Pricing Financial Derivatives with High Performance Finite Difference Solvers on GPUs

Daniel Egloff*

30th April, 2011

The calculation of the fair value and the sensitivity parameters of a financial derivative requires special numerical methods, which are often computationally very demanding. In this chapter we discuss the design and implementation of efficient GPU solvers for the partial differential equations (PDEs) of derivative pricing problems.

For derivatives on a single asset like a stock or an index we consider a massively parallel PDE solver which simultaneously prices a large collection of similar or related derivatives with finite difference schemes. We achieve a speedup of a factor of 25 on a single GPU and up to a factor of 40 on a dual GPU configuration against an optimized CPU version.

Often derivatives are written on multiple underlying assets, e.g. baskets, or the future asset price evolution is modeled with additional risk factors, like for instance stochastic volatilities. The resulting PDE is defined on a multidimensional state space. For these kind of derivatives it is not necessary to pool multiple pricing calculations: alternating direction implicit (ADI) schemes for PDEs on two or more state variables have enough parallelism for an efficient GPU implementation. We benchmark a specific ADI solver for the Heston stochastic volatility model against a fully multi-threaded and optimized CPU implementation. On a recent C2050 Fermi GPU we attain a speedup of a factor of 70 and more for a sufficiently large problem size.

Our results demonstrate the importance of the effective use of GPU resources such as fast on-chip memory and registers.

1 Introduction, Problem Statement, and Context

Under the assumption of no arbitrage, the fair value of a European derivative with maturity T is given by the expectation

$$\mathbb{E}_{\mathbb{Q}}\left[P(0,T)f(X_T)\right], \tag{1}$$

where X_t is usually a vector valued stochastic process modeling the evolution of the market uncertainty, $P(0,T)$ is the price of a zero bond of face value one at maturity, $\mathbb{Q}$ is a suitable risk-neutral probability measure, and $f(X_T)$ is the payoff at maturity. Analytical expressions for

*QuantAlea GmbH, Wasserfuristrasse 42, CH-8542 Wiesendangen, Switzerland

the expectation (1) can only be derived under very stringent conditions; almost all practically relevant models must be solved with numerical techniques.

Existing numerical methods can be broadly classified into three groups: Monte Carlo simulation, partial differential equation (PDE) methods, and numerical integration methods. Monte Carlo simulation is the most direct approach to calculate the expectation (1). It is very versatile and can handle complex and high dimensional models. Monte Carlo methods fall in the category of embarrassingly parallel problems and are usually parallelized on a per sample basis. As such, a GPU implementation is relatively straightforward — typically the most challenging component is the random number generator, which must be capable of producing independent streams of random numbers for every thread. Because GPU programs should be designed in such a way that a very large number of threads can be executed in parallel, the implementation of the random number generator has to carefully balance the quality and the size of the internal state of the random number generator. Impressive performance benefits from using GPUs for Monte Carlo simulations are already documented in several places. An early contribution is made by [1], a more recent work for Asian options is [2]. The principal disadvantages of Monte Carlo simulations are its significant computational cost, even on GPUs, and a possibly large simulation error, which often require special purpose or variance reduction schemes tuned to the specific payoff or product category.

Numerical integration techniques rely on a transformation to the Fourier domain. The idea is to express the expectation (1) as an integral with respect to the probability density function of the random variable X_T. For several important processes, including the Heston stochastic volatility model, the Bates model, and also Levy models, the Fourier transform of the density or the characteristic function is available in analytical form. Hence, a suitable transform of the integral can be calculated analytically and transformed back to obtain the option price. These methods are often used for path independent options and in the context of model calibration. Recently, some progress has been achieved to extend transform techniques also to American and some path-dependent options. An interesting implementation on GPUs can be found in [3].

The Feynman-Kac theorem links the expectation (1) under the risk-neutral measure to the solution of a partial differential equation (see for instance [4], section 4.4 and 5.7, or [5], section 8.2), which then can be solved with numerical methods such as finite difference or finite element schemes. The advantage of the PDE approach is its accuracy and generality. Moreover, handling the optimal stopping problem for American options and early exercise features is significantly less complicated than with Monte Carlo methods. The main drawback is the curse of dimensionality. PDE methods become computationally intractable at high dimensionality, and therefore PDE methods are mainly used for one and two factor models, and to some extent for three factor models.

PDEs in dimension two or higher can be solved with so called operator splitting methods, of which alternating direction implicit schemes are particular versions. The main idea of an ADI scheme is best explained in two dimensions with coordinates x and y under the assumption of no mixed spatial derivative terms. The discretized PDE operator is split into two parts, each representing only the discretization in one coordinate direction. Starting with the initial

conditions, every time step is handled in two stages: the first half-step is taken implicitly in the x-direction and explicitly in the y-direction, followed by a second half-step taken implicit in the y-direction and explicit in the x-direction. The presence of a mixed spatial derivative terms requires further steps to guarantee the stability of the solution.

