A PARALLEL IMPLEMENTATION ON GPUs OF ADI FINITE DIFFERENCE METHODS FOR PARABOLIC PDES WITH APPLICATIONS IN FINANCE

DUY MINH DANG, CHRISTINA C. CHRISTARA AND KENNETH R. JACKSON

ABSTRACT. We study a parallel implementation on a Graphics Processing Unit (GPU) of Alternating Direction Implicit (ADI) time-discretization methods for solving time-dependent parabolic Partial Differential Equations (PDEs) in three spatial dimensions with mixed spatial derivatives in a variety of applications in computational finance. Finite differences on uniform grids are used for the spatial discretization of the PDEs. As examples, we apply the GPU-based parallel methods to price European rainbow and European basket options, each written on three assets. Numerical results showing the efficiency of the parallel methods are provided.

1 Introduction

Closed-form solutions, such as the Black-Scholes formula [4] for vanilla European put and call options, are not available for most financial derivatives. Hence such derivatives must be priced by numerical techniques. Although several pricing approaches can be used, such as Monte Carlo simulations or tree (lattice) methods, for problems in low dimensions, i.e., less than five dimensions, the Partial Differential Equation (PDE) approach is a very popular choice, due to its efficiency and global character. In addition, accurate hedging parameters, such as delta and gamma, which are essential for risk-management of the financial derivatives, are generally much easier to compute via a PDE approach than via other methods.

In recent years, the complexity of the financial derivatives has increased significantly. For instance, option contracts can have more than

Keywords: Alternating Direction Implicit, ADI, Partial Differential Equation, PDE, Graphics Processing Units, GPUs, parallel computing, finite difference, multi-asset options.

Copyright © Applied Mathematics Institute, University of Alberta.
one underlying asset and different types of payoff functions. When solving the associated pricing problems by the PDE approach, each stochastic factor in the model gives rise to a spatial variable in the PDE. Due to the “curse of dimensionality” associated with high-dimensional PDEs, the pricing of such derivatives via the PDE approach is challenging. Moreover, when stochastic processes in the pricing model are correlated, as is common in financial modeling, the resulting PDE possesses mixed spatial derivatives, which makes solving the associate problems numerically even more challenging.

For the numerical solution of low-dimensional PDE models in finance, the valuation of the securities can be efficiently calculated by utilizing a level-splitting scheme of the Alternating Direction Implicit (ADI) type along the time-dimension, the computation of which requires the solution of a sequence of tridiagonal linear systems at each timestep. Examples of applications of ADI schemes in finance can be found in [3, 12, 19, 25, 34]. Among them, the most popular are perhaps the ADI schemes proposed by Craig and Sneyd in [6] and by Hundsdorfer and Verwer [16, 17], because they can handle effectively mixed spatial derivatives. More specifically, these two schemes, when combined with second-order central finite differences for the discretization of the space variables, are unconditionally stable and efficient second-order methods in both space and time when applied to PDEs with mixed derivative terms. However, a disadvantage of the Craig and Sneyd scheme is that it cannot maintain both unconditional stability and second-order accuracy when the number of spatial dimensions is greater than three [6, 18], which potentially prevents extending the method to higher-dimensional applications. In addition, it has been noted in [19] that the Craig and Sneyd scheme may exhibit undesirable convergence behavior when the payoff functions are non-smooth, which is quite common for financial applications. Hence smoothing techniques, such as Rannacher timestepping [30], may be required. On the other hand, the ADI scheme introduced by Hundsdorfer and Verwer is unconditionally stable for arbitrary spatial dimensions [18], and, at the same time, also effectively damps the errors introduced by non-smooth payoff functions [19]. It is worth noting that classical ADI algorithms, such as the Douglas and Rachford scheme [11], although unconditionally stable, are only first-order in time and second-order in space when mixed spatial derivatives are present.

Over the last few years, the rapid evolution of Graphics Processing Units (GPUs) into powerful, cost-efficient, programmable computing architectures for general purpose computations has provided application potential beyond the primary purpose of graphics processing. In com-
putational finance, there has been great interest in utilizing GPUs in developing efficient pricing architectures for computationally intensive problems (e.g., [1, 2, 36]). However, the applications mostly focus on option pricing and Monte Carlo simulations (e.g., [1, 2, 28]). The literature on GPU-based PDE methods for pricing basket options is rather sparse, with scattered work presented at conferences or workshops [14]. This shortcoming motivated our work. In addition, to the best of our knowledge, an efficient GPU-based parallelization of an unconditionally stable second-order ADI scheme that can handle effectively mixed spatial derivatives, such as the scheme proposed by Hundsdorfer and Verwer [17], and its applications in computational finance have not been previously discussed in the literature. This deficiency further motivated our work.

This paper discusses the application of GPUs to solve time-dependent parabolic PDEs in three spatial dimensions with mixed spatial derivatives via a parallelization of an ADI scheme, and investigates the performance of the resulting parallel methods in several applications in computational finance. We use the parallel architectural features of GPUs together with the Compute Unified Device Architecture (CUDA) framework on a NVIDIA Tesla T10 GPU to design and implement an efficient GPU-based parallel algorithm for solving such PDEs. The main components of our GPU-based parallelization of the ADI scheme are (i) an efficient parallel implementation of the explicit Euler predictor step, and (ii) a parallel solver for the independent tridiagonal systems arising in the three implicit, but unidirectional, corrector steps. We focus on the ADI scheme introduced by Hundsdorfer and Verwer [17], due to its favorable characteristics; however, the parallelization method presented in this paper can be easily tailored for other ADI schemes. As applications, we apply the GPU-based parallel methods to price European rainbow and European basket options, each written on three assets. The main motivation for the choice of basket asset options comes from the attractiveness of those options for risk-management, being both (i) cheaper than a portfolio of standard options written on the same underlying assets and (ii) more easily managed than a portfolio of standard options. The results of this paper demonstrate the efficiency of the parallel numerical methods and show that GPUs can provide a significant increase in performance over CPUs when pricing complex financial derivatives. Although we focus on a three-factor model, many of the ideas and results in this paper can be naturally extended to higher-dimensional applications.

The remainder of this paper is organized as follows. Section 2 de-
scribes the model time-dependent parabolic PDE in three space dimensions and the discretization schemes for the problem. A Finite Difference (FD) method on a uniform grid is employed for the spatial discretization of the PDE and the ADI technique introduced by Hundsdorfer and Verwer [17] is used for its time discretization. In Section 3, a brief overview of the GPU architecture and some programming basics required to understand our methods are provided. A GPU-based parallel implementation of the methods is discussed in Section 4. Descriptions of application problems in computational finance are provided in Section 5. Numerical results and related discussions are presented in Section 6. Section 7 concludes the paper and outlines possible future work.

2 The model problem and discretization

2.1 The model PDE

In finance, the time-dependent PDE, which governs the values of the financial instruments, derived from a stochastic pricing model is expressed in terms of the forward time variable, denoted by \( t \). The known payoff function at the maturity time \( T \) of the instruments yields the appropriate terminal condition. Since we solve the pricing PDE backward in time from \( T \) to the current time \( t \), the change of variable \( \tau = T - t \), where \( \tau \) denotes the time to expiry, is often used. The resulting PDE is then solved forward in \( \tau \), with the known payoff function being its initial condition. For consistency purposes, we present the model PDE and numerical methods in terms of the variable \( \tau \), while the financial examples in Section 5 are presented in terms of \( t \).

The price \( u(x_1, x_2, x_3, \tau) \) of each of the financial derivatives considered in this paper satisfies a parabolic PDE of the form

\[
\frac{\partial u}{\partial \tau} = \mathcal{L}u(x_1, x_2, x_3, \tau) = \sum_{i,j=1}^{3} c_{ij} \frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_{i=1}^{3} c_i \frac{\partial u}{\partial x_i} + c_0 u
\]

in \( \Omega \times (0, T] \) subject to the initial condition

\[
u(x_1, x_2, x_3, 0) = u_0(x_1, x_2, x_3) \quad \text{on} \quad (\partial \Omega \cup \Omega) \times \{0\}
\]

and appropriate boundary conditions (discussed below), where \( \Omega \subset \mathbb{R}^3 \) is the spatial domain; \( \partial \Omega \) is the boundary of \( \Omega \); \( (0, T] \) with \( T > 0 \) is the time domain; \( c_{ij}, i, j = 1, \ldots, 3 \), \( c_i, i = 0, \ldots, 3 \), are given functions of \( x_1, x_2, x_3 \) and/or \( \tau \). For the PDE (1) to be well-posed, we assume that the coefficient matrix \( [c_{i,j}]_{1 \leq i, j \leq 3} \) of the diffusion terms is symmetric positive semi-definite.
In many applications in finance, $\Omega$ is an infinite domain. However, to apply a finite difference method to solve (1) numerically, we must truncate $\Omega$ to a finite domain. We assume in this paper that this has already been done and that $\Omega = [L_{x_1}, U_{x_1}] \times [L_{x_2}, U_{x_2}] \times [L_{x_3}, U_{x_3}] \subset \mathbb{R}^3$ is a finite rectangular spatial domain. If the domain is truncated, it must be sufficiently large so that the boundary conditions on the truncated sides do not significantly affect the solution $u$ in the region of interest [35].

