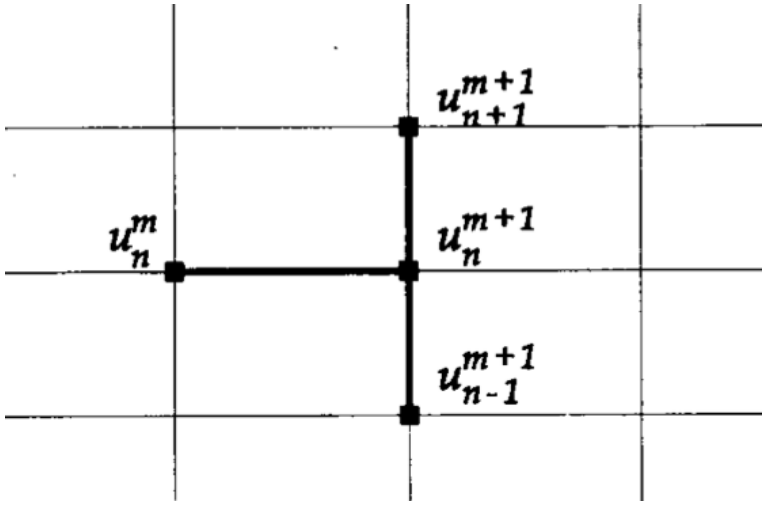


**Figure 1.** Implicit Finite Difference Method

such that $U_n^{m+1}$, $U_{n-1}^{m+1}$, $U_{n+1}^{m+1}$ all depend implicitly on the known value $U_n^m$. Khodja et al. did not include the derivation nor the result of the selected finite difference scheme. We chose to show the derivation for the simple case, and later extend it to the high-dimensional case. At each time step, we obtain the inequality:

$$\frac{\partial U}{\partial t} - \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 U}{\partial S^2} - rS\frac{\partial U}{\partial S} + rU \geq 0$$
$$\frac{U_n^{m+1}-U_n^m}{\delta t} - \frac{1}{2}\sigma^2 (n\delta S)^2 \frac{U_{n+1}^{m+1}-2U_n^{m+1}+U_{n-1}^{m+1}}{\delta S^2} - rn\delta S\frac{U_{n+1}^{m+1}-U_{n-1}^{m+1}}{2\delta S} + rU_n^{m+1} \geq 0$$
$$(1+\delta t(\sigma^2 n^2 + r))U_n^{m+1} - \frac{1}{2}\delta t(\sigma^2 n^2 - rn)U_{n+1}^{m+1} - \frac{1}{2}\delta t(\sigma^2 n^2 + rn)U_{n-1}^{m+1} \geq U_n^m$$
$$A_n U_{n-1}^{m+1} + B_n U_n^{m+1} + C_n U_{n+1}^{m+1} \geq U_n^m$$
$$A_n = -\frac{1}{2}\left(\sigma^2 n^2 - rn\right)\delta t$$
$$B_n = 1 + \left(\sigma^2 n^2 + r\right)\delta t$$
$$C_n = -\frac{1}{2}\left(\sigma^2 n^2 + rn\right)\delta t$$

The above approach might seem to be fully deterministic, a far cry from the probabilistic beginnings of the problem: asset price random walks. However, $U_n^m$ denotes the probability of the option value being at position $n$ at time-step $m$, and the coefficients $A_n, B_n, C_n$ denote the probability of it moving to their respective prices.

The series of inequalities for each time step constitute the discretized bounded linear complementarity problem defined on $t \times \Omega$, a constrained matrix problem:

$$\begin{aligned}
&\mathbf{A}\mathbf{U}^{m+1} \geq \mathbf{U}^m \\
&\left(\mathbf{A}\mathbf{U}^{m+1} - \mathbf{U}^m\right)\left(\mathbf{U}^{m+1} - \Phi\right) = 0 \\
&\mathbf{U}^{m+1} \geq \Phi
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{U}^m &= [U_0^m,\ ...,\ U_N^m]^T \\
\Phi &= \left[U_0^0,\ ...,\ U_N^0\right]^T
\end{aligned}$$

$$\mathbf{A} = \begin{bmatrix}
B_0 & C_0 & 0 & \cdots & 0 \\
A_1 & B_1 & C_1 & & \vdots \\
0 & A_2 & \ddots & \ddots & 0 \\
\vdots & & \ddots & \ddots & C_{N-1} \\
0 & \cdots & 0 & A_N & B_N
\end{bmatrix}$$

Note that the spatial discretization matrix $\mathbf{A}$ is a symmetric M-matrix, also known as a Stieltjes or positive definite matrix. And so, the convergence of synchronous and asynchronous iterative methods is a consequence of the properties of $\mathbf{A}$.

## SEQUENTIAL ITERATIVE METHODS

The constrained matrix problem is associated with the following fixed point mapping, the basis of subsequent iterative solution methods:

$$U_n^{m+1} = \frac{1}{B_n}\left(U_n^m - A_n U_{n-1}^{m+1} - C_n U_{n+1}^{m+1}\right)$$

Where the function is iteratively evaluated until convergence, given certain constraints. Clearly, for each $n$, algorithms must update $A_n, B_n, C_n$. And so, without a formal introduction to the chosen iterative methods, one can see that having fixed coefficients would significantly improve their efficiency.

