Pricing multi-asset American options on Graphics Processing Units using a PDE approach

Duy Minh Dang, Christina C. Christara and Kenneth R. Jackson
Department of Computer Science
University of Toronto, Toronto, ON, M5S 3G4, Canada
{dmdang, ccc, krj}@cs.toronto.edu

***Abstract*—We develop highly efficient parallel pricing methods on Graphics Processing Units (GPUs) for multi-asset American options via a Partial Differential Equation (PDE) approach. The linear complementarity problem arising due to the free boundary is handled by a penalty method. Finite difference methods on uniform grids are considered for the space discretization of the PDE, while classical finite differences, such as Crank-Nicolson, are used for the time discretization. The discrete nonlinear penalized equations at each timestep are solved using a penalty iteration. A GPU-based parallel Alternating Direction Implicit Approximate Factorization technique is employed for the solution of the linear algebraic system arising from each penalty iteration. We demonstrate the efficiency and accuracy of the parallel numerical methods by pricing American options written on three assets.**

***Keywords*-American option, multi-asset, penalty method, Alternating Direction Implicit Approximate Factorization, Graphics Processing Units, GPUs, parallel computing, finite difference**

I. Introduction

The pricing of an American option is a challenging task, mainly due to the early exercise feature of the option, which leads to an additional constraint that the value of an American option must be greater than or equal to its payoff [14]. This constraint requires special treatment, a fact that makes an explicit closed form solution for an American option intractable for most cases. Consequently, numerical methods must be used. Recently, multi-asset options, i.e. options written on more than one underlying asset, have become increasingly popular. The problem of pricing multi-asset American options is not only mathematically challenging but also very computationally intensive. As a result, there is great interest in developing efficient numerical methods for pricing multi-asset American options.

Although several approaches, such as lattice (tree) methods and Monte Carlo simulations, can be used for pricing an American option, for problems in low dimensions, i.e. less than five dimensions, the partial differential equation (PDE) approach is very popular, due to its efficiency, global character and ease in computing accurate hedging parameters, such as delta and gamma. Using a PDE approach, the American option pricing problem can be formulated as a time-dependent linear complementarity problem (LCP) with the inequalities involving the Black-Scholes PDE and some additional constraints [16]. In this paper, we adopt the penalty method of [6] to solve the LCP. In this approach, a penalty term is added to the discretized equations to enforce the early exercise constraint. The solution of the resulting discrete nonlinear equations at each timestep can be computed via a penalty iteration.[1] An advantage of the penalty method of [6] is that it is readily extendable to handle multi-factor problems. In a multi-dimensional application, applying direct methods, such as LU factorization, to solve the linear system arising at each penalty iteration can be computationally expensive. A very popular alternative is to use iterative methods, such as Biconjugate Gradient Stabilized (BiCGStab), in combination with a preconditioning technique, such as an Incomplete LU factorization [14]. Another possible approach is to employ Alternating Direction Implicit Approximate Factorization (ADI-AF) techniques, which involve solving only a few tridiagonal systems in each spatial dimension. It is rather surprising that, while these efficient techniques have been widely used in the numerical solution of multi-dimensional nonlinear PDEs arising in computational fluid dynamics [17], to the best of our knowledge, these techniques have not been successfully extended to multi-asset American option pricing.

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 computational finance, although there has been great interest in utilizing GPUs in developing efficient pricing architectures for computationally intensive problems [1], [4], the existing literature on GPU-based numerical methods for multi-asset American option pricing is rather sparse and mostly focuses on Monte Carlo simulations [1] and quadrature integrations [15]. The literature on GPU-based PDE methods for multi-asset American options is even less developed. In addition, to the best of our knowledge, a combination of an efficient GPU-based parallelization of ADI-AF techniques with a penalty approach for the pricing of multi-asset American options has not been previously discussed in the literature. These shortcomings motivated our work.

This paper discusses the application of GPUs to price multi-factor American options in the Black-Scholes framework via a PDE approach. Our approach is built upon the penalty method of [6] and an efficient GPU-based parallel ADI-AF algorithm for solving the linear algebraic system arising at each penalty iteration. Finite difference (FD) methods on uniform grids are considered for the space discretization of the pricing PDE, while classical finite differences, such as Crank-Nicolson, are used

[1] The penalty iteration described in [6] is essentially a Newton iteration, but, to be consistent with [6], we use the term penalty iteration throughout this paper.

Copyright 978-1-4244-9061-5/10/$26.00 ©2010 IEEE

for its time discretization. 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 multi-factor options with early exercise features. Although we primarily focus on a three-factor model, many of the ideas and results in this paper can be naturally extended to higher-dimensional applications with limitations.

The remainder of this paper is organized as follows. Section II presents a PDE formulation of the pricing problem for a multi-asset American option and the discretization methods. We restrict our attention to American put options on the arithmetic average of three underlying assets. A penalty iteration for the discretized American option and an associated ADI-AF scheme are discussed in Section III. Section IV discusses a GPU-based parallel implementation of the ADI-AF methods. Numerical results and related discussions are presented in Section V. Section VI concludes the paper and outlines possible future work.

II. Formulation of the pricing PDE and discretization

A. Formulation

We denote by $s_i(t)$, $i = 1, 2, 3$, the value at time t of the ith underlying asset, by T the expiry time of the option, and by $\tau = T - t$ the time to expiry. For simplicity, let $\mathbf{s} = (s_1, s_2, s_3)$. The early exercise constraint leads to the following LCP for the value $u(\mathbf{s}, \tau)$ of an American put option [16]

$$\left\{ \begin{array}{c} \frac{\partial u}{\partial \tau} - \mathcal{L}u = 0 \\ u - u^* \geq 0 \end{array} \right\} \text{ and } \left\{ \begin{array}{c} \frac{\partial u}{\partial \tau} - \mathcal{L}u > 0 \\ u - u^* = 0 \end{array} \right\}, \tag{1}$$
$$\mathbf{s} \in \Omega \equiv (0, s_{1,\infty}) \times (0, s_{2,\infty}) \times (0, s_{3,\infty}), \tau \in (0, T],$$

subject to the initial (payoff) condition

$$u^*(\mathbf{s}) = u(\mathbf{s}, 0) = \max\left(E - \sum_{i=1}^{3} w_i s_i, 0\right) \quad \text{on} \quad (\partial\Omega \cup \Omega) \times \{0\}, \tag{2}$$

and the boundary conditions [10]

$$u(\mathbf{s}, t) = u^*(\mathbf{s}) \quad \text{on} \quad \partial\Omega \times (0, T], \tag{3}$$

where

$$\mathcal{L}u \equiv \frac{1}{2} \sum_{i,j=1}^{3} \rho_{ij} \sigma_i \sigma_j s_i s_j \frac{\partial^2 u}{\partial s_i \partial s_j} + \sum_{i=1}^{3} (r - d_i) s_i \frac{\partial u}{\partial s_i} - ru. \tag{4}$$