2 Core Method

The partial differential equation to calculate the arbitrage free price of a derivative, as derived from the Feynman-Kac theorem, is a parabolic convection diffusion equation

$$\frac{\partial u}{\partial t} - \mathcal{L}_t u(t,x) = 0. \tag{2}$$

In one-factor models x is a suitable transform of the underlying asset. An example of a two-factor model is a stochastic volatility model, where $x = (s, v)$, with s the underlying asset and v the unobservable stochastic volatility.

We solve the PDE (2) by finite difference schemes on variable time and state grids. Inhomogeneous grids lead to more complicated expressions for the finite difference discretization of derivatives but are indispensable to achieve good stability and accuracy. For instance, payoff discontinuities and steep gradients lead to spurious oscillations and a significant loss in accuracy. A reasonably effective approach is to increase the finite difference approximation accuracy with locally refined grids in both the time and state dimensions, properly aligned to the points of singularities. For example the mesh must be concentrated at barriers and the time grid must be exponentially refined after payoff singularities or time window barriers. For additional details we refer the reader to [6].

For one-dimensional PDEs and implicit/mixed-time discretization schemes of Crank Nicolson type, a linear system of equations must be solved at every time step. The alternating direction implicit (ADI) schemes of Douglas or Craig-Sneyd [7, 8] and the more recent one of Hundsdorfer-Verwer [9, 10] also lead to many independent linear systems for every direction and at every time step. For the commonly used finite difference approximation of the spatial derivatives, these linear systems are essentially tridiagonal, up to a suitable permutation of the grid points.

The commonly used tridiagonal systems solver is Gaussian elimination without pivoting. It solves diagonally dominant tridiagonal system of n linear equations with $8n$ arithmetic operations. Because the algorithm minimizes the number of arithmetic operations, and because all vector elements are accessed with unit stride, it is optimal for a serial computer. Conversely, the algorithm displays no parallelism at all because all loops are serial recurrences, and therefore it is unsuitable for a parallel architecture with more than one processor.

Solving tridiagonal systems efficiently on parallel computers is very challenging, because of the inherent dependency between the rows of the system and the low computation to communication ratio. Fortunately, the architecture of modern GPUs, with memory close to the ALU and very fast thread synchronization, allows us to implement a fine grained parallel tridiagonal

solver based on the parallel cyclic reduction algorithm and opens the door to very efficient PDE solvers on GPUs.

For a GPU implementation we restrict ourselves to the CUDA parallel computing architecture and refer the reader to [11] for basic CUDA terms and concepts. The adjustments for an OpenCL implementation should be straightforward and are not further discussed.

3 Algorithms, Implementations, and Evaluations

3.1 No-Arbitrage Pricing PDEs

The most commonly used single factor model in the equity domain is the local volatility model with dynamics

$$dS_t = S_{t-}(r_t - q_t)dt + S_{t-}\sigma_{loc}(t, S_{t-})dW_t - \sum_{0<t_i\leq T} \{S_{t_i-}b_i + a_i\}\delta_{t=t_i}\, dt \tag{3}$$

for the underlying asset S_t, where r_t is the risk free rate, q_t a continuous dividend yield, and a_i, b_i discrete/proportional cash dividends at times t_i. In between the time points t_i, the arbitrage free price $V(t, s)$ of a derivative security with payoff $g(s)$ at maturity T satisfies the PDE

$$\frac{\partial V}{\partial t} + (r_t - q_t)s\frac{\partial V}{\partial s} + \frac{1}{2}\sigma^2_{\text{loc}}(t, s)s^2\frac{\partial^2 V}{\partial s^2} - r_t V = 0, \tag{4}$$

with final value $V(T, s) = g(s)$ and has the jump discontinuity

$$V(t_i-, s) = V(t_i+, s(1 - b_i) - a_i) \tag{5}$$

across every discrete dividend time t_i. By a financially motivated coordinate transformation as described in [12] we can remove the jump discontinuities (5) across discrete dividends. The PDE (4) is of the general form (2), with $\mathcal{L}_t$ a second order differential operator

$$\mathcal{L}_t u(t, x) = a(t, x)\frac{\partial^2 u}{\partial x^2}(t, x) + b(t, x)\frac{\partial u}{\partial x}(t, x) + c(t, x)u(t, x) + d(t, x). \tag{6}$$

Single factor convertible bond models, as studied in [13], [14], for example, and one-factor interest rate models, such as the Hull-White, Vasicek, Cox-Ingersoll-Ross, or Black-Derman-Toy model, [15] lead to similar no-arbitrage PDEs.