### A Dimensionless PDE

Through a series of variable changes,

$$S = Ke^x$$
$$\tau = T - t/\frac{1}{2}\sigma^2$$
$$U = Kv(x,\tau)$$
$$k = r/\frac{1}{2}\sigma^2$$
$$v = e^{-\frac{1}{2}(k-1)x-\frac{1}{4}(k+1)^2 t}u(x,t)$$

we can make the equation dimensionless, of the form of the classical obstacle problem, resulting in the linear complementarity form [9]:

$$\left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right)(u(x,t) - g(x,t)) = 0$$
$$\left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right) \geq 0$$
$$(u(x,t) - g(x,t)) \geq 0$$

where $g(x,t)$ is the modified payoff function

The implicit finite-difference method applied to the dimensionless American option problem leads to the inequality:

$$\frac{u_n^{m+1} - u_n^m}{\delta t} + O(\delta t) \geq \frac{u_{n+1}^{m+1} - 2u_n^{m+1} + u_{n-1}^{m+1}}{\delta x^2} + O(\delta x^2)$$
$$-\alpha u_{n-1}^{m+1} + (1+2\alpha)u_n^{m+1} - \alpha u_{n+1}^{m+1} \geq u_n^m$$
$$\alpha = \frac{\delta t}{(\delta x)^2}$$

And the constrained matrix problem:

$$\mathbf{A}\mathbf{U}^{m+1} \geq \mathbf{b}^m$$
$$\left(\mathbf{A}\mathbf{U}^{m+1} - \mathbf{b}^m\right)\left(\mathbf{U}^{m+1} - \mathbf{G}^{\mathbf{m+1}}\right) = 0$$
$$\mathbf{U}^{m+1} \geq \mathbf{G}^{\mathbf{m+1}}$$

where

$$N^- \leq n < N^+$$
$$\mathbf{U}^m = \left[U_{N^-+1}^m,\ ...,\ U_{N+-1}^m\right]^T$$
$$\mathbf{g}^m = \left[g_{N^-+1}^m,\ ...,\ g_{N+-1}^m\right]^T$$
$$\mathbf{b}^m = \mathbf{U}^m + \alpha\left[U_{N^-}^{m+1},\ 0,\ ...,\ 0,\ U_{N+}^{m+1}\right]^T$$
$$\mathbf{A} = \begin{bmatrix} 1+2\alpha & -\alpha & 0 & \cdots & 0 \\ -\alpha & 1+2\alpha & -\alpha & & \vdots \\ 0 & -\alpha & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & -\alpha \\ 0 & \cdots & 0 & 1+2\alpha & -\alpha \end{bmatrix}$$

Note that in making the Black-Scholes equation dimensionless, we have transformed a convection-diffusion equation into a diffusion equation. The characteristics of the associated operator play a major role in determining which algorithm is appropriate for the numerical solution of the resulting algebraic systems to solve.

### Projected Successive Over-Relaxation

Traditionally, we solve the above constrained matrix problem using iterative methods, namely the projected SOR method. Thus, the algorithm for time stepping from $\mathbf{u}^m$ to $\mathbf{u}^{m+1}$ is as follows [9]:

Let the vector $u^{m+1,k} = \left(u_{N-1,1}^{m+1,k}, \ldots, u_{N+-1}^{m+1,k}\right)$ denote the $k$-th iterate of the algorithm at the $(m+1)$-st time-step. (Thus, at time-step $m+1$, we start with the initial guess $u^{m+1,0}$ and, as we apply the projected SOR algorithm, we generate $u^{m+1,k+1}$ from $u^{m+1,k}$. We know that $u^{m+1,k} \to u^{m+1}$ as $k \to \infty$.)

1. Given $u^m$, first form the vector $b^m$, and calculate the constraint vector $g^{m+1}$.
2. Start with the initial guess $u^{m+1,0} = \max\left(u^m, g^{m+1}\right)$, that is, $u_n^{m+1,0} = \max\left(u_n^m, g_n^{m+1}\right)$.
3. In increasing $n$-indicial order, first form the quantity $y_n^{k+1}$ by

$$y_n^{m+1,k+1} = \frac{1}{1+2\alpha}\left(b_n^m + \alpha\left(u_{n-1}^{m+1,k} + u_{n+1}^{m+1,k}\right)\right)$$

   Then generate $u_n^{m+1,k+1}$ using

$$u_n^{m+1,k+1} = \max\left(g_n^{m+1}, u_n^{m+1,k} + \omega\left(y_n^{m+1,k+1} - u_n^{m+1,k}\right)\right)$$

   where $1 < \omega < 2$ is the over-relaxation parameter.
4. Test whether or not $\left\|u^{m+1,k+1} - u^{m+1,k}\right\|$ is smaller than a prechosen tolerance $\varepsilon$, that is, test whether

$$\sum_n \left(u_n^{m+1,k+1} - u_n^{m+1,k}\right)^2 \leq \varepsilon^2$$

   If it is, go on to step (5). If it is not, go back to step (3) and repeat the process with $k$ replaced by $k+1$.
5. When the vectors $u^{m+1,k}$ have converged to the required tolerance, put $u^{m+1} = u^{m+1,k+1}$
6. Return to step 1 until the necessary number of time-steps have been completed.