Here, $\partial\Omega$ is the boundary of Ω; $r > 0$ is the constant riskless interest rate; $d_i \geq 0$ is the constant asset dividend yield; $\sigma_i \geq 0$ is the constant volatility of the ith underlying asset; ρ_{ij} is the correlation factor between the ith and jth assets satisfying $|\rho_{ij}| \leq 1$ for $i, j = 1, 2, 3$, and $\rho_{ii} = 1$ for $i = 1, 2, 3$; $E > 0$ is the strike price of the option, $w_i > 0$ is the weight of the ith asset in the basket, and $s_{i,\infty}$, $i = 1, 2, 3$, is the right boundary of the spatial domain of the ith underlying asset. In the exact mathematical formulation of the problem, $s_{i,\infty} = \infty$, but, in the numerical approximation, we truncate the semi-infinite domain and take $s_{i,\infty}$ to be an appropriately chosen large value, as is explained in more detail in Section V.

Following [6], with a penalty parameter ζ, $\zeta \to \infty$, we consider the non-linear PDE for the penalty formulation of the price $u(\mathbf{s}, \tau)$ of an American put option written on three underlying assets

$$\frac{\partial u}{\partial \tau} - \mathcal{L}u = \zeta \max(u^* - u, 0), \quad \mathbf{s} \in \Omega, \tau \in [0, T], \tag{5}$$

subject to the initial and boundary conditions (2) and (3), respectively. The penalty parameter ζ effectively ensures that the solution satisfies $u - u^* \geq -\epsilon$ for $0 < \epsilon \ll 1$. Essentially, in the region where $u \geq u^*$, the PDE (5) resembles the three-dimensional (3-D) Black-Scholes equation. On the other hand, when $-\epsilon \leq u - u^* < 0$, the 3-D Black-Scholes inequality $\frac{\partial u}{\partial \tau} - \mathcal{L}u > 0$ is satisfied and $u \approx u^*$.

B. Discretization

For the discretization of the space variables in the differential operator $\mathcal{L}u$, we employ second-order FD schemes in the rectangular domain Ω. Let the number of subintervals be $n+1$, $p+1$, $q+1$ and l, in the s_1-, s_2-, s_3- and τ-directions, respectively. The uniform grid mesh widths in the respective direction are denoted by $\Delta s_1 = \frac{s_{1,\infty}}{n+1}$, $\Delta s_2 = \frac{s_{2,\infty}}{p+1}$, $\Delta s_3 = \frac{s_{3,\infty}}{q+1}$, and $\Delta\tau = \frac{T}{l}$. The gridpoint values of a FD approximation are denoted by

$$u_{i,j,k}^m \approx u(s_{1i}, s_{2j}, s_{3k}, \tau_m) = u(i\Delta s_1, j\Delta s_2, k\Delta s_3, m\Delta\tau),$$

where $i = 0, \ldots, n+1$, $j = 0, \ldots, p+1$, $k = 0, \ldots, q+1$, $m = 0, 1, \ldots, l$. Second-order FD approximations to the first and second partial derivatives of the space variables in (4) are obtained by *central* schemes, while the cross-derivatives are approximated by a four-point FD stencil. For example, at the reference point $(s_{1i}, s_{2j}, s_{3k}, \tau_m)$, $\frac{\partial u}{\partial s_1}$ and $\frac{\partial^2 u}{\partial s_1^2}$ are respectively approximated by

$$\frac{\partial u}{\partial s_1} \approx \frac{u_{i+1,j,k}^m - u_{i-1,j,k}^m}{2\Delta s_1}, \frac{\partial^2 u}{\partial s_1^2} \approx \frac{u_{i+1,j,k}^m - 2u_{i,j,k}^m + u_{i-1,j,k}^m}{(\Delta s_1)^2}, \tag{6}$$

while the cross derivative $\frac{\partial^2 u}{\partial s_1 \partial s_2}$ is approximated by

$$\frac{u_{i+1,j+1,k}^m + u_{i-1,j-1,k}^m - u_{i-1,j+1,k}^m - u_{i+1,j-1,k}^m}{4\Delta s_1 \Delta s_2}. \tag{7}$$

Similar approximations can be obtained for the remaining spatial derivatives. For brevity, we omit the derivations of (6) and (7), but, using Taylor expansions, it can be verified that each of these formulas has a second-order truncation error, provided that the function u is sufficiently smooth. The FD discretization of the spatial differential operator $\mathcal{L}$ of (5) is performed as follows. At the spatial grid Ω, each spatial derivative appearing in the operator $\mathcal{L}$ is replaced by its corresponding FD scheme (as in (6) and (7)). We denote by $\mathcal{L}u_{i,j,k}^m$ the FD discretization of $\mathcal{L}$ at $(s_{1i}, s_{2j}, s_{3k}, \tau_m)$.

To proceed from time τ_{m-1} to time τ_m, we apply the standard θ-timestepping discretization scheme to (5)

$$(\mathcal{I} - \theta\Delta\tau\mathcal{L})u_{i,j,k}^m = (\mathcal{I} + (1-\theta)\Delta\tau\mathcal{L})u_{i,j,k}^{m-1} + \mathcal{P}u_{i,j,k}^m, \tag{8}$$

where $0 \leq \theta \leq 1$, and incorporate the boundary conditions (3) by setting $u_{i,j,k}^m = u_{i,j,k}^*$, if $i = \{0, n+1\}$, or $j = \{0, p+1\}$, or $k = \{0, q+1\}$, with $u_{i,j,k}^*$ being the payoff value at the reference point $(s_{1i}, s_{2j}, s_{3k}, \cdot)$. Here, $\mathcal{I}$ and $\mathcal{P}$ denote the identity and penalty operators, respectively, where $\mathcal{P}$ is defined by

$$\mathcal{P}u_{i,j,k}^m = \zeta \max(u_{i,j,k}^* - u_{i,j,k}^m, 0).$$

In (8), the values $\theta = 1/2$ and $\theta = 1$ give rise to the standard Crank-Nicolson (CN) and the fully-implicit methods, respectively. It is known that the CN method is second-order accurate, but prone to producing spurious oscillations, while the fully-implicit method is first-order accurate, but maintains strong stability properties (e.g. [12]). To maintain the accuracy of CN as well as smoothness of the solution, we use the Rannacher smoothing technique [13], which applies the fully-implicit timestepping in the first few (usually two) timesteps followed by the CN method on the remaining timesteps. We adapt the penalty iteration algorithm in [6] to solve the set of discrete nonlinear penalized equations (8). In the following section, we present the penalty iteration algorithm and an associated ADI-AF scheme.