For many financial derivatives, $u(x_1, x_2, x_3, \tau)$ is approximately linear in $x_i$ at the boundaries $x_i = L_{x_i}$ and $x_i = U_{x_i}$ for $i = 1, 2, 3$. Consequently,

$$\frac{\partial^2 u}{\partial x_i^2} \bigg|_{L_{x_i}} = \frac{\partial^2 u}{\partial x_i^2} \bigg|_{U_{x_i}} = 0$$

for $i = 1, 2, 3$. For example, it can be verified easily from the Black-Scholes formula that vanilla European put and call options satisfy (3). All the options considered in this paper satisfy (3) also.

We can use (3) to derive so-called linear boundary conditions, a commonly used boundary condition in computational finance (see, for example, [23, 34, 35, 12, 21]). To derive one such linear boundary condition, suppose that $x_1 = L_{x_1}$ or $x_1 = U_{x_1}$, but that $x_2 \in (L_{x_2}, U_{x_2})$ and $x_3 \in (L_{x_3}, U_{x_3})$. Then, using (3), the PDE (1) reduces to

$$\frac{\partial u}{\partial \tau} = \sum_{i=2}^{3} c_i \frac{\partial^2 u}{\partial x_i^2} + \sum_{i,j=1,i\neq j}^{3} c_{ij} \frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_{i=1}^{3} c_i \frac{\partial u}{\partial x_i} + c_0 u$$

on the open faces of $\partial \Omega$ for which $x_1 = L_{x_1}$ or $x_1 = U_{x_1}$, but $x_2 \in (L_{x_2}, U_{x_2})$ and $x_3 \in (L_{x_3}, U_{x_3})$. A similar linear boundary condition holds on the other open faces of $\partial \Omega$.

Similarly, using (3), the PDE (1) reduces to

$$\frac{\partial u}{\partial \tau} = c_{33} \frac{\partial^2 u}{\partial x_3^2} + \sum_{i,j=1,i\neq j}^{3} c_{ij} \frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_{i=1}^{3} c_i \frac{\partial u}{\partial x_i} + c_0 u$$

on the open edges of $\partial \Omega$ for which

1. either $x_1 = L_{x_1}$ or $x_1 = U_{x_1}$,
2. either $x_2 = L_{x_2}$ or $x_2 = U_{x_2}$, and
3. $x_3 \in (L_3, U_3)$. 


A similar linear boundary condition holds on the other open edges of \( \partial \Omega \).

Finally, using (3), the PDE (1) reduces to

\[
\frac{\partial u}{\partial \tau} = \sum_{i,j=1, i \neq j}^3 c_{ij} \frac{\partial^2 u}{\partial x_i \partial x_j} + \sum_{i=1}^3 c_i \frac{\partial u}{\partial x_i} + c_0 u
\]

at the corners of \( \partial \Omega \) at which
1. either \( x_1 = L_{x_1} \) or \( x_1 = U_{x_1} \),
2. either \( x_2 = L_{x_2} \) or \( x_2 = U_{x_2} \), and
3. either \( x_3 = L_{x_3} \) or \( x_3 = U_{x_3} \).

2.2 Space discretization: finite difference schemes

For the discretization of the space variables in the PDE (1), we employ second-order central differences in the interior of the rectangular domain \( \Omega \). Let the number of subintervals be \( n - 1 \), \( p - 1 \), \( q - 1 \) and \( l \) in the \( x_1 \), \( x_2 \), \( x_3 \), and \( \tau \)-directions, respectively. The uniform grid mesh widths in the respective direction are denoted by

\[
\Delta x_1 = \frac{U_{x_1} - L_{x_1}}{n - 1}, \quad \Delta x_2 = \frac{U_{x_2} - L_{x_2}}{p - 1},
\]

\[
\Delta x_3 = \frac{U_{x_3} - L_{x_3}}{q - 1}, \quad \Delta \tau = \frac{T}{l}.
\]

The gridpoint values of a FD approximation are denoted by

\[
u_{i,j,k}^m \approx u(x_{1i}, x_{2j}, x_{3k}, \tau_m) = u(L_{x_1} + i \Delta x_1, L_{x_2} + j \Delta x_2, L_{x_3} + k \Delta x_3, m \Delta \tau),
\]

where \( i = 0, \ldots, n - 1, j = 0, \ldots, p - 1, k = 0, \ldots, q - 1, m = 0, \ldots, l \).

Second-order FD approximations to the first and second partial derivatives of the space variables in (1) are obtained by central schemes, while the cross-derivatives are approximated by a four-point FD stencil. For example, at the reference point \( (x_{1i}, x_{2j}, x_{3k}, \tau_m) \), the first and second partial derivatives with respect to \( x_1 \) (i.e., \( \partial u/\partial x_1 \) and \( \partial^2 u/\partial x_1^2 \)) are approximated by

\[
\frac{\partial u}{\partial x_1} \approx \frac{u_{i+1,j,k}^m - u_{i-1,j,k}^m}{2 \Delta x_1}
\]

(7)
and

\[ \frac{\partial^2 u}{\partial x_1^2} \approx \frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{(\Delta x_1)^2}, \]

respectively, while the cross-derivative \( \frac{\partial^2 u}{\partial x_1 \partial x_2} \) is approximated by

\[ \frac{\partial^2 u}{\partial x_1 \partial x_2} \approx \frac{u_{i+1,j+1,k} + u_{i-1,j-1,k} - u_{i-1,j+1,k} - u_{i+1,j-1,k}}{4\Delta x_1 \Delta x_2}, \]

which can be viewed as obtained by successively applying the FD scheme (7) for the first derivatives in the \( x_1 \)- and \( x_2 \)-directions. Similar approximations can be obtained for the remaining spatial derivatives.

For the boundary points, in order to avoid introducing gridpoints outside \( \Omega \), we use one-sided FD approximations to derivatives, as discussed below. (For an alternative discretization approach on the boundary, see Remark 2.1.) We use the one-sided forward and backward FD approximations

\[ \frac{\partial u}{\partial x_1} \bigg|_{L_{x_1}} \approx \frac{u_{1,j,k} - u_{0,j,k}}{\Delta x_1} \quad \text{or} \quad \frac{\partial u}{\partial x_1} \bigg|_{U_{x_1}} \approx \frac{u_{n-1,j,k} - u_{n-2,j,k}}{\Delta x_1} \]

at points on the boundary \( \partial \Omega \) for which \( x_1 = L_{x_1} \) or \( x_1 = U_{x_1} \), respectively. The mixed derivatives \( \frac{\partial^2 u}{\partial x_1 \partial x_2} \), \( j = 2, 3 \), in (4) can be discretized by successively applying the one-sided FD for the first derivative in the \( x_1 \)-direction (scheme (10)) and the standard central FD scheme for the first derivative in the other direction (scheme (7)). For instance, applying this discretization technique to \( \frac{\partial^2 u}{\partial x_1 \partial x_2} \) at \( x_1 = L_{x_1} \) and \( x_1 = U_{x_1} \) gives rise to the FD formulas

\[ \frac{\partial^2 u}{\partial x_1 \partial x_2} \bigg|_{L_{x_1}} \approx \frac{u_{1,j+1,k} + u_{0,j-1,k} - u_{0,j+1,k} - u_{1,j-1,k}}{2\Delta x_1 \Delta x_2}, \]

\[ \frac{\partial^2 u}{\partial x_1 \partial x_2} \bigg|_{U_{x_1}} \approx \frac{u_{n-1,j+1,k} + u_{n-2,j-1,k} - u_{n-2,j+1,k} - u_{n-1,j-1,k}}{2\Delta x_1 \Delta x_2}, \]

respectively. All other derivatives in (4) can be evaluated using the central FD schemes (7), (8) and (9).

We introduce one more one-sided FD scheme for mixed derivatives such as \( \frac{\partial^2 u}{\partial x_1 \partial x_2} \) in (5) or (6) at boundary points for which
1. \( x_1 = L_{x_1} \) or \( x_1 = U_{x_1} \), and
2. \( x_2 = L_{x_2} \) or \( x_2 = U_{x_2} \).