Classical PSOR is implemented as a point based iteration and uses the vector updates of Jacobi's method. However, we can accelerate the method's convergence with Gauss-Seidel vector updates, a refinement of Jacobi's method stemming from the realization that at iteration $n$, we have access to $u_{n-1}^{m+1,k+1}$. Consequently, we can replace $u_{n-1}^{m+1,k}$ with $u_{n-1}^{m+1,k+1}$ in step 3 to increase the rate of convergence of the algorithm.

## TOWARDS PARALLELIZATION

Clearly, simultaneously conducting relaxation step $k$ for all elements in the solution vector $\mathbf{U}$ would lead to a significant increase in algorithm performance. However, Gauss-Seidel vector updates do impose a sequential element to a parallel version of the algorithm, potentially decreasing its performance. Indeed, Khodja et al. have found that increased convergence speed does not compensate for the added limitation on thread-parallelization of the chosen algorithm.

# Parallel Iterative Methods

In the following part, we present a general framework of parallelizing iterative algorithms based on a fixed point mapping $F$ [5].

## GENERAL FRAMEWORK

Let $\alpha$ be a positive integer, $E = \mathbb{R}^M$ such that $E = \prod_{i=1}^{\alpha} E_i$ where $\sum_{i=1}^{\alpha} m_i = M$. Let $W \in E$ and consider the following block decomposition of $W$ into non-overlapping subdomains and the corresponding decomposition of $F$

$$\begin{aligned} &W = (W_1, \ldots, W_a) \\ &F(W) = (F_1(W), \ldots, F_\alpha(W)) . \end{aligned}$$

Also, Consider the general following fixed point problem

$$\begin{cases} \text{Find } W^* \in E \text{ such that} \\ W^* = F(W^*) \end{cases}$$

Consider the parallel asynchronous iterations, as a solution to the fixed point problem, defined as follows: let $W^0 \in E$ be given, $\forall p \in N, W^{p+1}$ is recursively defined by

$$W_i^{p+1} = \begin{cases} F_i\left(W_1^{\rho_1(p)}, \ldots, W_j^{\rho_j(p)}, \ldots, W_\alpha^{\rho_\alpha(p)}\right) & \text{if } i \in s(p) \\ W_i^p \text{ if } i \notin s(p) \end{cases}$$

where

$$\begin{cases} \forall p \in \mathbb{N}, \quad s(p) \subset \{1, \ldots, \alpha\} \text{ and } s(p) \neq \emptyset \\ \forall i \in \{1, \ldots, \alpha\}, \quad \{p \mid i \in s(p)\} \text{ is countable} \end{cases}$$

and $\forall j \in \{1, \ldots, \alpha\}$

$$\begin{cases} \forall p \in \mathbb{N},\ \rho_j(p) \in \mathbb{N},\ 0 \leq \rho_j(p) \leq p \ \text{ and } \ \rho_j(p) = p \ \text{ if } j \in s(p) \\ \lim_{p \to \infty} \rho_j(p) = +\infty \end{cases}$$

The previous asynchronous iterative scheme enables us to consider distributed computations whereby processors compute at their own pace according to their intrinsic characteristics and computational load. The parallelism between the processors is well described by $s(p)$ which contains at each step $p$ the index of the components relaxed by each processor in a parallel way while the use of delayed components permits one to model nondeterministic behavior [5].

To further bridge the apparent gap between the previously described sequential algorithms and this framework, let $s(p) = \{1, \ldots, \alpha\})\ \forall p \in \mathbb{N}$. The resulting scheme is a sequential iterative method using Jacobi's method vector updates. Also, if $s(p) = \{1 + p \bmod \alpha\}\ \forall p \in \mathbb{N}$, the resulting scheme is a sequential iterative method using Gauss-Seidel's method vector updates.

# High-Dimensional Options

The one-dimensional Black–Scholes PDE admits a natural extension when it comes to pricing options on several underlying assets, which is usually referred to as the multidimensional Black–Scholes equation [3].

$$\frac{\partial U}{\partial t} - \frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}\rho_{ij}\sigma_i\sigma_j S_i S_j \frac{\partial^2 U}{\partial S_i \partial S_j} - r\sum_{i=1}^{d} S_i \frac{\partial U}{\partial S_i} + rU \geq 0$$

## TOWARDS A NUMERICAL SOLUTION

In the interest of simplicity, we will assume that the underlying assets are independent, hence eliminating the cross-derivatives

$$\frac{\partial U}{\partial t} - \frac{1}{2}\sum_{i=1}^{d}\sigma_i^2 S_i^2 \frac{\partial^2 U}{\partial S_i^2} - r\sum_{i=1}^{d} S_i \frac{\partial U}{\partial S_i} + rU \geq 0$$