III. Penalty iteration and an associated ADI-AF technique

Unless otherwise stated, assume that the mesh points are ordered in the s_1-, s_2- , then s_3- directions. Let $\mathbf{u}^m$ denote the vector of values at time τ_m on the mesh Ω that approximates the exact solution $u^m = u(\mathbf{s}, \tau_m)$. Furthermore, denote by $\mathbf{u}^*$ the vector of the payoff values on Ω. Let κ, $\kappa \geq 0$, be the index of the penalty iteration. Let $\mathbf{u}^{m,(\kappa)}$ be the κth estimate of $\mathbf{u}^m$. The initial guess $\mathbf{u}^{m,(0)}$ is chosen to be the numerical solution at the previous timestep. At each penalty iteration, we need to solve an $npq \times npq$ algebraic system of the form [6]

$$(\mathbf{I} + \theta\Delta\tau\mathbf{A} + \mathbf{P}^{m,(\kappa)})\mathbf{u}^{m,(\kappa+1)} = (\mathbf{I} - (1-\theta)\Delta\tau\mathbf{A})\mathbf{u}^{m-1} + \mathbf{P}^{m,(\kappa)}\mathbf{u}^* + \Delta\tau\mathbf{g}. \quad (9)$$

Here, $\mathbf{I}$ denotes the identity matrix; $-\mathbf{A}$ is the matrix FD approximation to the differential operator $\mathcal{L}$; $\mathbf{P}^{m,(\kappa)}$ is the diagonal penalty matrix and $\mathbf{g}$ is a vector containing values arising from the boundary conditions. For brevity, we omit the explicit formula for $\mathbf{A}$. The penalty matrix $\mathbf{P}^{m,(\kappa)}$ is defined by

$$\left(\mathbf{P}^{m,(\kappa)}\right)_{ij} \equiv \begin{cases} \zeta & \text{if} \quad \mathbf{u}_i^{m,(\kappa)} < \mathbf{u}_i^* \quad \text{and} \quad i = j, \\ 0 & \text{otherwise.} \end{cases} \quad (10)$$

Note that the matrix on the left hand size of (9) is essentially the Jacobian of the discrete nonlinear penalized system arising from (8). In general, if we want (5) to be solved with a relative precision *tol*, we should have $\zeta \simeq \frac{1}{tol}$ [2]. For future use, we decompose the matrix $\mathbf{A}$ into four submatrices

$$\mathbf{A} = \mathbf{A}_0 + \mathbf{A}_1 + \mathbf{A}_2 + \mathbf{A}_3.$$

The matrices $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_3$ are the parts of $\mathbf{A}$ that correspond to the spatial derivatives in the s_1-, s_2- and s_3-directions, respectively, while the matrix $\mathbf{A}_0$ is the part of $\mathbf{A}$ that comes from the FD discretization of the mixed derivative terms in the operator $\mathcal{L}$. The term ru in $\mathcal{L}$ is distributed evenly over $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_3$. For simplicity, let $\mathbf{D}^{m,(\kappa)} = \mathbf{I} + \mathbf{P}^{m,(\kappa)}$.

We adapt the ADI-AF approach discussed in [17] to solve (9) by first writing an ADI-AF scheme for (9) in the form

$$\begin{aligned}
&\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_1\right)\left(\mathbf{D}^{m,(\kappa)}\right)^{-1}\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_2\right)\left(\mathbf{D}^{m,(\kappa)}\right)^{-1} \\
&\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_3\right)\mathbf{u}^{m,(\kappa+1)} = (\mathbf{I} - (1-\theta)\Delta\tau\mathbf{A})\mathbf{u}^{m-1} \\
&+ \mathbf{P}^{m,(\kappa)}\mathbf{u}^* + \Delta\tau\mathbf{g} \\
&+ \left(\mathbf{D}^{m,(\kappa)}\right)^{-1}(\theta\Delta\tau)^2\left(\mathbf{A}_1\mathbf{A}_2 + \mathbf{A}_1\mathbf{A}_3 + \mathbf{A}_2\mathbf{A}_3\right)\mathbf{u}^{m,(\kappa)} \\
&+ \left(\mathbf{D}^{m,(\kappa)}\right)^{-2}(\theta\Delta\tau)^3\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{u}^{m,(\kappa)} - \theta\Delta\tau\mathbf{A}_0\mathbf{u}^{m,(\kappa)}.
\end{aligned} \quad (11)$$

We then subtract $\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_1\right)\left(\mathbf{D}^{m,(\kappa)}\right)^{-1}\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_2\right)\left(\mathbf{D}^{m,(\kappa)}\right)^{-1}\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_3\right)\mathbf{u}^{m,(\kappa)}$ from both sides of (11). The resulting ADI-AF scheme for the correction $\Delta\mathbf{u}^{m,(\kappa)}$, where $\Delta\mathbf{u}^{m,(\kappa)} = \mathbf{u}^{m,(\kappa+1)} - \mathbf{u}^{m,(\kappa)}$, is given by

$$\begin{aligned}
&\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_1\right)\left(\mathbf{D}^{m,(\kappa)}\right)^{-1}\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_2\right)\left(\mathbf{D}^{m,(\kappa)}\right)^{-1} \\
&\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_3\right)\Delta\mathbf{u}^{m,(\kappa)} = -(\mathbf{I} + \theta\Delta\tau\mathbf{A})\mathbf{u}^{m,(\kappa)} \\
&+ (\mathbf{I} - (1-\theta)\Delta\tau\mathbf{A})\mathbf{u}^{m-1} + \mathbf{P}^{m,(\kappa)}\left(\mathbf{u}^* - \mathbf{u}^{m,(\kappa)}\right) + \Delta\tau\mathbf{g},
\end{aligned} \quad (12)$$

which can be equivalently written as the following steps:

$$\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_1\right)\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(1)} = \mathbf{b}^{m,(\kappa)}, \quad (13.1)$$

$$\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_2\right)\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(2)} = \mathbf{D}^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(1)}, \quad (13.2)$$

$$\left(\mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_3\right)\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(3)} = \mathbf{D}^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(2)}, \quad (13.3)$$

$$\Delta\mathbf{u}^{m,(\kappa)} = \left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(3)}. \quad (13)$$

Here, for simplicity, we denote by $\mathbf{b}^{m,(\kappa)}$ the right-hand-side of (12). The corresponding ADI-AF FD penalty algorithm is presented in Algorithm 1.