In (3) the volatility is a state dependent function. It can also be modelled with a separate stochastic process, which results in a two factor model. A prominent example is the Heston stochastic volatility model introduced in [16], with dynamics

$$dS_t = S_t(r_t - q_t)dt + S_t\sqrt{v_t}dW^1_t, \tag{7}$$

$$dv_t = \kappa(\eta - v_t)dt + \sigma\sqrt{v_t}dW^2_t, \tag{8}$$

where $dW_t^1 dW_t^2 = \rho dt$ with correlation $\rho \in [-1, 1]$. The model parameters are the initial volatility $v_0 > 0$, the mean reversion rate κ, long run variance η, volatility of variance σ, and correlation ρ. The corresponding no-arbitrage PDE is given by

$$\frac{\partial V}{\partial t} + \frac{1}{2} v s^2 \frac{\partial^2 V}{\partial s^2} + \rho \sigma v s \frac{\partial^2 V}{\partial s \partial v} + \frac{1}{2} \sigma^2 v \frac{\partial^2 V}{\partial v^2} + (r_t - q_t) s \frac{\partial V}{\partial s} + (\kappa(\eta - v) - \Lambda(t, s, v)) \frac{\partial V}{\partial v} - r_t V = 0 \quad (9)$$

where $\Lambda(t, s, v) = \lambda v$ is the (affine) market price of volatility risk and singles out a specific risk neutral measure. We note that the model structure allows us to re-parameterize it in such a way that $\lambda = 0$.

3.2 Finite Difference Discretization

3.2.1 Single Factor Models

We formulate the finite difference discretization of the PDE (2) with differential operator (6) on variable time and state grids. To this end, let $[0, T] \times [x_{\min}, x_{\max}]$ be a truncated domain and

$$\mathcal{T} = \{t_0, \ldots, t_{n_t - 1}\}, \quad \mathcal{X} = \{x_0, \ldots, x_{n_x - 1}\}, \quad (10)$$

be (possibly inhomogeneous) grids for the time and state variables. Also let

$$\Delta_t^n = t_{n+1} - t_n, \quad \Delta_x^i = x_{i+1} - x_i. \quad (11)$$

For any function $a(t, x)$, $a_i^n = a(t_n, x_i)$ is the value at the grid point (t_n, x_i). Likewise, u_i^n denotes an approximate solution of (2) at (t_n, x_i).

The finite difference approximations on $\mathcal{T} \times \mathcal{X}$ can be conveniently derived using the technique described in [17] and [18]. Fornberg's algorithm allows us to derive the finite difference weights; a version for symbolic calculus is implemented in the Mathematica function `FiniteDifferenceDerivative`. For instance, the second-order-accurate finite difference approximation of the first derivative is given by

$$\frac{\partial u}{\partial x}(t_n, x_i) \approx -\frac{\Delta_x^i}{\Delta_x^{i-1}\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i-1}^n + \left(\frac{1}{\Delta_x^{i-1}} - \frac{1}{\Delta_x^i}\right) u_i^n + \frac{\Delta_x^{i-1}}{\Delta_x^i\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i+1}^n, \quad (12)$$

whereas the first-order-accurate finite difference approximation of the second derivative is

$$\frac{\partial^2 u}{\partial x^2}(t_n, x_i) \approx \frac{2}{\Delta_x^{i-1}\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i-1}^n - \frac{2}{\Delta_x^{i-1}\Delta_x^i} u_i^n + \frac{2}{\Delta_x^i\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i+1}^n. \quad (13)$$

Both approximations are central finite differences, yielding, for all interior nodes $x_1, \ldots, x_{n_x - 2}$, the approximation

$$\mathcal{L}_{t_n} u(t_n, x_i) \approx (\mathcal{L}^n \mathbf{u}^n)_i = a_i^n \sum_{k=-1}^{1} w_k^2 u_{i+k}^n + b_i^n \sum_{k=-1}^{1} w_k^1 u_{i+k}^n + c_i^n u_i^n + d_i^n, \quad (14)$$

with weights w_k^l determined from (12), (13) and $\mathbf{u}_n = (u_0^n, \ldots, u_{n_x-1}^n)^\top$. Equation (14) defines a tridiagonal operator on all interior nodes. To fully specify the operator we must supply boundary conditions. Depending on the payoff, we choose either Dirichlet, Neumann or asymptotically linear boundary conditions

$$s^2 \frac{\partial^2 V}{\partial s^2} = 0 \quad \text{for } s \to \infty. \tag{15}$$

We refer to [19] and also [6] page 122. Note that asymptotically linear boundary conditions are formulated in the underlying coordinates and must be properly transformed if the PDE is solved in any other coordinate system, for example in log-spot coordinates.