And so, following the same steps as in the one dimensional case to discretize the PDE using an implicit time-marching scheme, we obtain the following inequality

$$\begin{aligned}
&Center \times U^{m+1}(x,y,z) + West \times U^{m+1}(x-h,y,z) + \\
&\quad East \times U^{m+1}(x+h,y,z) + South \times U^{m+1}(x,y-h,z) + \\
&North \times U^{m+1}(x,y+h,z) + Rear \times U^{m+1}(x,y,z-h) + \\
&Front \times U^{m+1}(x,y,z+h) \geq U^{m+1}(x,y,z) \\
&Center = 1 + \delta t \left( \sum_{i\in\{x,y,z\}} \sigma_i^2 i^2 + r \right) \\
&West = -\frac{1}{2}\left(\sigma_x^2 x^2 + rx\right)\delta t \\
&East = -\frac{1}{2}\left(\sigma_x^2 x^2 - rx\right)\delta t \\
&South = -\frac{1}{2}\left(\sigma_y^2 y^2 + ry\right)\delta t \\
&North = -\frac{1}{2}\left(\sigma_y^2 y^2 - ry\right)\delta t \\
&Rear = -\frac{1}{2}\left(\sigma_z^2 z^2 + rz\right)\delta t \\
&Front = -\frac{1}{2}\left(\sigma_z^2 z^2 - rz\right)\delta t
\end{aligned}$$

where the aptly named coefficients correspond to their positions in the discretized spatial matrix $\mathbf{A}$ and $h$ is the chosen spatial time-step.

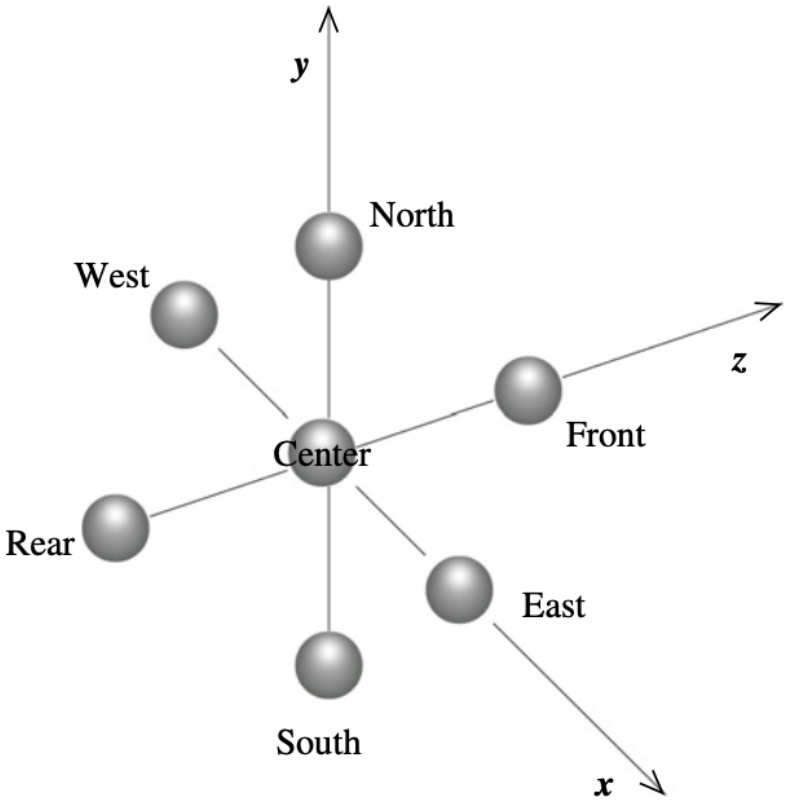


**Figure 2.** Discretized Spatial Matrix

### A Dimensionless PDE

As previously discussed, there are clear computational advantages to making the PDE dimensionless. The inequality can be turned into the standard, separable diffusion equation of order $n \in \mathbb{N}$, enabling the use of the projected Richardson iterative method,

$$\frac{\partial u}{\partial t} = \frac{1}{2}\sum_{i=1}^{n} \sigma_i^2 \frac{\partial^2 u}{\partial x_i^2}$$

with the following variable changes [3]

$$x_i = ln\left(\frac{S_i(t)}{S_i(0)}\right)$$
$$y_i = \frac{x_i}{\sigma_i}$$
$$U(x_1,\ldots,x_n,t) = exp\left(\sum_{i=1}^{n} a_i x_i\right) u(x_1,\ldots,x_n,t)$$
$$a_i = \frac{-(r-\sigma_i^2/2)\Pi_{j=1,j\neq i}^{n}\sigma_i}{\sigma_i^2 \times \Pi_{i=1,j\neq i}^{n}\sigma_i}$$

The implicit finite-difference method applied to the dimensionless American option problem leads to the inequality:

$$
\begin{aligned}
&Center \times u^{m+1}(x,y,z) + West \times u^{m+1}(x-h,y,z) + \\
&\quad East \times u^{m+1}(x+h,y,z) + South \times u^{m+1}(x,y-h,z) + \\
&\quad North \times u^{m+1}(x,y+h,z) + Rear \times u^{m+1}(x,y,z-h) + \\
&\quad Front \times u^{m+1}(x,y,z+h) \geq u^{m+1}(x,y,z) \\
&Center = 1+3\alpha \\
&\quad West = -\frac{1}{2}\alpha \\
&\quad East = -\frac{1}{2}\alpha \\
&\quad South = -\frac{1}{2}\alpha \\
&\quad North = -\frac{1}{2}\alpha \\
&\quad Rear = -\frac{1}{2}\alpha \\
&\quad Front = -\frac{1}{2}\alpha \\
&\qquad \alpha = \frac{\delta t}{h^2}
\end{aligned}
$$