REMARK 1: Due to the similarities between the ADI-AF scheme (12) and an ADI method, such as the Douglas and Rachford scheme [9], the fact that the mixed derivatives are treated solely explicitly in (12) would normally lead to an expectation that second-order of convergence of the numerical methods would be lost. This is a typical problem of ADI methods (e.g. see [9]). However, as the numerical results indicate, the ADI-AF scheme (12) exhibits second-order convergence. This does not contradict with the aforementioned problem of ADI methods, since these ADI methods are used in a non-iterative context, whereas, in our case, the ADI-AF scheme is applied iteratively. While the first iterate $\mathbf{u}^{m,(1)}$ could be a first-order accurate approximate solution, it seems that, with

Algorithm 1: ADI-AF FD penalty iteration for American options

1: initialize $\mathbf{u}^{m,(0)}$;
2: construct $\mathbf{P}^{m,(0)}$ using (10)
3: **for** $\kappa = 0, \ldots,$ until convergence **do**
4: solve (13) for $\Delta \mathbf{u}^{m,(\kappa)}$;
set $\mathbf{u}^{m,(\kappa+1)} = \mathbf{u}^{m,(\kappa)} + \Delta \mathbf{u}^{m,(\kappa)}$;
5: construct $\mathbf{P}^{m,(\kappa+1)}$ using (10)
6: **if** $\left[\max_{1 \le i \le npq} \left\{ \frac{|\mathbf{u}_i^{m,(\kappa+1)} - \mathbf{u}_i^{m,(\kappa)}|}{\max(1, |\mathbf{u}_i^{m,(\kappa+1)}|)} \right\} < tol \right]$ or $\left[\mathbf{P}^{m,(\kappa)} = \mathbf{P}^{m,(\kappa+1)} \right]$ **then**
7: break;
8: **end if**
9: **end for**
10: $\mathbf{u}^{m+1} = \mathbf{u}^{m,(\kappa+1)}$;

further penalty iterations, $\mathbf{u}^{m,(\kappa)}$ can converge to a second-order accurate approximate solution at each timestep. Detailed results are given in Tables I and II, with a discussion in Section V.

REMARK 2: There are at least two possible approaches to extend the ADI-AF scheme (12). In the first approach, one can develop an ADI-AF scheme based on the second-order backward differentiation formula (BDF2) for the time derivative

$$\frac{\partial u}{\partial \tau} = \frac{3u^m - 4u^{m-1} + u^{m-2}}{2\Delta\tau} + \mathcal{O}((\Delta\tau)^2).$$

While having approximately the same computational cost per penalty iteration as the ADI-AF scheme (12), an ADI-AF scheme based on this approach takes advantage of the numerical solution available from the previous two timesteps i.e. the timesteps $m-1$ and $m-2$, to compute the numerical solution at the current timestep, i.e. the timestep m. In this case, the initial guess $\mathbf{u}^{m,(0)}$ can be given by the second-order linear two-level extrapolation $2\mathbf{u}^{m-1} - \mathbf{u}^{m-2}$. Note that, in the context of pricing American put options written on one asset using the same penalty approach, extensive experiments have shown that using as initial guess for the penalty iteration an approximation based on linear two-level extrapolation of the numerical solutions from the previous two timesteps gives rise to a more efficient technique than using the standard choice of the numerical solution at the previous timestep [2]. Thus, we expect that an ADI-AF scheme based on the BDF2 formula may converge faster, i.e. using a smaller average number of penalty iterations per timestep, than the ADI-AF scheme presented in this paper.

In the second approach, to maintain the second-order accuracy for $\mathbf{u}^{m,(\kappa)}$ at each penalty iteration, a special treatment to the cross-derivative terms, similar to those suggested in [3] or [8] in the context of ADI timestepping methods, could be added to the scheme (12). However, a scheme based on this approach entails solving an additional tridiagonal linear system along each spatial dimension, similar to (13.1) - (13.3), possibly with different right-side vectors. That is, the computational cost per penalty iteration of an ADI-AF scheme based on this approach is approximately double that of the scheme (12). It is not obvious whether an ADF-AF scheme based on this approach would be more efficient than the scheme (12). We plan to investigate the efficiency of the aforementioned two approaches in a future paper.

REMARK 3: The FD discretization for the spatial variables described in (6) implies that, if the gridpoints are ordered appropriately, all the linear systems in (13) are block-diagonal with tridiagonal blocks. As a result, the number of floating-point operations per iteration 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 solution of the linear system (13). However, it is less obvious how the computation of the vector $\mathbf{b}^{m,(\kappa)}$ can be efficiently parallelized. We address this point in more detail in the following section.

IV. GPU IMPLEMENTATION OF THE ADI-AF SCHEME

A. *GPU device architecture*

The modern GPU can be viewed as a set of independent streaming multiprocessors (SMs), each of which contains, amongst others, several scalar processors which can execute floating-point arithmetic, a multi-threaded instruction unit, and shared memory which can be accessed by all scalar processors of a multiprocessor. In addition, the global (device) memory in a GPU 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. Functions that run on the GPUs are called *kernels*. When a program invokes a kernel, many copies of this kernel, referred to as *threads*, are distributed to the available multiprocessors, where they are executed. We use the programming model of CUDA, which is an instance of the widely used Single Instruction Multiple Data (SIMD) parallel programming style. Within the CUDA framework, threads are grouped into *threadblocks*, which are in turn arranged on a *grid*. It is important to note that threads within the same threadblock are able to communicate with each other very efficiently via the shared memory or 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. For a more detailed description of the GPU, see [11].

Due to various hardware considerations, the current generation of CUDA devices bundles multiple threads for execution. To optimize performance, it is important to ensure coalesced data loads from the global memory. Since the CUDA memory consists of 16 memory banks, to achieve maximum memory performance, memory accesses at any one time are handled by groups of 16 threads, referred to as *half-warps*, each thread accessing a different bank. Global memory access by threads in a half-warp is coalesced if (i) the threads in the half-warp access consecutive global memory locations and (ii) the number of threads along the first dimension of the threadblock is a multiple of 16. We refer interested readers to [11] for a more complete discussion of all the requirements.