To this end, we apply the one-sided FD scheme (10) in both the \( x_1 \)- and \( x_2 \)-directions. For example, at \( x_1 = L_{x_1} \) and \( x_2 = L_{x_2} \), we use

\[
\frac{\partial^2 u}{\partial x_1 \partial x_2} \bigg|_{L_{x_1}, L_{x_2}} \approx \frac{u_{1,1,k}^{m} + u_{0,0,k}^{m} - u_{1,0,k}^{m} - u_{0,1,k}^{m}}{\Delta x_1 \Delta x_2}.
\]

Other central FD schemes are replaced by one-sided FD schemes similar to those described above whenever using a central FD scheme in one of (4), (5) or (6) would require the use of a gridpoint outside \( \Omega \).

For brevity, we omit the detailed derivations of (7)–(12), but, through Taylor expansions, it can be verified that each of (7), (8) and (9) has a second-order truncation error, while (10), (11) and (12) are first-order, provided that the function \( u \) is sufficiently continuously differentiable. It is worth noting that, although (10), (11) and (12) are first-order approximations to the first and the mixed derivatives, when the computational domain is properly truncated, the far-field effects of the boundaries on the spot price of the derivatives is insignificant. Thus, we still observe second-order convergence for the spot price of the derivatives in practice.

The FD discretization of the spatial differential operator \( \mathcal{L} \) of (1) on the spatial grid \( \Omega \) is performed by replacing each spatial derivative appearing in the operator \( \mathcal{L} \) by its corresponding FD scheme (as in (7)–(12)).

**Remark 2.1.** A possible alternative approach to using the one-sided forward and backward FD approximations (10) is to introduce layers of "ghost" gridpoints outside the computational domain \( \Omega \), and use the standard central FD schemes described above to discretize the PDE (1) and the boundary conditions (3) at gridpoints on the boundaries. Then, the "ghost" gridpoints can be eliminated using the equations arising from (1) and (3), assuming that these equations are linearly independent.

### 2.3 Time discretization: ADI schemes

Several techniques, such as Crank-Nicolson or ADI, can be used for the time discretization of (1). Let \( u^m \) denote the vector of values at time \( \tau_m \) on the mesh \( \Omega \) that approximates the exact solution \( u^m = u(x_1, x_2, x_3, \tau_m) \). The Crank-Nicolson method defines an approximation \( u^m \) to the true solution \( u^m \), \( m = 1, 2, \ldots, l \), by

\[
\left( I - \frac{1}{2} \Delta \tau A^m \right) u^m = \left( I + \frac{1}{2} \Delta \tau A^{m-1} \right) u^{m-1},
\]
where $I$ denotes the $npq \times npq$ identity matrix and $A^m$ is the matrix of the same size as $I$ arising from the FD discretization of the differential operator $L$. For brevity, we omit the explicit formula for $A^m$. Applying direct methods, such as $LU$ factorization, to solve this linear system can be computationally very expensive for several reasons: (i) the matrix $I - \frac{1}{2} \Delta t A^m$ possesses a bandwidth proportional to $\min\{np, nq, pq\}$, depending on the ordering of the gridpoints, (ii) sparse solvers suffer considerable fill-in when solving systems derived from PDEs of the form (1) and (iii) this matrix may need to be factored at each timestep because of its possible dependence on the timestep index $m$ arising from time-dependent coefficients in the PDE (1). Note that the latter case happens quite frequently in financial applications, such as option pricing, where the time-dependent cross-correlations between stochastic processes in the model are present. To avoid the high computational cost of direct methods, iterative methods, such as GMRES or the Conjugate Gradient method, can be combined with a preconditioning technique, such as the Fast Fourier Transform or an Incomplete LU factorization, to solve (13). The reader is referred to [10] for a detailed discussion of this approach.

For the time discretization of the PDE (1), we consider in this paper a splitting technique of ADI type. We decompose the matrix $A^m$ into four submatrices:

$$A^m = A_0^m + A_1^m + A_2^m + A_3^m.$$  

The matrix $A_0^m$ is the part of $A$ that comes from the FD discretization of the mixed derivative terms in (1), while the matrices $A_1^m$, $A_2^m$ and $A_3^m$ are the three parts of $A^m$ that correspond to the spatial derivatives in the $x_1$-, $x_2$- and $x_3$-directions, respectively. The term $c_0 u$ in (1) is distributed evenly over $A_1^m$, $A_2^m$ and $A_3^m$. The FD discretization for the spatial variables described in (7), (8) and (10) implies that, if the gridpoints are ordered appropriately, the matrices $A_1^m$, $A_2^m$ and $A_3^m$ are block-diagonal, with tridiagonal blocks. More specifically, if the mesh points are ordered in the $x_i$-direction first, $i = 1, \ldots, 3$, then the matrix $A_i$ is block-diagonal with tridiagonal blocks.

We consider the splitting scheme proposed by the Hundsdorfer and Verwer [16, 17], referred to henceforth as the HV scheme. Starting from an approximation $u^{m-1}$ to the exact solution $u^{m-1}$, the HV scheme generates an approximation $u^m$ to the exact solution $u^m$, $m = 1, 2, \ldots, l$, ...
by\(^1\)

\begin{align}
\text{(14a)} & \quad v_0 = u^{m-1} + \Delta \tau A^{m-1} u^{m-1}, \\
\text{(14b)} & \quad (I - \theta \Delta \tau A^m_i)v_i = v_{i-1} - \theta \Delta \tau A_i^{m-1} u^{m-1}, \quad i = 1, 2, 3, \\
\text{(14c)} & \quad \tilde{v}_0 = v_0 + \frac{1}{2} \Delta \tau (A^m v_3 - A^{m-1} u^{m-1}), \\
\text{(14d)} & \quad (I - \theta \Delta \tau A^m_i)\tilde{v}_i = \tilde{v}_{i-1} - \theta \Delta \tau A_i^m v_3, \quad i = 1, 2, 3, \\
\text{(14e)} & \quad u^m = \tilde{v}_3.
\end{align}

The parameter \( \theta \) in (14) is directly related to the stability and accuracy of the ADI scheme. As mentioned in [16], a recommended range for \( \theta \) is \( \frac{1}{2} \leq \theta \leq 1 \), where \( \theta = 1/2 \) is the most accurate, but \( \theta = 1 \) gives more damping of the error terms arising from nonsmooth initial conditions (i.e., the payoff function). To take advantage of this, we apply the HV scheme with \( \theta = 1 \) for the first few (usually two) initial timesteps and then switch to \( \theta = 1/2 \) for the remaining timesteps. We refer to this time-stepping technique as \textit{HV smoothing}. We emphasize that choosing the parameter \( \theta = 1 \) gives a "partially" fully implicit time-stepping method only, not a fully implicit one. Hence, HV smoothing is not the same as Rannacher smoothing [30], which initially uses a few (usually two or three) steps of fully implicit time-stepping before switching to another time-stepping method, such as Crank-Nicolson.

The above splitting scheme treats the mixed derivative part (\( A_0^m \)) in a fully explicit way while the \( A_i^m \) parts, \( i = 1, 2, 3 \), are treated implicitly. The first two lines of (14) can be viewed as an explicit Euler predictor steps followed by three implicit, but unidirectional, corrector steps aiming to stabilize the predictor step. Several well-known ADI methods, such as the Douglas and Rachford method [11], are special instances of these two steps. The purpose of the additional stages that compute \( \tilde{v}_i, i = 0, \ldots , 3 \), is to restore second-order convergence for the general case with cross-derivatives, while retaining the unconditional stability of the scheme. Since the matrices \( A_i^m, i = 1, 2, 3 \), are tridiagonal, the number of floating-point operations per timestep is directly proportional to \( npq \), which yields a significant reduction in computational cost compared to the application of a direct method. Moreover, the block diagonal structure of these matrices gives rise to a natural, efficient parallelization for the solutions of the linear systems in Steps (14b) and (14d). However, it is less obvious how Steps (14a) and (14c) can be efficiently parallelized. We address this point in more detail in Section 4.

\(^1\)This is the scheme (1.4) in [18] with \( \mu = 1/2 \).
3 GPU device architecture  In this section, we summarize some key properties of the GPU device architecture and the CUDA Application Programming Interface (API) necessary to understand our implementation of the parallelization of the ADI schemes, which is discussed in the next section. Although the focus here is on NVIDIA products, offerings from other GPU manufacturers, such as ATI, are similar. This section is intended as a short overview of the basic concepts only. More detailed descriptions of the architecture and the programming model can be found, for example, in [26, 24].

![Architectural visualization of a GPU device and memory](image)

The modern GPU can be viewed as a set of independent streaming multiprocessors (SMs) [26]. One such SM contains, amongst other things, several scalar processors which can execute integer and single-precision floating-point arithmetic, a multi-threaded instruction unit (I/U), and shared memory. The latest GPUs can perform double-precision floating-point arithmetic. A graphical illustration of the layout of a GPU is given in Figure 1. The shared memory can be accessed by all scalar processors of a multiprocessor, while the registers have processor scope. The device (or global) memory can be accessed by all processors on the chip. Furthermore, constant cache, a small part of the device
memory dedicated to storing constants, is also available. Note that the constant cache is read-only and has faster access than the shared memory.