Using a one-sided forward difference for the time derivative and the usual mixed scheme for $\theta \in [0, 1]$ gives

$$\frac{u_i^{n+1} - u_i^n}{\Delta_t^n} = (1-\theta)(\mathcal{L}^n \mathbf{u}_n)_i + \theta(\mathcal{L}^{n+1}\mathbf{u}_{n+1})_i, \tag{16}$$

For every for time step $n = n_t - 2, \ldots, 0$ this is a tridiagonal system

$$(\mathrm{id} + (1-\theta)\Delta_t^n \mathcal{L}^n)\, \mathbf{u}_n = \left(\mathrm{id} - \theta \Delta_t^n \mathcal{L}^{n+1}\right) \mathbf{u}_{n+1} \tag{17}$$

for the unknown $\mathbf{u}_n$, in terms of $\mathbf{u}_{n+1}$ determined in the previous time step or from the terminal payoff condition $\mathbf{f}$ at time step $n_t - 1$. The algorithm to find the solution $\mathbf{u}_0$ is now as follows:

Algorithm 1: PDE solver pseudo code

input : discretized operators $\mathcal{L}^n$, terminal condition f, time steps Δ_t^n, parameter θ
result: value curve $\mathbf{u}_0$ at time t_0

$\mathbf{A}_r \leftarrow \mathcal{L}^{N+1}$
$\mathbf{u}_{n_t-1} \leftarrow \mathbf{f}$
for $n \leftarrow n_t - 2$ **to** 0 **do**
  $\mathbf{rhs} \leftarrow (\mathrm{id} - \theta\Delta_t^n \mathbf{A}_r)\, \mathbf{u}_{n+1}$
  $\mathbf{A}_l \leftarrow \mathcal{L}^n$
  $\mathbf{u}_n \leftarrow$ solution of $(\mathrm{id} + (1-\theta)\Delta_t^n \mathbf{A}_l)\, \mathbf{x} = \mathbf{rhs}$
  swap $\mathbf{A}_r \leftrightarrow \mathbf{A}_l$
end

Once the matrices $\mathcal{L}^n$ are assembled, the most performance critical section of Algorithm 1 is to solve the tridiagonal system (17).

3.2.2 Two Factor Models

For a two-factor model the Crank-Nicolson time discretization and finite difference discretization of the differential operator $\mathcal{L}_t$ in (2) leads to a linear system of equations as in (17). However, the matrices $\mathcal{L}^n$ are no longer tridiagonal but instead have a bandwidth proportional

to the minimum of the number of grid points along the coordinate axes of the two factors. The system can be solved with LU factorization, but this becomes ineffective for larger numbers of grid points.

We therefore consider ADI-style splitting schemes. We restrict our exposition to the particular case of the Heston stochastic volatility model, and only briefly explain the ideas and introduce the required notations to discuss the GPU implementation. Further details can be found in [20].

Let $[0,T] \times [s_{\min}, s_{\max}] \times [v_{\min}, v_{\max}]$ be a truncated domain and

$$\mathcal{T} = \{t_0, \ldots, t_{n_t-1}\}, \quad \mathcal{S} = \{s_0, \ldots, s_{n_s-1}\}, \quad \mathcal{V} = \{v_0, \ldots, v_{n_v-1}\}, \tag{18}$$

be suitable grids. Let $\mathbf{v}_n = (V(t_n, s_i, v_j), i = 0, \ldots, n_s - 1, j = 0, \ldots, n_v - 1)^\top$ be an approximation of the solution of (9) on the discretization nodes (18) at time t_n. We approximate the PDE (9) with finite differences, complement it with the proper boundary conditions and decompose the resulting linear operator $\mathcal{L}^n$ into three submatrices

$$\mathcal{L}^n = \mathcal{L}_0^n + \mathcal{L}_1^n + \mathcal{L}_2^n, \tag{19}$$

where $\mathcal{L}_0^n$ is the part of $\mathcal{L}^n$ coming from the mixed derivative term, $\mathcal{L}_1^n$ is the part that contains all the spacial derivatives, and $\mathcal{L}_2^n$ collects all the derivatives in direction of the variance coordinate v. The zero order term $r_t V$ is evenly distributed to $\mathcal{L}_1^n$ and $\mathcal{L}_2^n$. In contrast to [20], we apply finite difference approximations such that the resulting operators $\mathcal{L}_1^n$ and $\mathcal{L}_2^n$ are essentially tridiagonal in a suitable permutation of the grid points.

The Douglas scheme successively calculates the solution at time t_0 by

$$\mathbf{y}_0 = \mathbf{v}_{n+1} + \Delta_t^n \mathcal{L}^{n+1} \mathbf{v}_{n+1} \tag{20}$$

$$\mathbf{y}_j = \mathbf{y}_{j-1} + \Delta_t^n \theta \left(\mathcal{L}_j^n \mathbf{y}_j - \mathcal{L}_j^{n+1} \mathbf{v}_{n+1} \right), \quad j = 1, 2 \tag{21}$$

$$\mathbf{v}_n = \mathbf{y}_2 \tag{22}$$

A forward Euler predictor step is followed by two implicit but unidirectional corrector steps, which stabilize the predictor step. The Douglas scheme is a direct generalization of the classical ADI scheme for two dimensional diffusion equations applied to the situation of a non-vanishing mixed spatial derivative term. It is first-order-accurate, and only stable when applied to two dimensional convection-diffusion equations with a mixed derivative term if $\theta \geq \frac{1}{2}$. A more refined scheme is the Hundsdorfer-Verwer scheme given by