The NVIDIA Tesla series is the first family of GPUs that is dedicated to general purpose computing. The NVIDIA Tesla 10-series (T10) GPUs (Tesla S1060/S1070 - server version), which are used for the experiments in this paper, consists of 30 independent SMs, each containing 8 processors running at 1.44GHz, a total of 16384 registers, and 16 KB of shared memory.

B. GPU implementation

We now discuss a GPU-based parallel algorithm for each of the penalty iterations of Algorithm 1. In particular, we focus on describing the parallel implementation of the ADI-AF scheme (13) (Line 4) and the stopping criterion (Line 6) of the penalty algorithm. For presentation purposes, let

$$\begin{aligned}
\mathbf{w}^{m-1} &= (1-\theta)\Delta\tau\mathbf{A}\mathbf{u}^{m-1}, \\
\mathbf{w}^{(\kappa)} &= \theta\Delta\tau\mathbf{A}\mathbf{u}^{m,(\kappa)}, \\
\widehat{\mathbf{A}}_i^{m,(\kappa)} &= \mathbf{D}^{m,(\kappa)} + \theta\Delta\tau\mathbf{A}_i, \quad i=1,2,3, \\
\widehat{\Delta\mathbf{u}}^{(\kappa),i} &= \mathbf{D}^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(i-1)}, \quad i=2,3,
\end{aligned}$$

and notice that

$$\begin{aligned}
\mathbf{b}^{m,(\kappa)} = \mathbf{u}^{m-1} - \mathbf{u}^{m,(\kappa)} - (\mathbf{w}^{m-1} + \mathbf{w}^{(\kappa)}) \\
+ \mathbf{P}^{m,(\kappa)}\left(\mathbf{u}^* - \mathbf{u}^{m,(\kappa)}\right) + \Delta\tau\mathbf{g}.
\end{aligned}$$

Here, to simplify the notation, we do not indicate the superscript for the timestep index of the vectors $\mathbf{w}^{(\kappa)}$ and $\widehat{\Delta\mathbf{u}}^{(\kappa),i}$, $i = 2, 3$. The computation of the ADI-AF scheme (13) and the checking of the stopping criterion of Algorithm 1 consist of the following steps:

(i) Step a.1: Compute the matrices $\mathbf{A}_i$, $\widehat{\mathbf{A}}_i^{m,(\kappa)}$, $i = 1, 2, 3$, and $\mathbf{D}^{m,(\kappa)}$, and the vectors $\mathbf{w}^{m-1}$, $\mathbf{w}^{(\kappa)}$ and $\mathbf{b}^{m,(\kappa)}$;

(ii) Step a.2: Solve $\widehat{\mathbf{A}}_1^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(1)} = \mathbf{b}^{m,(\kappa)}$;

(iii) Step a.3: Compute $\widehat{\Delta\mathbf{u}}^{(\kappa),2}$ and solve $\widehat{\mathbf{A}}_2^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(2)} = \widehat{\Delta\mathbf{u}}^{(\kappa),2}$;

(iv) Step a.4: Compute $\widehat{\Delta\mathbf{u}}^{(\kappa),3}$ and solve $\widehat{\mathbf{A}}_3^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(3)} = \widehat{\Delta\mathbf{u}}^{(\kappa),3}$;

(v) Step a.5: Check the stopping criterion.

In [5], a past work of ours, we present a parallel ADI timestepping method implemented efficiently on GPUs for the solution of multi-dimensional linear parabolic PDEs. We observe similarities between the computation of the scheme (13) and that of the ADI timestepping method in [5]. More specifically, the computation of the vector $\mathbf{b}^{m,(\kappa)}$ in Step a.1 resembles the explicit Euler predictor step, while Steps a.2-a.4 are essentially the same as the three implicit corrector steps in [5], each of which involves solving a block-diagonal system with tridiagonal blocks along a spatial dimension. The GPU-based parallelization of the ADI-AF scheme considered in this paper can be viewed as a natural extension of the parallelization of the ADI timestepping method presented in [5]. For brevity, we only present the main steps of the parallel algorithm for the ADI-AF scheme (13). A more detailed discussion of the similar ADI algorithm can be found in [5].

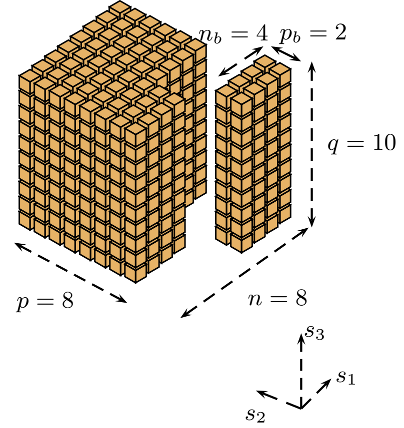

Figure 1. An illustration of the partitioning approach considered for Step a.1. The computational domain is partitioned into 3-D blocks of size $n_b \times p_b \times q \equiv 4 \times 2 \times 10$, each of which can be viewed as consisting of ten 4×2 tiles, or as $8 (= 4 \times 2)$ stacks of 10 gridpoints.

1) Step a.1: We assume that, initially, the vectors $\mathbf{u}^{m-1}$, $\mathbf{u}^{m,(\kappa)}$ and $\mathbf{u}^*$ are in the global memory, and any needed constants (model parameters) are in the constant cache. Note that the data copying from the host memory to the device memory occurs on the first timestep only, for the initial condition (payoff) data and the model constants. Data for the subsequent timesteps and steps of the ADI-AF scheme (13) are stored in the global memory.

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 s_3-direction as a "stack" of q gridpoints. 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 $\mathbf{A}_i$, $\widehat{\mathbf{A}}_i^{m,(\kappa)}$, $i = 1, 2, 3$, (or $\mathbf{P}^{m,(\kappa)}$) and components of vectors $\mathbf{w}^{m-1}$, $\mathbf{w}^{(\kappa)}$ and $\mathbf{b}^{m,(\kappa)}$) to a different thread. More specifically, we partition the computational grid of size $n \times p \times q$ into 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 $\texttt{ceil}\left(\frac{n}{n_b}\right) \times \texttt{ceil}\left(\frac{p}{p_b}\right)$ grid of threadblocks, where `ceil` denotes the ceiling function. Each of the threadblocks, in turn, consists of a total of $n_b p_p$ 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 in the s_3 direction (see Figure 1).