The series of inequalities for each time step constitute the discretized bounded linear complementarity problem defined on $t \times \Omega$ ($\Omega \subset \mathbb{R}^n$), a constrained matrix problem, as in the one dimensional case:

$$
\begin{aligned}
&\mathbf{A}\mathbf{u}^{m+1} \geq \mathbf{u}^m \\
&\left(\mathbf{A}\mathbf{u}^{m+1} - \mathbf{u}^m\right)\left(\mathbf{u}^{m+1} - \Phi\right) = 0 \\
&\mathbf{u}^{m+1} \geq \Phi
\end{aligned}
$$

Note that all derivations are similar to those of the one underlying asset case, albeit involving more algebraic manipulations.

## ALGORITHMS

The iterations of the projected Richardson method, based on Jacobi's method, in a three-dimensional spatial domain are defined as follows [5]

$$
\begin{aligned}
u^{p+1}(x,y,z) = \tfrac{1}{\text{Center}}\,(g(x,y,z) - (\ &\text{Center}\cdot u^p(x,y,z) + \text{West}\cdot u^p(x-h,y,z) + \text{East}\cdot u^p(x+h,y,z) \\
&+ \text{South}\cdot u^p(x,y-h,z) + \text{North}\cdot u^p(x,y+h,z) + \text{Rear}\cdot u^p(x,y,z-h) + \text{Front}\cdot u^p(x,y,z+h)))
\end{aligned}
$$

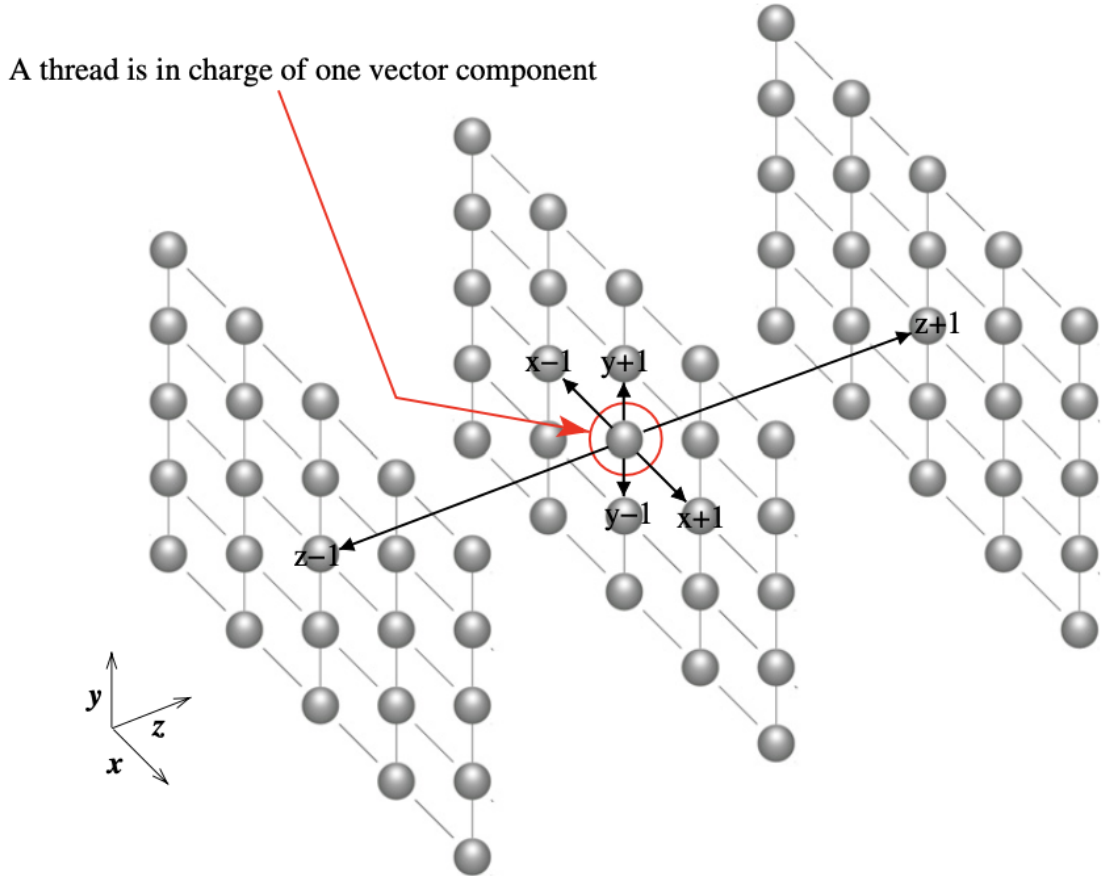


**Figure 3.** Projected Richardson Relaxation Visualization

And the iterative solution method is akin to the afore-described parallelized PSOR, using the above equation in step 3 for the computation of $y^{m+1,k+1}$. Below are the driver functions of the solution method (1, 2) where the problem is initialized, the loop invariant resides, and the kernels(4) are launched to compute the matrix-vector multiplication $AU$ and update the vector components of the solution vector $U$.