$$\mathbf{y}_0 = \mathbf{v}_{n+1} + \Delta_t^n \mathcal{L}^{n+1} \mathbf{v}_{n+1} \tag{23}$$

$$\mathbf{y}_j = \mathbf{y}_{j-1} + \Delta_t^n \theta \left(\mathcal{L}_j^n \mathbf{y}_j - \mathcal{L}_j^{n+1} \mathbf{v}_{n+1} \right), \quad j = 1, 2 \tag{24}$$

$$\tilde{\mathbf{y}}_0 = \mathbf{y}_0 + \frac{1}{2} \Delta_t^n \left(\mathcal{L}^n \mathbf{y}_2 - \mathcal{L}^{n+1} \mathbf{v}_{n+1} \right) \tag{25}$$

$$\tilde{\mathbf{y}}_j = \tilde{\mathbf{y}}_{j-1} + \Delta_t^n \theta \left(\mathcal{L}_j^n \tilde{\mathbf{y}}_j - \mathcal{L}_j^n \mathbf{y}_2 \right), \quad j = 1, 2 \tag{26}$$

$$\mathbf{v}_n = \tilde{\mathbf{y}}_2 \tag{27}$$

The Hundsdorfer-Verwer scheme is an extension of the Douglas scheme, performing a second predictor step, followed by two unidirectional corrector steps. The advantage of the Hundsdorfer-Verwer scheme is that it attains order of consistency two for general operators $\mathcal{L}_0^n$, $\mathcal{L}_1^n$, $\mathcal{L}_2^n$. Further alternatives are given by the Craig-Sneyd and its modified version, for which we refer to [20].

3.3 Tridiagonal Solver

The implicit time step updates (17) and (21), (24), (26) require linear systems of equations to be solved. However, efficiently solving tridiagonal systems in parallel is a demanding task and requires specialized algorithms. The first parallel algorithm for the solution of tridiagonal systems was cyclic reduction developed by [21], which is also known as the odd-even reduction method. Later [22] introduced the recursive doubling algorithm. Both cyclic reduction and recursive doubling are designed for fine-grained parallelism, where each processor owns exactly one row of the tridiagonal matrix. For further details we refer to section 5.4 in [23] and [24]. For a description of the cyclic reduction, the parallel cyclic reduction and some hybrid schemes on the GPU please refer to A Hybrid Method for Solving Tridiagonal Systems on the GPU, Numerical Algorithms, chapter ???.

A highly optimized GPU based parallel cyclic reduction solver is also introduced in [12]. The implementation slightly differs from the parallel cyclic reduction presented in chapter ???. The main difference is that it can handle systems of any number of dimensions, not necessarily a power of two, uses registers instead of shared memory for temporary variables and uses preprocessor techniques to unroll loops. Finally it uses a problem size dependent kernel dispatching method to call the optimal algorithm for a given dimension. The solver is tuned for systems of dimension 128 to 512 because these dimensions are relevant in practical applications. Its performance is comparable to those presented in [25] and in A Hybrid Method for Solving Tridiagonal Systems on the GPU, Numerical Algorithms, chapter ???.

3.4 GPU Implementation

One Factor PDE Solver Pricing a single derivative in a one-factor model context with Algorithm 1 does not lead to enough parallel work to keep a modern GPU busy. Fortunately, in a realistic financial application it is often the case that many derivative prices must be recalculated at the same time. If an underlying changes, usually a large collection of options with different strikes, maturities, and payoff profiles needs to be repriced. Another example comes from risk management, where whole books of derivatives have to be priced under multiple scenarios.

We therefore design the GPU PDE solver for single factor models based on Algorithm 1 to price a large collection of similar, or related, derivatives in parallel on one or multiple GPUs. The overall collection of pricing problems is split into subsets according to a suitable load balancing strategy. Each subset is scheduled for execution on one of the available GPUs. For every GPU a dedicated CPU thread is responsible for the scheduling and execution of subsets

of pricing problems. This thread also performs the data transfer to and from the GPU and launches the data preparation and pricing kernels.

A single PDE pricing problem out of a subset is solved with a block of threads, where each thread is handling a discretization node of the finite difference scheme. This thread organization optimally utilizes the hierarchical hardware structure of the GPU. It allows us to use shared memory for the data exchange between threads and to synchronize threads working on the same PDE, and we can apply the fine-grained parallel tridiagonal solver from section 3.3 for implicit or Crank Nicolson time stepping schemes. Because thread blocks are executed on one of the available streaming multiprocessors of the GPU we can process several PDE pricing problems in parallel on a single GPU. Note that if we assign more pricing problems to a GPU than numbers of multiprocessors on the GPU, the hardware utilization can be further optimized because the hardware thread manager can switch between different PDE pricing problems, thereby hiding memory latency more efficiently.