Typically, the CPU (the host) runs the program skeleton and offloads the more computationally demanding code sections to the GPUs (the device). Functions that run on the device are called kernel functions or simply kernels. When a C program using CUDA extensions and running on the CPU invokes a kernel, many copies of this kernel, which are referred to as threads, are distributed to the available multiprocessors, where they are executed. Since all threads of a parallel phase execute the same code, the programming model of CUDA is an instance of the widely used Single Instruction Multiple Data (SIMD) parallel programming style. Within the CUDA framework, operations are performed by threads that are grouped into threadblocks, which are in turn arranged on a grid. A grid of threadblocks could be one- or two-dimensional, with up to 65,535 blocks in each dimension. The total size of a threadblock is limited to 512 threads, with the flexibility of distributing these elements into one-, two-, or three-dimensional arrays, depending on the problem being solved. The CUDA framework assigns unique coordinates, referred to as threadIdx and blockIdx to each thread and each threadblock, respectively, which are accessible in standard C language via built-in variables. From the programmer’s point-of-view, the main functionality of blockIds and threadIds is to provide threads with a means to distinguish among themselves when executing the same kernels. This makes it possible for each thread to identify its share of work in a SIMD application.

Threads in a threadblock are executed by processors within a single SM. One or more threadblocks may be assigned to the same SM at a time, depending on the available resources and the size of each threadblock. All threads in a threadblock can read from and write on any shared memory location assigned to that threadblock. Consequently, threads within the same threadblock are able to communicate with each other very efficiently via the shared memory. Furthermore, all threads in a threadblock are able to synchronize their executions. On the other hand, threads belonging to different threadblocks are not able to communicate efficiently with each other, nor to synchronize their executions.

Within the CUDA framework, threadblocks can execute in any order relative to each other, which allows transparent scalability in the parallelism of CUDA kernels. Regarding the execution timing of threads within each threadblock, the correctness of executing a kernel should be independent of whether certain threads execute in synchrony with each other. However, due to various hardware considerations, the current
generation of CUDA devices actually does bundle multiple threads for execution in groups of 32 threads, referred to as warps. A group of 16 threads is a half-warp. Threads of one warp are handled on the same multiprocessor. If the threads of a given warp diverge by a data-induced conditional branch, each branch of the warp will be executed serially and the processing time of the given warp will be the sum of the processing times for all the branches. This must be avoided whenever possible. We return to this point in the next section, where we discuss memory coalescing.

The Tesla series is the first family of GPUs that is dedicated to general purpose computing. The NVIDIA Tesla 10-series GPUs (Tesla S1060/S1070 (server version)) consists of 30 independent SMs, each containing 8 processors running at 1.44 GHz, a total of 16384 registers, and 16 KB of shared memory.

4 GPU implementation of ADI schemes

We now discuss a GPU-based parallel algorithm for the solution of the model PDE problem. We emphasize that we do not aim to parallelize across time, but rather we focus on the parallelism within one timestep, via a parallelization of the HV scheme ((14a)–(14e)).

As mentioned earlier, the HV scheme can be divided into two phases. The first phase consists of a forward Euler step (predictor step (14a)), followed by three implicit, but unidirectional, corrector steps (14b), the purpose of which is to stabilize the predictor step. The second phase (i.e., (14c) and (14d)) restores second-order convergence of the discretization method if (1) contains mixed derivatives. Step (14e) is trivial. With respect to the CUDA implementation, the two phases are essentially the same; they can both be decomposed into matrix-vector multiplications and solving independent tridiagonal systems. Hence, for brevity, we focus on describing a GPU parallelization of the first phase, only briefly mentioning the second phase.

For presentation purposes, let

\[ w_i = \Delta \tau A_i^{m-1} u^{m-1}, \quad i = 0, 1, 2, 3, \]

\[ \tilde{A}_i^m = I - \theta \Delta \tau A_i^m, \quad \tilde{v}_i = v_{i-1} - \theta w_i, \quad i = 1, 2, 3, \]

and note that

\[ v_0 = u^{m-1} + \sum_{i=0}^{3} w_i. \]
It is worth noting that the vectors $w_i, v_i, i = 0, 1, 2, 3$, and $\tilde{v}_i, i = 1, 2, 3$, depend on $\tau$, but, to simplify the notation, we do not indicate the superscript for the timestep index.

The first phase of the HV scheme consists of the following steps:

(i) Step a.1: Compute the matrices $A_i^m, i = 0, 1, 2, 3$, and $\tilde{A}_i^m, i = 1, 2, 3$, and the vectors $w_i, i = 0, 1, 2, 3$, and $v_0$.

(ii) Step a.2: Set $\tilde{v}_1 = v_0 - \theta w_1$ and solve $A_1^m v_1 = \tilde{v}_1$;

(iii) Step a.3: Set $\tilde{v}_2 = v_1 - \theta w_2$ and solve $A_2^m v_2 = \tilde{v}_2$;

(iv) Step a.4: Set $\tilde{v}_3 = v_2 - \theta w_3$ and solve $A_3^m v_3 = \tilde{v}_3$.

Among these steps, Steps a.2, a.3, and a.4 are inherently parallelizable, due to the block diagonal structure of the matrices $A_i^m, i = 1, 2, 3$, while it is not as obvious how Step a.1, in particular the computation of the vectors $w_i, i = 0, 1, 2, 3$, can be efficiently parallelized. This is the most sensitive part of the algorithm. Below, we discuss in more detail how to implement these steps on a GPU with particular focus on Step a.1, since Steps a.2, a.3 and a.4 are relatively straightforward.

4.1 First phase—Step a.1

In this subsection, we first describe the partitioning of the computational grid into a grid of blocks, then we discuss the assignment of gridpoints to threads of threadblocks. We next illustrate how the matrices $A_i^m, i = 0, 1, 2, 3$, and $\tilde{A}_i^m, i = 1, 2, 3$, are assembled. Then we present an efficient GPU-based computation algorithm for the vectors $w_i, i = 0, 1, 2, 3$, and $v_0$.

4.1.1 Partitioning of the computational grid and assignment of gridpoints to threads

Recall that we have a discretization grid of $n \times p \times q$ points. We can view a set of $q$ consecutive gridpoints in the $x_3$-direction as a “stack” of $q$ gridpoints. Clearly, there are $np$ such stacks in the discretization grid. The general idea for distributing the data and computation of Step a.1 is to assign the work associated with each stack of $q$ gridpoints (and the respective rows of matrices $A_i^{m-1}, i = 0, 1, 2, 3$, $\tilde{A}_i^{m-1}, i = 1, 2, 3$, and components of vectors $u_i^{m-1}, w_i, i = 0, 1, 2, 3$, and $v_0$) to a different thread. Thus, there is an one-to-one correspondence between the set of stacks and the set of threads. Moreover, just as CUDA threads are grouped into threadblocks, we also group gridpoints into blocks. We partition the computational grid of size $n \times p \times q$ into three-dimensional (3-D) blocks of size $n_b \times p_b \times q$, each of which can be viewed as consisting of $q$ two-dimensional (2-D) blocks, referred to as tiles, of size $n_b \times p_b$. For Step a.1, we let the kernel generate a $\lceil \frac{n}{n_b} \rceil \times \lceil \frac{p}{p_b} \rceil$ grid of threadblocks, where $\lceil \cdot \rceil$ denotes the ceil-
A PARALLEL IMPLEMENTATION ON GPUS

\[ p = 8 \]
\[ n = 8 \]
\[ n_b = 4 \]
\[ p_b = 2 \]
\[ q = 10 \]

**FIGURE 2:** An illustration of the partitioning approach considered for the first phase, Step a.1.

The partitioning function. Each of the threadblocks, in turn, consists of a total of \( n_b p_b \) threads arranged in 2-D arrays, each of size \( n_b \times p_b \). All gridpoints of a \( n_b \times p_b \times q \) 3-D block are assigned to one threadblock only, with one thread for each stack of \( q \) gridpoints, i.e., there is an one-to-one correspondence between the set of 3-D blocks of gridpoints and the set of threadblocks (see also Figure 2). Note that, since each 3-D block has a total of \( q n_b \times p_b \) tiles and each threadblock is of size \( n_b \times p_b \), the approach that we use here suggests a \( q \)-iteration loop in the kernel. During each iteration of this loop, each thread of a threadblock carries out all the computations/work associated with one gridpoint, and each threadblock processes one \( n_b \times p_b \) tile.