**Algorithm 1** Algorithm for solving nonlinear systems of the obstacle problem on GPUs

**Input**: $A$ (matrix), $G$ (right-hand side), $\varepsilon$ (error tolerance threshold), *MaxRelax* (maximum number of relaxations), *NbSteps* (number of time steps)
**Output**: $U$ (solution vector)
Initialization of the parameters of the obstacle problem
Allocate and fill the data in the global memory GPU
**for** $i \leftarrow 1, \text{NbSteps}$ **do**
  $G \leftarrow \frac{1}{k}U + F$
  *Solve(A, U, G, ε, MaxRelax)*
**end for**
Copy results back from GPU memory

All data is first initialized in the CPU and transferred to the GPU using the CUBLAS library, namely `cublasAlloc()` for memory allocations in the GPU, and `cublasSetVector()`

/ `cublasGetVector()` to transfer data to/from the GPU.

**Algorithm 2** A global algorithm of the iterative solvers

**Input**: $A$ (matrix), $G$ (right-hand side), $\varepsilon$ (error tolerance threshold), *MaxRelax* (maximum number of relaxations)
**Output**: $U$ (solution vector)
$p \leftarrow 0$
**repeat**
  $tmp \leftarrow U$
  *Computation_New_Vector_Components(A, G, U)*
  $tmp \leftarrow tmp - U$
  $\rho \leftarrow ||tmp||_2$
  $p \leftarrow p + 1$
**until** $(p < \varepsilon)$ **or** $(p \geq MaxRelax)$

In addition, we use other CUBLAS library functions to implement vector operations on the GPU in Algorithm 2:

- `cublasDaxpy()` to compute the error vector (line 5)
- `cublasDnrm2()` to compute the Euclidean norm of the error vector (line 6)

### Parameters

Since Khodja et al. did not include the derivation and results of the discretization of **A**, we cannot exactly mimic their experiments. Also, we have access to more powerful GPUs, Tesla V100s vs Tesla C1060, enabling us to increase the thread block size from 512 to 1024. However, we have preserved all remaining explicitly set parameters:

- $\sigma_X^2 = \sigma_Y^2 = \sigma_Z^2 = 0.4$
- $r = 1.1\%$
- $u^0(x,y,z) = 0$
- $\varepsilon = 10^{-4}$
- *MaxRelax* = $10^6$
- Sizes of the discretized obstacle problems $NX \times NY \times NZ \in \{32^3, 64^3, 128^3, 256^3\}$
- $Blocks = \frac{NX \times NY \times NZ + Threads - 1}{Threads}$, the number of required thread blocks

## CODE

```
/* Kernel of the matrix-vector multiplication */
__global__ void MV_Multiplication (int n, double* U, double* Y)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x; //thread ID
    double sum;
    if(tid < n){
        int x = tid % NX;           //x-coordinate
        int y = (tid / NX) % NY;    //y-coordinate
        int z = tid / (NX * NY);    //z-coordinate
        sum = Center * U[tid];
        if(x != 0)          sum += West * U[tid-1];
        if(x != (NX-1))     sum += East * U[tid+1];
        if(y != 0)          sum += South * U[tid-NX];
        if(y != (NY-1))     sum += North * U[tid+NX];
        if(z != 0)          sum += Rear  * U[tid-NX*NY];
        if(z != (NZ-1))     sum += Front * U[tid+NX*NY];
        Y[tid] = sum;
    }
}

/* Kernel of the vector components updates */
__global__ void Vector_Updates(int n, double* G, double* Y, double* U)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x; //thread ID
    double var;
    if(tid < n){
        var = (G[tid] - Y[tid]) / Center + U[tid];
        if(var < 0) var = 0; //projection
        U[tid] = var;
    }
}

/* Function to be executed by the CPU */
void Computation_New_Vector_Components(double* A, double* G, double* U)
{
    cudaError_t cuda_ret;

    int nnn = nnn_h;     //number of vector elements
    int Threads = 1024;          //size of a thread block
    int Blocks = (nnn + Threads - 1) / Threads; //number of thread blocks
    double* Y;

    //Matrix coefficients filled in the constant memory:
    //Center, West, East, South, North, Rear, Front

    //vector Elements of U are filled in the texture memory
    //Allocate a GPU memory space for the vector Y

    cuda_ret = cudaMalloc((void**) &Y, sizeof(double)*nnn );
    if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

    MV_Multiplication<<<Blocks,Threads>>>(nnn, U, Y);
    Vector_Updates<<<Blocks,Threads>>>(nnn, G, Y, U);
}
```