Some design and implementation considerations are worth mentioning. A good overall GPU utilization for the one factor PDE solver can only be achieved with a large problem set. Therefore, in order to maximize the number of derivatives which can be bundled for parallel execution, the PDE solver is designed to handle all kinds of payoff features, including single and double barriers, time window barriers, as well as early exercise of Bermudan or American type. On the implementation side, we must pay attention to scalability within a GPU and across multiple GPUs, with sophisticated data management to reuse common data and ensure efficient data transfer.

For a multi-GPU configuration the splitting strategy is important to achieve scalability in the number of GPUs. A fairly simple and still efficient approach is to first group pricing problems of the same underlying and sort them according to the computational cost, measured in terms of the product of time steps and discretization nodes. The idea is to build subsets of pricing problems of roughly the same computational cost and with as much common input data as possible.

Because the transfer of data from CPU host memory to GPU device memory and back can easily cost a significant amount of the overall processing time, an optimized data management is indispensable for high performance and low latency. We design our data management along the following guidelines:

1. Keep the data on the GPU as long as possible and reuse it for multiple computations;
2. Avoid lots of small data transfers, instead, pack data into blocks before sending it to the GPU device memory;
3. Optimize the layout of the packed data by introducing padding, memory alignment, or interleaving, such that the data structure allows for efficient memory access patterns from multiple blocks of threads.

A further optimization is achieved by asynchronous data transfer such that memory transfer and computations can overlap. For additional details we refer to [26].

Two Factor PDE Solver In contrast to single factor models, the ADI schemes for two factor models offer sufficient parallelism for an effective GPU implementation of a single pricing problem; i.e. there is no need to pool multiple pricing problems in order to increase the data parallelism.

We decompose every time step into multiple kernel calls. The Douglas scheme requires three kernel calls per time step. The first kernel performs the explicit forward Euler predictor step (20) by exploiting the decomposition (19). Because the operators $\mathcal{L}_j^n$, $j = 0, 1, 2$ are only used in matrix vector operations, we are free to use a discretization of the spatial derivatives, which does not necessarily lead to tridiagonal operators. This is particularly convenient for the mixed derivative terms in $\mathcal{L}_0^n$ at the boundary, where we can use second order forward and backward difference approximations. This kernel also calculates $\mathcal{L}_j^{n+1}\mathbf{v}_{n+1}$ which will be reused in next two kernel calls to perform (21).

The second kernel calculates $\mathbf{y}_1$ in (21) for $j = 1$ by sweeping over the n_v slices $v = v_0, \ldots, v = v_{n_v-1}$. In the inner nodes the kernel solves a tridiagonal system, and at the face nodes $v = v_0$ and $v = v_{n_v-1}$ it uses the proper boundary conditions. Finally the third kernel determines $\mathbf{y}_2$ from (21) for $j = 2$ by sweeping over the n_s slices $s = s_0, \ldots, s = s_{n_s-1}$. For these two kernels we must pay attention that the operators $\mathcal{L}_j^n$, $j = 1, 2$, which are used to solve the system of equations, are essentially tridiagonal such that we can apply the parallel cyclic reduction solver or a variation thereof.

The implementation of the Hundsorfer-Verwer scheme is similar, but slightly more complex.

4 Final Evaluation

4.1 One Factor PDE Solver

All benchmarks for the one factor PDE solver are compiled with the Microsoft 32-bit C++ compiler and CUDA version 3.1, and executed on an Intel Core 2 Quad CPU Q6602 system, running at 2.4 GHz, with two NVIDIA Tesla C1060 GPUs. The test set to benchmark the efficiency of the one-factor PDE solver consists of a large collection of European put and call options of different maturities and strikes. The finite difference scheme uses 50 time steps per year. This time we measure overall execution time, including the time required to transfer data to and from GPU device memory and the overhead for thread management in case of multiple GPUs. Tables 1 and 2 display the timings based on the fully optimized tridiagonal solver kernel, where all the vectors of the tridiagonal system fit into shared memory.

To gain further insight into the overall efficiency it is interesting to analyze the runtime cost for the data transfer. Because we minimize the number of memory copy operations by packing data into large blocks, we find that only 2% to 4% of the overall execution time is required for the data transfer, including the conversion from double to single precision.

So far we only considered the pricing of European options. The benchmark results for barrier options are very much similar. We handle the pricing of American options with a parallel

Problem size	CPU	1 GPU	Speedup 1 GPU	2 GPUs	Speedup 2 GPUs	GPU Scaling
300	582	33	17.6	28	20.8	1.2
600	1149	53	21.7	45	25.5	1.2
900	1707	76	22.5	61	28.0	1.2
1200	2277	94	24.2	72	31.6	1.3
1800	3437	141	24.4	101	34.0	1.4

Table 1: European option, state grid size 128. Timings are measured in milliseconds.