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. While the construction of the matrices $\mathbf{A}_i$ and $\widehat{\mathbf{A}}_i^{m,(\kappa)}$, $i = 1, 2, 3$, (or $\mathbf{P}^{m,(\kappa)}$) is relatively simple, the computation of matrix-vector multiplications embedded in $\mathbf{w}^{m-1}$ and $\mathbf{w}^{(\kappa)}$ is more complicated. More specifically, due to the FD schemes (6) and (7), at each iteration, a threadblock carrying the computation of a tile needs the values of neighbouring gridpoints from adjacent tiles in the s_1 and s_2 directions, referred to as *halo* values. Since

different tiles belong to different threadblocks, it is essential that the halo values that a threadblock needs be loaded into the shared memory as well. Because 16KB of shared memory available per multiprocessor are not sufficient to store many data tiles, each threadblock works with three data tiles of size $n_b \times p_b$ and their halo values at a time and proceeds in the s_3-direction. More specifically, during the kth iteration of the q-iteration loop in the kernel, each threadblock

1. loads from the global memory into its shared memory the old data (vectors $\mathbf{u}^{m-1}$, $\mathbf{u}^{m,(\kappa)}$ and $\mathbf{u}^{*}$) corresponding to the $(k+1)$st tile, and the associated halos (in the s_1- and s_2-directions), if any,
2. computes and stores new values of vectors $\mathbf{w}^{m-1}$, $\mathbf{w}^{(\kappa)}$ and $\mathbf{b}^{m,(\kappa)}$, and new rows of the matrices $\mathbf{D}^{m,(\kappa)}$, $\mathbf{A}_i$ and $\widehat{\mathbf{A}}_i^{m,(\kappa)}$, $i = 1, 2, 3$ (a total of three entries per row - by computing the corresponding rows of $\mathbf{P}^{m,(\kappa)}$, a total of one entry per row, and of $\mathbf{A}_i$, a total of three entries per row) corresponding the kth tile using data of the $(k-1)$st, kth and $(k+1)$st tiles, and of the associated halos, if any,
3. copies the newly computed data of the kth tile (vectors $\mathbf{w}^{m-1}$ and $\mathbf{b}^{m,(\kappa)}$ and matrices $\mathbf{D}^{m,(\kappa)}$, $\mathbf{A}_i$ and $\widehat{\mathbf{A}}_i^{m,(\kappa)}$, $i = 1, 2, 3$) 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.

REMARK 4: Regarding the GPU computation for the vectors $\mathbf{w}^{m-1}$ and $\mathbf{w}^{(\kappa)}$, to ensure the data transfer coalescing, it is necessary to have the tile size in the s_1-direction, i.e. n_b, be a multiple of 16, since each half-warp is of size 16 and that gridpoints at Step a.1 are ordered in the s_1-direction first. (See a discussion in Subsection IV-A.) Under these conditions, the data loading strategy described above allows the interior and the halos along the s_1-direction of the data tiles to be read from global memory into the shared memory in a coalesced way. However, halos along the s_2-direction cannot be accessed via a coalesced pattern. The reader is referred to [5] for a more detailed discussion. It is worth emphasizing that the vector $\mathbf{w}^{m-1}$ is computed only in the first penalty iteration of the mth timestep. This vector is then loaded in a coalesced fashion from the global memory to the shared memory for use in subsequent penalty iterations of that timestep. As experimental results indicate, although the data loading approach for Step a.1 is not fully coalesced, it is highly effective.

REMARK 5: Note that the matrices $\mathbf{A}_i$, $i = 1, 2, 3$, do not depend on the timestep and iteration indices. Hence, after these matrices have been constructed in the first penalty iteration of the first timestep, they can be loaded (via a coalesced pattern) from the global to the shared memory for use in Step a.1 of the subsequent penalty iterations.

2) Steps a.2, a.3, a.4: The data partitioning for Steps a.2, a.3 and a.4 is different from that for Step a.1 and is motivated by the block structure of the tridiagonal matrices $\widehat{\mathbf{A}}_i^{m,(\kappa)}$, $i = 1, 2, 3$. For example, $\widehat{\mathbf{A}}_1^{m,(\kappa)}$ has pq diagonal blocks, and each block is tridiagonal of size $n \times n$, while $\widehat{\mathbf{A}}_2^{m,(\kappa)}$ has nq diagonal blocks, and each block is tridiagonal of size $p \times p$. Each of these independent tridiagonal systems is assigned to a different thread. For example, the solution of $\widehat{\mathbf{A}}_1^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(1)} = \mathbf{b}^{m,(\kappa)}$ (Step a.2) is computed by first partitioning $\widehat{\mathbf{A}}_1^{m,(\kappa)}$ and $\mathbf{b}^{m,(\kappa)}$ into pq independent $n \times n$ tridiagonal systems, and then assigning each tridiagonal system to one of the pq threads generated. 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 $\widehat{\mathbf{A}}_1^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(1)} = \mathbf{b}^{m,(\kappa)}$, a 2-D grid of threadblocks of size $\mathtt{ceil}(\frac{p}{r_t}) \times \mathtt{ceil}(\frac{q}{c_t})$ is invoked.

As an example, a summary of the Step a.2 is given below: A grid of $\mathtt{ceil}(p/r_t) \times \mathtt{ceil}(q/c_t)$ threadblocks is invoked, each of which consists of an $r_t \times c_t$ array of threads. Each thread is assigned n points along the s_1-direction.
Each threadblock

1. loads from the global memory to its shared memory its rows of $\widehat{\mathbf{A}}_1^{m,(\kappa)}$ and its components of $\mathbf{b}^{m,(\kappa)}$;
2. solves $r_t c_t$ tridiagonal $n \times n$ systems (its part of $\widehat{\mathbf{A}}_1^{m,(\kappa)}\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(1)} = \mathbf{b}^{m,(\kappa)}$), with each thread solving one system;
3. copies its newly computed components of $\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(1)}$ from its shared memory to the global memory.