**Figure 4.** Implementation of the Kernels

```
//// Algorithm 1, Step 1:
// Initialization of the parameters of the obstacle problem ================

printf("\nSetting up the problem..."); fflush(stdout);
startTime(&timer);
int nnn = nnn_h;

unsigned int MaxRelax;
unsigned int NbSteps;
double epsilon;
if(argc == 1) {
    MaxRelax = 1000000;
    NbSteps  = 1000;
    epsilon  = 1.0/10000;
} else if(argc == 4) {
    MaxRelax = atoi(argv[1]);
    NbSteps  = atoi(argv[2]);
    epsilon  = atoi(argv[3]);
} else {
    printf("\n    Invalid input parameters!"
       "\n    Usage: ./richardson # Default: MaxRelax = 1000000, NbSteps = 1000, epsilon = 10^-4"
       "\n    Usage: ./richardson <MaxRelax> <Nbsteps> <epsilon>"
       "\n");
    exit(0);
}

double* A_h = (double*) malloc( sizeof(double)*nnn );
for (unsigned int i=0; i < nnn; i++) { A_h[i] = (rand()%100)/100.00; }

double* G_h = (double*) malloc( sizeof(double)*nnn );
for (unsigned int i=0; i < nnn; i++) { G_h[i] = (rand()%100)/100.00; }

double* U_h = (double*) malloc( sizeof(double)*nnn );
for (unsigned int i=0; i < nnn; i++) { U_h[i] = 0.0; }

stopTime(&timer); printf("%f s\n", elapsedTime(timer));
printf("Matrix size (unrolled) = %u\n", nnn);

//// Algorithm 1, Step 2:
// Allocate and fill the data in the global memory GPU ====================

printf("Allocating device variables..."); fflush(stdout);
startTime(&timer);

double* A_d;
cuda_ret = cudaMalloc((void**) &A_d, sizeof(double)*nnn );
if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

double* G_d;
cuda_ret = cudaMalloc((void**) &G_d, sizeof(double)*nnn );
if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

double* U_d;
cuda_ret = cudaMalloc((void**) &U_d, sizeof(double)*nnn );
if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

cudaDeviceSynchronize();

stopTime(&timer); printf("%f s\n", elapsedTime(timer));

// Copy host variables to device ------------------------------------------

printf("Copying data from host to device..."); fflush(stdout);
startTime(&timer);

cuda_ret = cudaMemcpy(A_d, A_h, sizeof(double)*nnn , cudaMemcpyHostToDevice);
if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to device");

cuda_ret = cudaMemcpy(G_d, G_h, sizeof(double)*nnn , cudaMemcpyHostToDevice);
if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to device");

cudaDeviceSynchronize();
stopTime(&timer); printf("%f s\n", elapsedTime(timer));

//// Algorithm 1, Step 3:
// for (i = 1) to NbSteps do ----------------------------------------------

double k = 0.0066;

for (unsigned int step=1; step < NbSteps; step++) {
    for (unsigned int i=0; i < nnn; i++) {
        //// Algorithm 1, Step 4:
        // G <== U/k + F
        G_h[i] = U_h[i]/k;

        //// Algorithm 1, Step 4:
        // Solve(A, U, G, epsilon, MaxRelax)
        Solve(A_d, U_d, G_d, epsilon, MaxRelax);
    }
}

// Copy device variables from host ----------------------------------------

printf("Copying data from device to host..."); fflush(stdout);
startTime(&timer);

cuda_ret = cudaMemcpy(U_h, U_d, sizeof(double)*nnn, cudaMemcpyDeviceToHost);
if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to host");

cudaDeviceSynchronize();
stopTime(&timer); printf("%f s\n", elapsedTime(timer));

// Free memory ------------------------------------------------------------

free(A_h);
free(G_h);
free(U_h);

cudaFree(A_d);
cudaFree(G_d);
cudaFree(U_d);

return 0;
```

**Figure 5.** Implementation of Algorithm 1

```
//// Algorithm 2: A global algorithm of the iterative solvers
void Solve(double* A_d, double* U_d, double* G_d, double epsilon, int MaxRelax)
{
    int nnn = nnn_h;

    cudaError_t cuda_ret;
    cublasHandle_t handle;
    cublasStatus_t stat;
    stat = cublasCreate(&handle);
    if(stat != CUBLAS_STATUS_SUCCESS) FATAL("CUBLAS initialization failed");

    // Allocate space for temporary U matrix
    double* U_temp_d;
    cuda_ret = cudaMalloc((void**) &U_temp_d, sizeof(double)*nnn );
    if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

    double* U_ptr_d;
    cuda_ret = cudaMalloc((void**) &U_ptr_d, sizeof(double)*nnn );
    if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

    double* rho_h;
    double* eps = &epsilon;
    double a = -1.0;
    const double* alpha = &a;
    //// Algorithm 2, Step 1-2, 7:
    // p <== 0;
    // repeat
    // p <== p + 1;

    //// Algorithm 2, Step 8:
    // until (p >= MaxRelax)
    for (int p = 0; p < MaxRelax; p++) {

        //// Algorithm 2, Step 3:
        // tmp <== U;
        cuda_ret = cudaMemcpy(U_temp_d, U_d, sizeof(double)*nnn , cudaMemcpyDeviceToDevice);
        if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to device");

        //// Algorithm 2, Step 4:
        // Computation New Vector Components(A, G, U);
        Computation_New_Vector_Components(A_d, G_d, U_d);

        //// Algorithm 2, Step 5:
        // tmp <== tmp - U;
        stat = cublasDaxpy(handle, nnn, alpha, U_d, 1, U_temp_d, 1);
        if(stat != CUBLAS_STATUS_SUCCESS) FATAL("CUBLAS Daxpy failed");

        //// Algorithm 2, Step 6:
        // rho <== L2Norm(tmp);
        stat = cublasDnrm2(handle, nnn, U_temp_d, 1, rho_h);
        if(stat != CUBLAS_STATUS_SUCCESS) FATAL("CUBLAS Dnrm2 failed");

        //// Algorithm 2, Step 8:
        // until (rho < epsilon)
        if (rho_h < eps) {
            break;
        }
    }

    // Destroy cublas handle
    stat = cublasDestroy(handle);
    if(stat != CUBLAS_STATUS_SUCCESS) FATAL("CUBLAS destruction failed");
}
```