Figure 2 illustrates an application of the aforementioned partitioning approach on an example computational grid of size \( n \times p \times q \equiv 8 \times 8 \times 10 \), with \( n_b = 4 \) and \( p_b = 2 \). In this example, the computational domain is partitioned into 3-D blocks of size \( n_b \times p_b \times q \equiv 4 \times 2 \times 10 \), each of which can be viewed as consisting of ten \( 4 \times 2 \) tiles, or as 8 (= \( 4 \times 2 \)) stacks of \( 10 \) gridpoints. The kernel generates a grid of threadblocks of size \( \text{ceil}(\frac{n}{n_b}) \times \text{ceil}(\frac{p}{p_b}) \equiv 2 \times \frac{8}{2} \equiv 2 \times 4 \), with each 2-D threadblock having size \( 4 \times 2 \). Each 3-D block of gridpoints is assigned to a threadblock which carries out all the computations/work associated with all gridpoints of
the ten $4 \times 2$ tiles, in a 10-iteration loop in the kernel, proceeding tile-by-tile.

4.1.2 Construction of the matrices $A^m_i$, $i = 0, 1, 2, 3$, and $\hat{A}^m_i$, $i = 1, 2, 3$

Note that each of the matrices $A^m_i$, $i = 0, 1, 2, 3$, and $\hat{A}^m_i$, $i = 1, 2, 3$, has a total of $npq$ rows, with each row corresponding to a gridpoint of the computational domain. Our approach is to assign each of the threads to assemble $q$ rows of each of the matrices (a total of three entries per row of each matrix, since all matrices are tridiagonal). More specifically, during each iteration of the $q$-iteration loop in the kernel, each group of $nbp$ rows corresponding to a tile is assembled in parallel by a $nb \times pb$ threadblock, with one thread for each row. That is, a total of $np$ consecutive rows are constructed in parallel by the threadblocks during each iteration. In this way, the first $np$ rows are processed in parallel during the first iteration, then the second $np$ rows (from the $(np + 1)$st row to the $2np$-th row) are processed in a similar way during the second iteration, and so on.

4.1.3 Computation of the vectors $w_i$, $i = 0, 1, 2, 3$, and $v_0$

Recall that the vectors $w_i$, $i = 0, 1, 2, 3$, are given by

$$w_i = \Delta \tau A^{m-1}_i u^{m-1},$$

and the vector $v_0$ by $v_0 = u^{m-1} + \sum_{i=0}^{3} w_i$. Note that the data of the previous timestep (old data), i.e., the vector $u^{m-1}$, and model constant parameters, if any, are available in the global memory and constant cache, respectively. We emphasize that the data copying from the host memory to the device memory occurs on the first timestep only, for the initial condition data, $u^0$, and the model constants. Data for the subsequent timesteps and the ADI timestepping scheme (14) are stored on the device memory. The initial host-to-device copying can be achieved via CUDA functions `cudaMemcpy2D()` for data and `cudaMemcpyToSymbol()` for constants.

Before describing how the computation of the vectors $w_i$, $i = 0, 1, 2, 3$, is carried out, we need to draw the reader’s attention to certain facts.

(a) The FD discretization scheme used in this paper, as described by (7)–(12), gives rise to a 19-point stencil. This implies that, from the point-of-view of a thread, the product of a row of $A^{m-1}_i$ by $u^{m-1}$ involves 19 components of $u^{m-1}$, three of which are assigned to the thread, and 16 others are assigned to neighbouring threads.
A PARALLEL IMPLEMENTATION ON GPUS

(b) A threadblock carrying the computation of a stack of \( q \) tiles (the stack being in the \( x_3 \) direction) needs the values of neighbouring gridpoints from adjacent tiles in the \( x_1 \) and \( x_2 \) directions, referred to as halo values.

(c) Threads within the same threadblock can communicate with each other effectively via the shared memory, while threads in different threadblocks cannot.

(d) Accessing the global memory is costly, and, therefore, the number of accesses of the global memory should be kept to a minimum. Also, global memory access coalescing is desirable whenever possible, as it minimizes the cost of global memory access. Memory coalescing is discussed in more detail below.

(e) To calculate the values corresponding to gridpoints of the \( k \)th tile (i.e., the tile on the \( k \)th \( x_1 \)-\( x_2 \) plane), the data of the two adjacent tiles in the \( x_3 \) direction (i.e., the \((k - 1)\)st and the \((k + 1)\)st tiles) are needed as well. Since 16KB of shared memory available per multiprocessor are not sufficient to store many data tiles, each threadblock works with three data tiles of size \( n_b \times p_b \) at a time and proceeds in the \( x_3 \)-direction [27].

Taking into account the above considerations, we adopt an effective data loading strategy, so that each thread of a threadblock has the necessary data to compute the associated components of \( w \), the number of global memory accesses is small, and partial memory coalescing is achieved. More specifically, during the \( k \)th iteration of the \( q \)-iteration loop in the kernel, assuming the data corresponding to the \( k \)th and \((k - 1)\)st tiles and associated halos are in the shared memory from the previous iteration, each threadblock

1. loads from the global memory into its shared memory the old data (vector \( u^{m-1} \)) corresponding to the \((k + 1)\)st tile, and the associated halos (in the \( x_1 \)- and \( x_2 \)-directions), if any;
2. computes and stores new values (vectors \( w_i, i = 0, 1, 2, 3 \) and \( v_0 \)) for the \( k \)th tile using data of the \((k - 1)\)st, \( k \)th and \((k + 1)\)st tiles, and of the associated halos, if any;
3. copies the newly computed data of the \( k \)th tile (vectors \( w_i, i = 1, 2, 3 \) and \( v_0 \)) from the shared memory to the global memory, and frees the shared memory locations taken by the data of the \((k - 1)\)st tile, and associated halos, if any, so that they can be used in the next iteration.

From the point-of-view of a thread, during each iteration of the \( q \)-iteration loop, each thread of a threadblock loads the data associated with one gridpoint, plus either one halo, if the gridpoint is at an edge of
a tile, or two halos, if the gridpoint is at a corner of a tile.

In Figure 3, an example illustrating the aforementioned data loading approach with tiles having size $n_t \times p_t \equiv 8 \times 8$ (marked in ■ with halos (marked in □) is presented. Note that, from the viewpoint of the central tile, the halos are not part of the tile, but come from adjacent tiles: the North, South, East, West halos are from neighboring tiles in those respective directions. As can be seen from Figure 3, during each iteration of the $q$-iteration loop in the kernel, only threads whose gridpoints are close to a boundary of a tile need to load into the shared memory the halo values along that boundary in addition to their own gridpoints’ old data. Threads corresponding to interior gridpoints of a tile need to read in only their gridpoints’ old data. Thus, old data corresponding to interior gridpoints of a tile (i.e., the majority of the gridpoints) are loaded once only. The old data corresponding to a point along the North, South, East, or West boundaries of a tile are loaded twice, once by the thread associated with the point, and once more by the thread of the neighbouring threadblock as a halo value. The data at the tiles’ corners are loaded four times. This loading strategy keeps the repetition of data loading, and, therefore, the number of global memory accesses to a minimum. Furthermore, this loading approach is partially coalesced, as is discussed in more detail towards the end of this section.

FIGURE 3: An example of $n_t \times p_t = 8 \times 8$ tiles with halos.
Once in the shared memory of a threadblock, the data corresponding to a gridpoint is accessed (at most) 19 times: three times by the thread to which the gridpoint is assigned, and 16 times by the 8 neighbouring threads in the same threadblock. We emphasize that these repeated shared memory accesses are cheap.

**Remark 4.1.** It is important to note that, in our implementation, barrier synchronization among threads in the same threadblock, which can be achieved by using the function `syncthreads()`, is used in both the loading and computing phases of each iteration of the $q$-iteration loop in the kernel. The purpose of barrier synchronization is to ensure that all threads in the same threadblock have completed a phase (e.g., loading the data) before any of them move to the next phase (e.g., accessing the data). This is essential because one thread may need the data loaded or computed by a neighboring thread in the same threadblock.

4.1.4 Memory coalescing To optimize performance, we must ensure coalesced data loads from the global memory. It is important to recall that threads in a warp should execute the same instruction at any given time, in order to avoid potential serial execution of threads. Furthermore, in CUDA, the shared memory consists of 16 memory banks. To achieve maximum memory performance, memory accesses at any one time are handled by half-warps (16 threads), each thread accessing a different bank. When the threads of a half-warp execute global loads, the loads are consolidated if they meet certain constraints necessary for the hardware to perform coalesced data loads. These constraints include (i) the threads in the half-warp must access consecutive global memory locations; and (ii) the number of threads matters only along the first dimension of the threadblock. We refer interested readers to [26] for a more complete discussion of all the requirements. Regarding the GPU computation step for vectors $w_i$, $i = 0, 1, 2, 3$, to ensure the data transfer coalescing, it is necessary to have the tile size in the $x_1$-direction, i.e., $n_0$, be a multiple of 16, since each half-warp is of size 16 and that gridpoints at this step are ordered in the $x_1$-direction first. Assuming that $n_0$ is a multiple of 16, and taking into account the data loading strategy for $u^{m-1}$ that we adopted, the interior of the data tiles can be read from global memory into the shared memory in a coalesced way. The halos along the $x_1$-direction (i.e., North and South halos in Figure 3) can also be loaded in a coalesced fashion. However, halos along the $x_2$-direction (East and West halos) cannot be accessed via a coalesced pattern, since they do not belong to consecutive memory locations. As a result, the
data loading approach for Step a.1 is not fully coalesced, although it is highly effective. We believe it is impossible to attain full memory coalescing for the data-loading part of this phase.