Problem size	CPU	1 GPU	Speedup 1 GPU	2 GPUs	Speedup 2 GPUs	GPU Scaling
300	1091	52	21.0	41	26.6	1.3
600	2204	93	23.7	72	30.6	1.3
900	3254	132	24.9	91	35.8	1.4
1200	4405	174	25.1	120	36.7	1.5

Table 2: European option, state grid size 256. Timings are measured in milliseconds.

operator splitting method, which resolves the nonlinear early exercise constraint independently for every discretization state and can therefore be implemented in a data-parallel manner. The resulting performance figures are even better than for European options.

4.2 PDE Solver for Heston Stochastic Volatility Model

The GPU ADI solver for the two dimensional Heston stochastic volatility model is benchmarked against an optimized fully multi-threaded CPU implementation, which we based on the Intel thread building blocks. In the following we only consider the Douglas scheme in single precision, see Figure 1 for a graphical illustration.

On an Intel dual core E5200 2.5 GHz with a GTX260 GPU the ADI solver runs about 40 times faster than the CPU single core version and 27 times faster than the optimized multi-core version. The shared memory requirements of the tridiagonal solver limit the GPU ADI solver on a Tesla C1060 or GTX260 to state grids of at most 1004 points. The best speedup is achieved when the state grid size is near this limit. For small scale problems the speedup is not very large, due to the cost of allocating device memory. If the problem size is growing, the time required to allocate memory on the GPU becomes less dominant and the speedup increases significantly.

Interesting performance results are obtained with a recent C2050 Fermi GPU. The ADI solver on a C2050 faces fewer hardware limitations. More registers and a larger amount of shared memory per block allow us to execute problems with up to 3056 state grid nodes. At these very large grid points we obtain a speedup factor of more than 70 against a single core implementation. Besides the capability to handle larger problem sizes, the speed increase of the

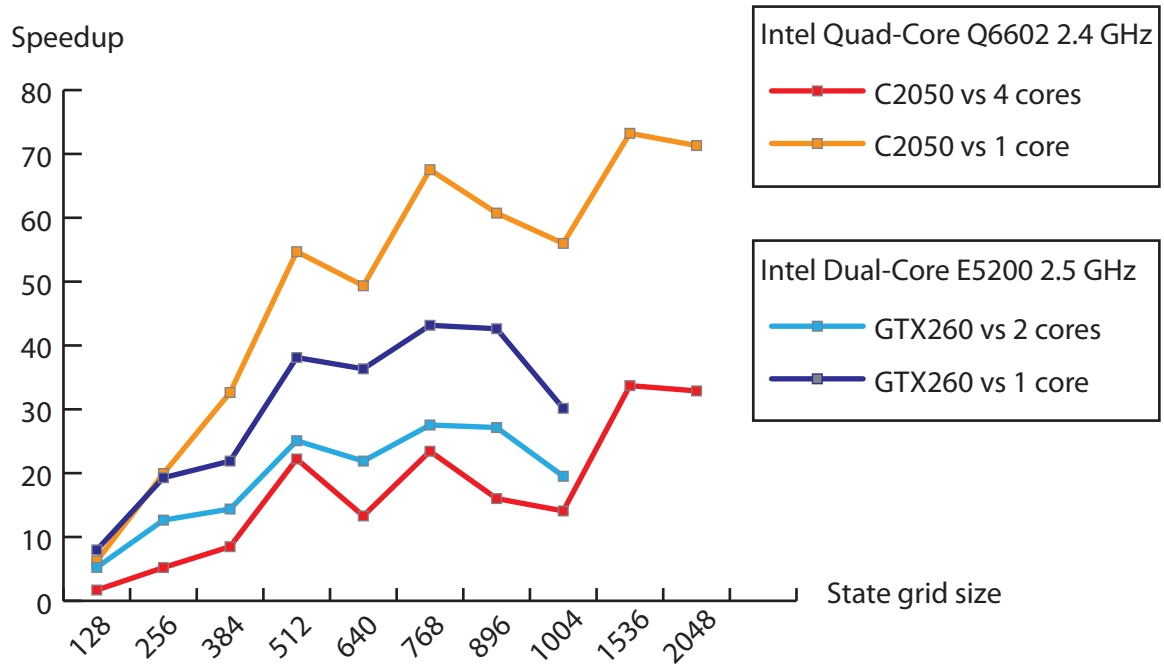

Figure 1: Speedup ADI Douglas scheme on GTX260 versus dual core and C2050 versus quad core.

C2050 against the GTX260 ranges from a factor of 1.6 up to a factor 2 for the state grid size of 896 and more, even though the C2050 has only 14 multiprocessors and no Fermi-specific optimizations were made in the code.