3) Step a.5: Although we can launch a separate kernel to check the stopping criterion after Step a.4 has completed, in the current implementation, the checking of the stopping criterion is done during the kernel generated in Step a.4. More specifically, each threadblock of the kernel launched in Step a.4 needs to load its components of the vectors $\mathbf{u}^{m,(\kappa)}$ and $\mathbf{u}^{*}$ (in addition to its rows of the matrices $\mathbf{D}^{m,(\kappa)}$ and $\mathbf{A}_3^m$, and its components of the vector $\left(\Delta\mathbf{u}^{m,(\kappa)}\right)^{(2)}$ for the solution of the independent tridiagonal systems). After each thread of a threadblock computes its component of the vector $\widehat{\Delta\mathbf{u}}^{(\kappa),3}$ corresponding to the reference point $(s_{1i}, s_{2j}, s_{3k}, \cdot)$, it computes the quantity

$$\frac{|u_{i,j,k}^{m,(\kappa+1)} - u_{i,j,k}^{m,(\kappa)}|}{\max(1, |u_{i,j,k}^{m,(\kappa+1)}|)} \tag{14}$$

and the corresponding row of the penalty matrix $\mathbf{P}^{m,(\kappa+1)}$ (one entry). If the quantity (14) is greater than or equal to the tolerance *tol*, the thread then changes the pre-set value of a flag variable stored in a global memory location. Similarly, if two corresponding rows of the matrices $\mathbf{P}^{m,(\kappa)}$ (obtained from the matrix $\mathbf{D}^{m,(\kappa)}$) and $\mathbf{P}^{m,(\kappa+1)}$ are different, the thread then changes the pre-set value of another flag variable stored in a different global memory location. Note that the two pre-set flag variables are copied from the host memory to the device memory before the kernel of Step a.4 is launched. After the kernel has ended, the values of the two flag variables are copied back to the host memory to be checked. (These host-device copies are cheap.) The stopping criterion is satisfied if the two pre-set values were not altered during the kernel. Although it may happen that multiple threads try to write to the

same memory location of a flag variable at the same time, it is guaranteed that one of the writes will occur. Although we do not know which one, for the purpose of checking the stopping criterion, this approach suffices and works well.

REMARK 6: In the current implementation, the data between Steps a.1, a.2, a.3 and a.4 are ordered in the s_1-, then s_2-, then s_3-directions. As a result, memory coalescence is fully achieved only for the tridiagonal solves in the s_2- and s_3-directions, but not in the s_1-direction. (Hence, the loading of components of the vectors $\mathbf{u}^{m,(\kappa)}$ and $\mathbf{u}^*$ used for the checking of the stopping criterion in Step a.5 is fully coalesced.) See [5] for a more detailed discussion.

V. NUMERICAL RESULTS

We use the set of parameters for three assets taken from [10]: $E=100, r=0.03, T=0.25$, $\sigma_1=\sigma_2=\sigma_3=0.2$, $d_1=d_2=d_3=0$, $\rho_{12}=\rho_{13}=\rho_{23}=0.5$. The spot prices are chosen to be $s_1(0)=s_2(0)=s_3(0)=E$. We consider the weights of the assets to be $w_1=w_2=w_3=\frac{1}{3}$, so that we have $\sum_{i=1}^{3} w_i s_i(0) = \frac{1}{3}\sum_{i=1}^{3} s_i(0) = E$. The penalty parameter $\zeta = 10^7$ is used. We choose $s_{1,\infty}=s_{2,\infty}=s_{3,\infty}=3E=300$.

We used the CUDA 3.1 driver and toolkit, and all the experiments with the GPU code were conducted on a NVIDIA Tesla T10 connected to a two quad-core Intel "Harpertown" host system with Intel Xeon E5430 CPUs running at 2.66GHz, 8GB of FBM PC 5300 RAM. However, only one CPU core was employed for the experiments with the CPU code, i.e. the CPU version of the GPU code of the ADI-AF methods to price multi-asset American options, written by us. Note that the CPU code is not multi-threaded. The CPU and GPU computation times, respectively denoted by "CPU time" and "GPU time", measure the total computational times using the functions `cutStartTimer()` and `cutStopTimer()`. The GPU times include the overhead for memory transfers from the CPU to the device memory. All computations are carried out in double-precision. 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 and a.4 is $r_t \times c_t \equiv 32 \times 4$, which appears to be optimal on a Tesla T10.

The quantity "speedup" is defined as the ratio of the CPU time over the corresponding GPU time. The quantity "value" denotes the spot value of the option, and the quantity "error" is computed as the absolute difference between our numerical solutions and an accurate reference solution. To show convergence, we compute the quantity "$\log_\eta$ ratio" which is defined by $\log_\eta$ ratio $=\log_\eta\left(\frac{u_{ref}-u_{approx}(\Delta s)}{u_{ref}-u_{approx}(\frac{\Delta s}{\eta})}\right)$, where u_{ref} is an accurate reference solution. We denote by "iter. #" the total number of penalty iterations over all timesteps and by "avg. iter. #" the average number of penalty iterations per timestep required by the penalty method.

Although most basket options are written on arithmetic averages, using geometric averages instead allows us to compute an accurate benchmark solution using a dimension reduction approach. We take the payoff of a geometric average American put option to be $\max(E-g(t),0)$, where $g(t)$ is defined by $g(t) = \left(\prod_{i=1}^{3} s_i(t)\right)^{\frac{1}{3}}$. Using the multi-dimensional Itô's formula, it can be shown [7] that this option is equivalent to an American put option written on one asset with starting value $g(0) = \left(\prod_{i=1}^{3} s_i(0)\right)^{\frac{1}{3}}$, strike E, volatility $\sigma_g = \left(\frac{1}{3^2}\sum_{i,j=1}^{3}\rho_{ij}\sigma_i\sigma_j\right)^{\frac{1}{2}}$ and risk-neutral drift $r_g = r - \left(\frac{1}{3}\sum_{i=1}^{3}\left(d_i+\frac{1}{2}\sigma_i^2\right)-\frac{1}{2}\sigma_g^2\right)$. With the set of parameters used, we have $g(0)=100$, $\sigma_g=0.1633$, and $r_g=0.03-0.0067$. The benchmark solution is 3.00448 obtained using an accurate, adaptive, high-order pricing method developed in [2] for pricing American put options written on one asset.

The reference price of the arithmetic average American put option is 2.94454 [10]. However, since this price may not be very accurate, we estimate the rate of convergence by computing the "change" as the difference in values between a coarser grid and a finer one, and the "ratio" as the ratio of changes between successive grids.

Tables I and II present selected numerical results for the American options on the geometric and arithmetic average of three assets. First, we comment on the computed prices and their convergence. In both cases, the computed prices for the American put option on the CPU and GPU are identical and exhibit second-order convergence. We believe that this favorable behavior is due to the iterative application of the ADI-AF scheme (12); see also Remark 1.

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. It is worth noting that the average number of penalty iterations per timestep required for convergence is about 3-4 iterations, which is slightly higher than that in the case of American options written on one-asset; see, for example, [2], [6], where 1-2 iterations per timestep are required. This may be a mild side-effect of the error in the approximate factorization scheme.