4.2 First phase—Steps a.2, a.3, a.4 Recall that, in Steps a.2, a.3 and a.4, we need to solve \( \hat{A}_m^i v_i = \tilde{v}_i, \quad i = 1, 2, 3 \), where

\[
\tilde{v}_i = v_{i-1} - \theta w_i, \quad i = 1, 2, 3.
\]

After Step a.1, the matrices \( \hat{A}_m^i, i = 1, 2, 3 \), and the vectors \( w_i, i = 1, 2, 3 \) and \( v_0 \) are stored in the device memory. Similarly, the data after Steps a.2, a.3, a.4 (i.e., \( v_i, i = 1, 2, 3 \)) are held in the device memory, and as a result, the elements of the right-side vectors \( \tilde{v}_i, i = 1, 2, 3 \), can be easily computed in parallel. The parallel solution of the tridiagonal systems \( \hat{A}_m^i v_i = \tilde{v}_i, i = 1, 2, 3 \), can be achieved via a data partitioning different from the one described for Step a.1. The data partitioning for Steps a.2, a.3 and a.4 is motivated by the block structure of the tridiagonal matrices \( \hat{A}_m^i \). For example, \( \hat{A}_1^m \) has \( pq \) diagonal blocks, and each block is tridiagonal of size \( n \times n \), while \( \hat{A}_2^m \) has \( nq \) diagonal blocks, and each block is tridiagonal of size \( p \times p \). Our approach for the solution of \( \hat{A}_m^i v_i = \tilde{v}_i, i = 1, 2, 3 \), is based on the parallelism arising from independent tridiagonal solutions, rather than the parallelism within each one. To this end, each independent tridiagonal system is assigned to a different thread. Moreover, when we solve in one direction, the data are partitioned with respect to the other two. For example, the solution of \( \hat{A}_1^m v_1 = \tilde{v}_1 \) (Step a.2) is computed by first partitioning \( \hat{A}_1^m \) and \( \tilde{v}_1 \) into \( pq \) independent \( n \times n \) tridiagonal systems, and then assigning each tridiagonal system to one of \( pq \) threads. Similarly, the solution of \( \hat{A}_2^m v_2 = \tilde{v}_2 \) (Step a.3) is done via re-partitioning \( \hat{A}_2^m \) and \( \tilde{v}_2 \) into \( nq \) independent \( p \times p \) tridiagonal systems, and then assigning each tridiagonal system to one of \( nq \) threads. Between Steps a.2 and a.3, the data of vector \( v_1 \) are written back to the global memory, since, in Step a.3, a different partitioning of \( v_1 \) is needed to compute the right-side vector \( \tilde{v}_2 \) before \( \hat{A}_2^m v_2 = \tilde{v}_2 \) is solved.

To have one thread handle the solution of each tridiagonal system, we need to use sufficiently many threadblocks. Because the number of threads in a threadblock is limited to 512, and because we do not wish to have limitations on the grid sizes, we use many threadblocks for the solution of the independent tridiagonal systems. More specifically, at Steps a.2, a.3 and a.4, we have \( pq, nq \) and \( np \) total threads, respectively. In our implementation, each of the 2-D threadblocks used in Steps a.2,
a.3 and a.4 has the identical size \(r_t \times c_t\), where the values of \(r_t\) and \(c_t\) are determined by numerical experiments to maximize the performance. The size of the grid of threadblocks is determined accordingly. For example, for the parallel solution of \(\mathbf{A}^{i_0}_m\mathbf{v}_i = \mathbf{v}_i\), a 2-D grid of threadblocks of size \(\text{ceil}(\frac{m}{r_t}) \times \text{ceil}(\frac{n}{c_t})\) is invoked. An example of this approach is illustrated in Figure 4, where the 2-D grid of threadblocks is of size \(\text{ceil}(\frac{m}{r_t}) \times \text{ceil}(\frac{n}{c_t})\) \(\equiv \tilde{m} \times \tilde{n}\).

![Thread assignment for the parallel solution of independent tridiagonal systems. Each thread handles one tridiagonal system.](image)

Regarding the memory coalescing for Steps a.2, a.3 and a.4, note that, in the current implementation, the data between Steps a.1, a.2, a.3 and a.4 are ordered in the \(x_1\)-, then \(x_2\)-, then \(x_3\)-directions. As a result, the data partitionings for the tridiagonal solves in the \(x_2\)- and \(x_3\)-directions, i.e., for solving \(\mathbf{A}^{i_0}_m\mathbf{v}_i = \mathbf{v}_i\), \(i = 2, 3\), allow full memory coalescence, while the data partitioning for solving \(\mathbf{A}^{i_0}_m\mathbf{v}_1 = \mathbf{v}_1\) does not.

**Remark 4.2.** It may be possible to achieve memory coalescence for the tridiagonal solves in all three directions, if we renumber the grid-points between Steps a.1, a.2, a.3 and a.4 appropriately. However, such a renumbering will involve some overhead. In a future paper, we plan to investigate the trade-off between this overhead and achieving memory coalescence in two of the three directions only. The numerical experiments indicate that the current implementation is effective.
Remark 4.3. Another approach that can be employed for the solution of the tridiagonal systems arising in Steps a.2, a.3, and a.4 is to use parallel cyclic reduction methods \([15]\). In certain implementations \([33]\), these techniques have been shown to be more efficient and scalable than the technique adopted in this paper. However, a GPU-based implementation of these techniques is much more involved than that of the approach presented above.

Remark 4.4. Another possible approach that could be used for Steps a.2, a.3, and a.4 is based on partitioning the computational domain into 3-D subcubes of gridpoints, assigning one subcube to each thread, then employing a Schur complement domain decomposition method. To solve \(A^m_iv_i = \tilde{v}_i, \ i = 1, 2, 3\), each thread first solves a block of equations corresponding to its subcube of points in the \(x_i\) direction, \(i = 1, 2, 3\), i.e., a subdomain problem. Each thread then corrects the solution of the subdomain problem by taking into account the equations on the interfaces between its subcube and neighbouring subcubes, i.e., equations corresponding to points on the boundaries of the subcube. (All the subdomain problems are solved simultaneously in parallel, with each thread solving one problem.) In this approach, the communication cost (associated with copying from/to the global memory) is much less than the computational cost, assuming that the dimensions of the subcubes are chosen so that there are significantly more interior points within a subcube than interface points, whereas the communication cost of our approach is of the same order as the computational cost. Consequently, the alternative approach is expected to be more efficient and scalable than our approach. However, the implementation of the alternative approach on GPUs is far more complicated. In particular, how to handle memory coalescing is not obvious. Furthermore, to avoid any re-arrangement of data between Steps a.1 and a.2, the parallel implementation of Step a.1 needs to be adjusted to take into account the assignment of data according to subcubes. We plan to investigate the alternative approach in a future work.

### 4.3 Summary of the first phase

In this section, we summarize the data loading and computation of the first phase of the ADI scheme (14), when stepping from time \(\tau_{m-1}\) to \(\tau_m\).

We assume that, initially, the vector \(u^{m-1}\) is in the global memory and any needed constants (model parameters) are in the constant cache.

Step a.1:
A grid of $\text{ceil}(n/n_b) \times \text{ceil}(p/p_b)$ threadblocks is invoked, each of which consists of an $n_b \times p_b$ array of threads. Each threadblock does a $q$-iteration loop, processing an $n_b \times p_b$ tile at each iteration, and thus each thread does a $q$-iteration loop, processing one gridpoint at each iteration.

During one iteration of the $q$-iteration loop, each threadblock

1. loads from the global memory to its shared memory the components of $u^{n-1}$ corresponding to a tile, and the associated halo values;
2. computes the rows of $A_{1}^{m_i}, A_{1}^{m_i}, i = 1, 2, 3$ and components of $w_i, i = 0, 1, 2, 3$, and $v_0$ corresponding to a tile;
3. copies the newly computed rows of $A_{1}^{m_i}, A_{1}^{m_i}, i = 1, 2, 3$ and components of $w_i, i = 1, 2, 3$, and $v_0$ from its shared memory to the global memory.