**Figure 6.** Implementation of Algorithm 2

## RESULTS

The implementation of the iterative Richardson algorithm achieved two key objectives. One, the solutions converged within the maximum number of relaxations. Two, the algorithm remained stable under random initial conditions. An analysis of the runtimes under different discretization steps shows an expected performance relationship. Memory allocation carries a constant overhead, while initialization of parameters and copying between host and device is proportional to the memory requirements.

Notably, the time taken in the iterative solver, where parallelization of the computations occur, does not scale linearly with the total memory requirements. Considering the limitations of our implementation, which is limited to a single GPU, we may have exceeded the number of threads available in a single GPU.

| Time (s) | 32 Steps | 64 Steps | 128 Steps | 256 Steps |
|---|---|---|---|---|
| Initialization | 0.001217 | 0.010099 | 0.075061 | 0.591463 |
| Malloc | 0.123339 | 0.131646 | 0.122630 | 0.123486 |
| Host-Dev Copy | 0.000200 | 0.001079 | 0.007195 | 0.056455 |
| Iterative Solver | 0.216316 | 0.285389 | 0.728330 | 6.869373 |
| Dev-Host Copy | 0.000068 | 0.000332 | 0.003257 | 0.027039 |
| Total | 0.341140 | 0.428545 | 0.936473 | 7.667816 |

# Conclusion

The report is a complete example of the derivation and implementation of parallel solution methods for high-dimensional PDE's, starting at the one dimensional instance, and extending the problem to the multi-dimensional instance.

We detail the necessary steps to obtain the fixed point problem associated with the constrained matrix problems resulting from the discretization of the PDEs. Then, we describe two sequential solution algorithms, elaborating on how their different characteristics may affect the effectiveness of their parallelization. Moreover, we formally present the framework to parallelize iterative methods, showing how the sequential algorithms are a special instance of the framework. Afterwards, we adapt it to the high-dimensional American option pricing problem, subsequently providing code to implement the parallel projected Richardson method on GPUs in CUDA C.

Therefore, the convergence of the algorithm on the GPU supports the correctness of our derivations and implementation.

## FUTURE CONSIDERATIONS

Khodja et al. have made use of more advanced memory structures, namely texture memory, a form of rapid read-only memory. Since they wrote the paper, new versions of CUDA provide access to surface memory, a development of texture memory with rapid read and write access. Future iterations of our implementation should consider the use of surface memory. Also, to enable running the code on multiple GPUs, one must use the MPI library, another potential improvement on the code.

## REFERENCES


[1] Gaikwad, A. and Toke, I. M. (2010). Parallel iterative linear solvers on gpu: A financial engineering case. *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*.

[2] Ghosh, A. and Mishra, C. (2022). High-performance computation of pricing two-asset american options under the merton jump-diffusion model on a gpu. *Computers Mathematics with Applications*, 105:29–40.

[3] Guillaume, T. (2019). On the multidimensional black–scholes partial differential equation. *Annals of Operations Research*, 281(1–2):229–251.

[4] Jaillet, P., Lamberton, D., and Lapeyre, B. (1990). Variational inequalities and the pricing of american options. *Acta Applicandae Mathematicae*, 21(3):263–289.

[5] Khodja, L. Z., Chau, M., Couturier, R., Bahi, J., and Spitéri, P. (2013). Parallel solution of american option derivatives on gpu clusters. *Computers Mathematics with Applications*, 65(11):1830–1848.

[6] Mahony, A. O., Zeidan, G., Hanzon, B., and Popovici, E. (2022). A parallel and pipelined implementation of a pascal-simplex based multi-asset option pricer on fpga using opencl. *Microprocessors and Microsystems*, 90:104508.

[7] Maringanti, A., Athavale, V., and Patkar, S. B. (2009). Acceleration of conjugate gradient method for circuit simulation using cuda. pages 438–444.

[8] Pagès, G., Pironneau, O., and Sall, G. (2016). The parareal algorithm for american options. *Comptes Rendus Mathematique*, 354(11):1132–1138.

[9] Wilmott, P., Howison, S., and Dewynne, J. (2009). *The Mathematics of Financial Derivatives*. Cambridge University Press, New York.

[10] Zhao, Y. (2008). Lattice boltzmann based pde solver on the gpu. *The Visual Computer*, 24(5):323–333.