VI. CONCLUSIONS AND FUTURE WORK

This paper discusses a GPU-based parallel algorithm for pricing multi-asset American options via a PDE approach. The algorithm is based on a combination of the penalty approach for handling the LCP and an efficient parallel ADI-AF method on GPUs for the solution of the linear algebraic system arising at each penalty iteration. Numerical results indicate that the proposed GPU-based parallel algorithm is very effective for pricing such derivatives.

At the time of writing this paper, more powerful GPUs with more processors, such as NVIDIA Tesla 20-series based on the "Fermi" architecture, have become available on the market. The increase in the number of parallel processors (448 processors in Tesla C2050) and significant improvements of peak double precision performance (515 GFLOPS in Tesla C2050), as well as the increase in the memory bandwidth (144GB/s in Tesla C2050) should increase the performance of the parallel GPU-based ADI-AF algorithm. In addition, each SM on the new "Fermi" GPUs has 64KB of on-chip memory that can be configured between the shared memory (16KB/48KB) and the

l (τ)	n (s_1)	p (s_2)	q (s_3)	value	error	$\log_2$ ratio	iter. #	avg. #	CPU time (s.)	GPU time (s.)	speed up
20	45	45	45	2.9571	4.7e-2		76	3.8	2.4	0.4	5.9
40	90	90	90	2.9931	1.1e-2	2.0	160	4.0	44.8	3.5	12.8
80	180	180	180	3.0016	2.8e-3	2.0	340	4.3	803.3	46.8	17.1

Table I
SPOT VALUES AND PERFORMANCE RESULTS FOR PRICING AN AMERICAN PUT OPTION ON THE GEOMETRIC AVERAGE OF THREE ASSETS.

l (τ)	n (s_1)	p (s_2)	q (s_3)	value	error	ratio	iter. #	avg. #	CPU time (s.)	GPU time (s.)	speed up
20	45	45	45	2.8885			55	2.8	1.5	0.3	5.1
40	90	90	90	2.9292	4.1e-2		128	3.2	32.0	2.6	12.3
80	180	180	180	2.9403	1.1e-2	3.7	278	3.5	598.3	32.4	18.4

Table II
SPOT VALUES AND PERFORMANCE RESULTS FOR PRICING AN AMERICAN PUT OPTION ON THE ARITHMETIC AVERAGE OF THREE ASSETS.

L1 cache (48KB/16KB). Tripling the amount of shared memory could yield performance improvements for the GPU-based ADI-AF methods. Also, with the availability of L1/L2 cache on this new GPU architecture, the programming is expected to be much simpler.

We conclude the paper by mentioning some possible extensions of this work. It would be desirable to have a theoretical analysis of the second-order convergence of the ADI-AF techniques that we have observed in the experiments. In addition, it would also be interesting to investigate the efficiency of other possible ADI-AF schemes (see Remark 2). To further increase the accuracy and efficiency of the numerical methods, support for non-uniform grids presented in [2] could be incorporated. From a parallelization perspective, extending the current implementation to a multi-GPU platform should increase the performance of the GPU algorithm presented here.

ACKNOWLEDGMENT

This research was supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada. Access to a GPU cluster was provided by the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca).

REFERENCES

[1] L. A. ABBAS-TURKI AND B. LAPEYRE, *American options pricing on multi-core graphic cards*, in Proceedings of International Conference on Business Intelligence and Financial Engineering, IEEE Computer Society, 2009, pp. 307–311.

[2] C. CHRISTARA AND D. M. DANG, *Adaptive and high-order methods for valuing American options*, To appear in the Journal of Computational Finance, (2010), pp. 1–25.

[3] J. CRAIG AND A. SNEYD, *An alternating-direction implicit scheme for parabolic equations with mixed derivatives*, Comp. Math. Appl., 16 (1988), pp. 341–350.

[4] D. M. DANG, *Pricing of cross-currency interest rate derivatives on Graphics Processing Units*, in Proceedings of the 3rd International Workshop on Parallel and Distributed Computing in Finance, IEEE Computer Society, 2010, pp. 1–8.

[5] D. M. DANG, C. CHRISTARA, AND K. JACKSON, *Parallel implementation on GPUs of ADI finite difference methods for parabolic PDEs with applications in finance*, To appear in the Canadian Applied Mathematics Quarterly (CAMQ), (2010), pp. 1–30.

[6] P. A. FORSYTH AND K. VETZAL, *Quadratic convergence for valuing American options using a penalty method*, SIAM J. Sci. Comput., 23 (2002), pp. 2095–2122.

[7] J. C. HULL, *Options, Futures, and Other Derivatives*, Prentice Hall, seventh ed., 2008.

[8] W. HUNDSDORFER AND J. VERWER, *Numerical Solution of Time-Dependent Advection-Diffusion-Reaction Equations, Springer Series in Computational Mathematics*, vol. 33, Springer-Verlag, Berlin, 2003.

[9] K. IN'T HOUT AND B. WELFERT, *Unconditional stability of second-order ADI schemes applied to multi-dimensional diffusion equations with mixed derivative terms*, Appl. Numer. Math, 59 (2009), pp. 677–692.

[10] P. KOVALOV, V. LINETSKY, AND M. MARCOZZI, *Pricing multi-asset American options: A finite element method-of-lines with smooth penalty*, Journal of Scientific Computing, 33 (2007), pp. 209–237.

[11] NVIDIA, *NVIDIA Compute Unified Device Architecture programming guide version 2.3*, NVIDIA Developer Web Site, (2009). http://developer.download.nvidia.com.

[12] D. M. POOLEY, K. R. VERZAL, AND P. A. FORSYTH, *Convergence remedies for non-smooth payoffs in option pricing*, Journal of Computational Finance, 6 (2003), pp. 25–40.

[13] R. RANNACHER, *Finite element solution of diffusion problems with irregular data*, Numerische Mathematik, 43 (1984), pp. 309–327.

[14] D. TAVELLA AND C. RANDALL, *Pricing financial instruments: The finite difference method*, John Wiley & Sons, Chichester, 2000.

[15] A. H. TSE, D. B. THOMAS, AND W. LUK, *Option pricing with multi-dimensional quadrature architectures*, in Proceedings of the 2009 International Conference on Field-Programmable Technology, IEEE Computer Society, 2009, pp. 427 – 430.

[16] P. WILMOTT, J. DEWYNNE, AND S. HOWISON, *Option pricing: Mathematical Models and Computation*, Oxford Financial Press, 1993.

[17] T. P. WITELSKI AND M. BOWEN, *ADI schemes for higher-order nonlinear diffusion equations*, Applied Numerical Mathematics, 45 (2001), pp. 331–351.