Step a.2:

A grid of $\text{ceil}(p/r_t) \times \text{ceil}(q/c_t)$ threadblocks is invoked, each of which consists of an $r_t \times c_t$ array of threads. Each threadblock is assigned a subgrid of $n \times r_t \times c_t$ points, and thus each thread is assigned $n$ points along the $x_1$-direction. Each threadblock

1. loads from the global memory to its shared memory its rows of $\hat{A}_{1}^{m_i}$ and its components of $\hat{v}_1$ and $w_1$;
2. computes its components of $\hat{v}_1$;
3. solves $r_tc_t$ tridiagonal $n \times n$ systems (its part of $\hat{A}_{1}^{m_i}v_1 = \hat{v}_1$), with each thread solving one system;
4. copies its newly computed components of $v_1$ from its shared memory to the global memory.

Step a.3:

A grid of $\text{ceil}(n/r_t) \times \text{ceil}(q/c_t)$ threadblocks is invoked, each of which consists of an $r_t \times c_t$ array of threads. Each threadblock is assigned a subgrid of $r_t \times p \times c_t$ points, and thus each thread is assigned $p$ points along the $x_2$-direction. Each threadblock

1. loads from the global memory to its shared memory its rows of $\hat{A}_{2}^{m_i}$ and its components of $\hat{v}_1$ and $w_2$;
2. computes its components of $\hat{v}_2$;
3. solves $r_tc_t$ tridiagonal $p \times p$ systems (its part of $\hat{A}_{2}^{m_i}v_2 = \hat{v}_2$), with each thread solving one system;
4. copies its newly computed components of $v_2$ from its shared memory to the global memory.

Step a.4: A grid of $\text{ceil}(n/r_t) \times \text{ceil}(p/c_t)$ threadblocks is invoked, each of which consists of an $r_t \times c_t$ array of threads. Each threadblock is assigned a subgrid of $r_t \times c_t \times q$ points, and thus each thread is assigned $q$ points along the $x_3$-direction. Each threadblock

1. loads from the global memory to its shared memory its rows of $A_3^n$ and its components of $v_2$ and $w_3$;
2. computes its components of $\tilde{v}_3$;
3. solves $r_t c_t$ tridiagonal $q \times q$ systems (its part of $A_3^n v_3 = \tilde{v}_3$), with each thread solving one system;
4. copies its newly computed components of $v_3$ from its shared memory to the global memory.

At the end of the first phase, the vector $v_3$ is available in the global memory, and will be used in the second phase.

4.4 Second phase Several components required in the second phase of the HV scheme are available as results of the computations in the first phase, and hence are not recomputed. For example, the term $A_1^n u_1^{m-1}$, required in (14c), is needed to compute $v_0$ in Step a.1 of the first phase. The computation of $A_3^n v_3$ in (14c) can be achieved in the same way as the computation of $A_1^n u_1^{m-1}$ in Step a.1. The solutions of the tridiagonal systems in (14d) can be implemented in essentially the same way as in Steps a.2, a.3, a.4, described above. Note that all the tridiagonal matrices $I - \theta \Delta r A_i^n$ are already computed in Step a.1 of the first phase.

5 Application problems in finance In this section we discuss the pricing of multi-asset European options in a log-normal model. In particular, we consider European rainbow and basket options, each written on three underlying assets.

We denote by $s_i(t)$, $i = 1, 2, 3$, the value at time $t$ of the $i$th underlying asset. We consider the 3-D Black-Scholes PDE which governs the value $u \equiv u(s_1, s_2, s_3, t)$ of an option on the three assets [22]

\[
(15) \quad \frac{\partial u}{\partial t} + \mathcal{L} u \equiv \frac{\partial u}{\partial t} + \frac{1}{2} \sum_{i,j=1}^{3} \rho_{ij} \sigma_i \sigma_j s_i s_j \frac{\partial^2 u}{\partial s_i \partial s_j} + r \sum_{i=1}^{3} s_i \frac{\partial u}{\partial s_i} - ru = 0,
\]
with \(0 \leq s_i < \infty\), \(i = 1, \ldots, 3\), and \(t \in [0, T]\), where \(T\) is the maturity time of the option. Here, \(r > 0\) is the constant riskless interest rate; \(\sigma_i \geq 0\) is the constant volatility of the \(i\)th underlying asset; \(\rho_{ij}\) is the correlation factor between the \(i\)th and \(j\)th assets satisfying \(|\rho_{ij}| \leq 1\) for \(i, j = 1, 2, 3\), and \(\rho_{ii} = 1\) for \(i = 1, 2, 3\).

To solve the PDE (15) numerically by finite difference methods, we must truncate the domain to \(0 \leq s_i \leq U_{s_i}, i = 1, 2, 3\), for appropriately chosen \(U_{s_i}\) [35]. As mentioned earlier, since we solve the PDE (15) backward in time from \(T\) to the current time, \(t\), the change of variable \(\tau = T - t\) is used. Under this change of variable, the PDE (15) becomes \(\frac{\partial u}{\partial \tau} = Lu\). This brings the PDE (15) to the form (1) discussed in Section 2. Then the PDE is solved forward in \(\tau\).

A multi-asset European call option contract gives the holder the right, but not an obligation, to buy a specified basket of more than one underlying asset for a predetermined exercise price \(E\), also called the strike price. Such options belong to the so-called class of exotic options. In particular, we consider below in more detail European rainbow options and European basket options written on three underlying assets.

### 5.1 Rainbow options
Rainbow options can take various forms, but their common characteristic is that their payoff function depends on the underlying assets sorted by their values at maturity. As an example, we consider a European call-on-minimum rainbow option, the payoff of which is

\[
(16) \quad u(s_1, s_2, s_3, T) = \max(\min(s_1, s_2, s_3) - E, 0).
\]

Closed-form solutions of rainbow put/call options on the maximum or minimum on several assets are provided in [20]. Thus, we can use rainbow options as a benchmark solution to test the accuracy of our numerical methods.

### 5.2 Basket options
The payoff function of a European basket call option is typically based on the weighted sum of \(d\) assets \(s_i, i = 1, \ldots, d\), in the basket. For \(d = 3\), the payoff is

\[
(17) \quad u(s_1, s_2, s_3, T) = \max\left(\sum_{i=1}^{3} w_i s_i - E, 0\right),
\]

where \(w_i\) are the weights of the assets in the basket.
5.3 Numerical valuations The prices of the above options can be obtained by solving the PDE (15) backward in time from the maturity $T$ of the option to the current time $t \in [0, T)$. The payoff function provides the terminal condition at time $T$. The discretization schemes are similar to those described for the model problem (1) in Section 2. Regarding the parallel implementation, the host (CPU) first sets up the payoff vectors (terminal conditions) then offloads the computations of the numerical solution for the PDE to a GPU. The ADI technique (14) is employed for each timestep and is executed in a highly parallel fashion as described in the previous sections.

As frequently noted in the literature, nonsmoothness in the payoff function of financial contracts often gives rise to erratic observed orders of convergence \cite{19, 29, 34} and causes inaccuracies for numerical schemes. Hence, smoothing techniques, such as Rannacher timestepping \cite{30} (essentially a few steps of fully implicit timestepping followed by another timestepping method, such as Crank-Nicolson), must be employed. From our numerical experiments with rainbow and basket options, it appears that HV smoothing, discussed in Subsection 2.3, exhibits good damping of the nonsmooth payoff functions, and the expected second-order convergence of the numerical scheme is observed. Hence, Rannacher timestepping is not necessary. Similar favorable behavior of the HV scheme is also noted in \cite{19} for European options with stochastic volatility, where the value $\theta = \frac{1}{2} + \frac{\sqrt{3}}{6}$ is used for all timesteps.

6 Numerical results In this section, we present selected numerical results to demonstrate the effectiveness of the GPU-based parallel methods applied to the aforementioned financial problems. We used the CUDA 3.2 driver and toolkit, and all the experiments with the GPU code were conducted on a NVIDIA Tesla T10, which allows double-precision computations, and is connected to a two quad-core Intel “Harpertown” host system with Intel Xeon E5430 CPUs running at 2.66 GHz, 8 GB of FBM PC 5300 RAM. However, only one CPU core was employed for the experiments with the (nonmulti-threaded) CPU code. Note that, although CUDA supports the debug mode which allows us to execute the GPU code on the CPU, i.e., a nonparallel version of the GPU code, we do not opt for this choice, since this prevents optimized execution on the CPU. Rather, for a fair comparison, we wrote a separate CPU code. The size of each tile used in Step a.1 is chosen to be $n_b \times p_b \equiv 32 \times 4$, and the size of each threadblock used in the parallel solution of the independent tridiagonal systems in Steps a.2, a.3, a.4 is $r_t \times c_t \equiv 32 \times 4$, which appears
to be optimal on a Tesla T10. The CPU and GPU computation times, respectively denoted by “CPU time” and “GPU time”, measure the total computational times required for solving the model-dependent PDE over the length of the maturity of the securities. The GPU times include the overhead for memory transfers from the CPU to the device memory. The CPU and GPU times were obtained using the CUDA functions cutStartTimer( ) and cutStopTimer( ). The quantity “speedup” is defined as the ratio of the CPU time over the corresponding GPU time. All computations are carried out in double-precision. In this section, the quantity “log$_n$ ratio” provides an estimate of the convergence rate of the algorithm by measuring the true errors, since, for the applications we are considering, analytical or very accurate numerical solutions are known. More specifically, this quantity is defined by

$$\log_n \text{ratio} = \log_n \left( \frac{u_{\text{exact}} - u_{\text{approx}}(\Delta x)}{u_{\text{exact}} - u_{\text{approx}}(\frac{\Delta x}{n})} \right),$$

where $u_{\text{exact}}$ is the accurate reference solution, and $u_{\text{approx}}(\Delta x)$ is the approximate solution computed with discretization stepsize $\Delta x$. For second-order methods, such as those considered in this paper, the quantity log$_n$-ratio is expected to be about 2.

We use the set of parameters for three assets taken from [23]: $E = 100$, $r = 4\%$, $T = 1$, $\sigma_1 = 0.3$, $\sigma_2 = 0.35$, $\sigma_3 = 0.4$, $\rho_{12} = \rho_{13} = \rho_{23} = 0.5$. The spot prices are chosen to be $s_1(0) = s_2(0) = s_3(0) = E$. For basket options, we consider the weights of the assets to be $w_1 = w_2 = w_3 = \frac{1}{3}$, so that we have $\sum_{i=1}^{3} w_i s_i = \frac{1}{3} \sum_{i=1}^{3} s_i = E$. The analytic solution for the European call-on-minimum rainbow option with this set of parameters is computed to be 4.4450, using formulas in [20]. The reference solution for the basket option is 13.2449 obtained using an accurate numerical FFT-based pricing technique [23]. We refer to these two values as reference solutions for the rainbow and basket options, respectively. We truncate the unbounded domain into a finite-sized computational one $\{(s_1, s_2, s_3, \tau) \in [0, S_1] \times [0, S_2] \times [0, S_3] \times [0, T] \} \equiv \Omega \times [0, T]$, where $S_1 = S_2 = S_3 = 3E = 300$, which is also used in [23]. Note that, with this choice of the truncated computational domain and for all grid sizes considered for the multi-asset options, each of the spot prices is one of the gridpoints in the respective asset price grid.

Table 1 presents selected numerical results for the multi-asset options. In Table 1, the quantity “value” denotes the spot value of the option, the quantity “error” is computed as the absolute difference between the numerical solutions and the reference value (4.4450 and 13.2449 for the
rainbow option and the basket option, respectively). The computed prices for the rainbow and basket options on the CPU and GPU are identical and exhibit second-order convergence for the HV scheme, as expected. Regarding the timing results, the GPU is significantly faster than the CPU for any size of the discretized problem and has a speedup ratio of about 18 for the largest grid we considered. In Figure 5, we plot errors versus computation times required by each method. It is evident that GPU-based parallel methods are significantly more efficient than standard sequential CPU-based methods in pricing these multi-asset options.

It is evident from Table 1 and Figure 5 that the numerical methods are more efficient when applied to pricing a basket option than when applied to a rainbow option. For the same grid sizes, the errors for the basket option are about 3 to 4 times smaller than the errors for the rainbow option. To investigate this further, we look at the payoff functions of the two options. To simplify the analysis, we first consider a two-stock counterpart of (16) and (17), i.e., the European call-on-minimum rainbow option and the European basket option written on two assets with the payoff functions specified by

\[ u(s_1, s_2, T) = \max(\min(s_1, s_2) - E, 0) \]
FIGURE 5: Efficiency comparison of the sequential CPU-based and parallel GPU-based methods applied to the European rainbow and European basket option pricing problems.

and

\[ u(s_1, s_2, T) = \max \left( \sum_{i=1}^{2} w_i s_i - E, 0 \right), \]

respectively. For the basket option on two stocks, the “kink region” for the payoff, i.e., the region where the first derivative of the payoff with respect to the space variables is not continuous, is just the line segment that connects the two points \((E/w_1, 0)\) and \((0, E/w_2)\) in the \(s_1-s_2\) plane. However, for the call-on-minimum rainbow option on two stocks, the kink region consists of two half lines, starting from the point \((E, E)\) and running parallel to the stock value axes. Plots of the kink regions of the two payoff functions are given in Figure 6.

Similarly, for three assets, the kink region for the payoff of the basket option is a plane segment determined by \((E/w_1, 0, 0)\), \((0, E/w_2, 0)\), and \((0, 0, E/w_3)\). However, the kink region for the payoff of the call-on-minimum rainbow option consists of three half planes, starting from the point with coordinates \((s_1, s_2, s_3) = (E, E, E)\) and running parallel to
The kink region for the payoff function of a basket option (left) and of a call-on-minimum rainbow option (right).

FIGURE 6: The kink region for the payoff function of a basket option (left) and of a call-on-minimum rainbow option (right).

the three faces of the rectangular spatial domain. Thus, in a sense, the topology of the payoff function of a European rainbow option is more complex and “harder” to handle than that of a European basket option. This may explain the observed loss of efficiency of the numerical methods applied to pricing a rainbow option.

7 Summary and future work This paper discusses a GPU-based parallel algorithm for solving three-dimensional parabolic PDEs which arise in various applications in finance. The algorithm is built upon an efficient parallelization of the ADI timestepping scheme, which exploits a parallel computation of the explicit Euler predictor step and a parallel solution of independent tridiagonal systems arising in the corrector steps. We illustrate the effectiveness of the method by applying it to price European rainbow and European basket options, each written on three assets. For these multi-asset option pricing problems, for which analytical/reference solutions are given in the literature, we verified that our method is second-order, and that the HV smoothing technique has good damping properties. We obtained a speedup of about 18 with double-precision when comparing the parallel GPU to the optimized sequential CPU computing times.

We conclude the paper by mentioning some extensions of this work. From a numerical methods perspective, support for nonuniform grids with more points concentrated in the regions of interest, such as around
the strike, or in areas where the problem is difficult, such as around the barriers of barrier options, could be added to the current implementation to further increase the accuracy and efficiency of the methods. To achieve even a higher efficiency, one can consider incorporating adaptive techniques, such as those developed in [5], which dynamically adjust the location of the gridpoints to control the error in the approximate solution. However, for such adaptive methods, the stability of the ADI method on a nonuniform mesh needs to be studied. On the other hand, a GPU-based parallelization of higher-order ADI schemes, such as fourth-order schemes, is of much interest. From a parallelization perspective, extending the current implementation to a multi-GPU platform should increase the performance of the GPU algorithm presented here. It would be interesting to investigate the effects of the inter-GPU communication on the total speedup in a context of multi-GPUs. Certainly, one may combine the high-order and adaptive methods with a multi-GPU implementation for an even more efficient pricing framework.

With respect to applications in option pricing, the GPU-based parallelization of the ADI timestepping method discussed in this paper can, after straightforward modifications, be extended to a parallelization of the ADI Approximate Factorization techniques, which can be combined with the discrete penalty approach introduced in [13] for pricing multi-asset American options. A detailed discussion of this approach can be found in [9]. Regarding other application problems in finance, the GPU-based parallel ADI timestepping method presented in this paper can be employed for pricing cross-currency/foreign exchange interest rate derivatives, such as Power-Reverse Dual Currency swaps [32], under a three-factor model consisting of the spot foreign exchange rate and two one-factor Gaussian interest rate models for the domestic and foreign interest short rates. We refer interested readers to [7, 8] for a detailed discussion of such an application.

In many practical financial applications, the pricing models may have significantly more than three stochastic factors. Examples include basket or index options on dozens of stocks, or cross-currency models with multi-factor Gaussian interest rate models and stochastic volatility (at least six stochastic factors). As a result, standard PDE-based pricing approaches, such as those based on the ADI methods presented in this paper, may become infeasible, due to the curse of dimensionality associated with high-dimensional PDEs. In such cases, advanced computational techniques for high-dimensional PDEs, such as sparse grids or high-dimensional approximations [31], could be employed.
Acknowledgment  The authors thank the anonymous referees for their constructive comments and suggestions. This research was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada and was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca).

REFERENCES


CORRESPONDING AUTHOR: D. M. DANG
DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF TORONTO,
TORONTO, ON, M5S 3G4, CANADA
E-mail address: dmdang@cs.toronto.edu

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF TORONTO,
TORONTO, ON, M5S 3G4, CANADA
E-mail address: ccc@cs.toronto.edu

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF TORONTO,
TORONTO, ON, M5S 3G4, CANADA
E-mail address: krj@cs.toronto.edu