4.3 Single Precision versus Double Precision

The current GPU generation features substantially more throughput in single precision than in double precision. Our numerical experiments show that for practical applications, the accuracy of single precision is usually sufficient. The stability and accuracy of the solver benefits from a careful implementation:

1. The proper parallel algorithm for solving tridiagonal systems must be selected. Our implementation of the parallel cyclic reduction does not provide pivoting, hence it can exhibit numerical instabilities for general tridiagonal systems. However, it is stable for diagonally dominant matrices, which occur from finite difference discretization of diffusion-dominated PDEs. [25] found that the recursive doubling algorithm does not achieve a good accuracy and may even suffer from overflow.

2. The time grid and discretization nodes of the finite difference scheme must have proper concentration and alignment of grid points to avoid the propagation of oscillation effects in the solution.

3. Accumulation of discrete dividends in small time intervals leads to many jump discontinuities, which adversely affect the accuracy of the result. In such a case, the PDE is

Problem size (n_s, n_v)	One core	Four cores	GPU	Speedup single core	Speedup multi-core
128	2.1	0.6	0.3	6.2	1.7
256	8.5	2.2	0.4	19.9	5.2
384	19.4	5.0	0.6	32.6	8.5
512	46.8	19.0	0.9	54.7	22.2
640	55.9	15.1	1.1	49.3	13.3
768	97.9	34.0	1.4	67.5	23.4
896	111.0	29.3	1.8	60.7	16.0
1004	133.6	33.6	2.4	56.0	14.1
1536	445.8	205.2	6.1	73.2	33.7
2048	793.3	365.7	11.1	71.3	32.9

Table 3: Timing in seconds. Time to maturity = 2 years, time steps = 200 (Intel Quad-Core Q6602 2.4GHz, Fermi C2050, Cuda 3.1, Windows Vista).

better solved in suitably transformed coordinates, which remove the jump discontinuities completely.

4. Use higher order finite difference schemes to calculate Greeks and sensitivities on the grid of discretization nodes.

4.4 Conclusion

The architecture of the GPU and the hardware limitations make the development of high performance general purpose solvers for one dimensional finite difference schemes a challenging task. Our benchmark results prove that, with the proper algorithm and a well designed GPU resource management, it is possible to implement solvers which perform exceptionally well in the range of numbers of discretization nodes from 128 to 512. Fortunately, most realistic problems can be handled within that range. For single factor models, the performance figures clearly document the importance of a large problem set of around 300 or more pricing problems. On the other hand, ADI schemes for two factor models are well suited for an efficient GPU implementation, even for a single pricing problem. On the new Fermi hardware architecture our ADI solver can run significantly larger problems and shows a large performance speedup of up to a factor of 70 against an optimized single core CPU implementation, even though we did not specifically tune the solver for the Fermi architecture.

5 Future Directions

Yet another interesting application for GPU application is the calibration of financial models.

Let us consider the local volatility model. The calibration algorithm determines the state dependent local volatility surface $(t, s) \mapsto \sigma_{loc}(t, s)$ in (3) from quoted option prices or implied volatilities. Dupire's formula (see, for example, section 2 in [27]), expresses the local volatility

as an expression of the first and second derivatives of the implied volatility surface, which must be calculated on a relatively fine grid of time and state values, either through numerical differentiation or by exploiting the analytical representation of the implied volatility surface interpolation. The calculations are fully data parallel because the grid point can be processed independently. An alternative approach is the PDE based local volatility calibration method of [27], which requires a series of tridiagonal systems to be solved for every time step. The tridiagonal systems are decoupled over time and can be solved independently.

An interesting approach is to combine a local volatility calibration algorithm with the finite difference solver for pricing on the GPU. The first advantage is that the input data is reduced significantly because instead of transferring a possibly large local volatility matrix, only a few implied volatility slices have to be copied to the GPU. Caching the local volatility surface directly on the GPU and reusing it for multiple calculation further enhances the overall performance. The advantage is even more pronounced if the Greeks are calculated by taking volatility smile dynamics into account, which would require a recalibration of the local volatilities after shifting the spot value.

A further application comes from spline interpolation and spline smoothing, which is often applied to preprocess implied volatility smiles before passing them to local volatility calibration.

Last but not least it would be interesting to explore three factor models with ADI methods and the tridiagonal parallel cyclic reduction solver. In [28] a GPU implementation is discussed but they do not use a fine-grained parallel solver for the resulting tridiagonal systems. Instead they use a much simpler approach where individual linear systems are solved in a single thread.

References

[1] C. Bennemann, M. W. Beinker, D. Egloff, and M. Gauckler. Teraflops for games and derivatives pricing. *Wilmott Magazine*, 36:50–54, 2008.

[2] M. Joshi. Graphical Asians. Technical report, University of Melbourne - Centre for Actuarial Stud, 2009.

[3] C. W. Oosterlee and B. Zhang. Acceleration of option pricing technique on graphics processing units. Technical report, Delft University of Technology, Report 10-3, 2010.

[4] I. Karatzas and S. E. Shreve. *Brownian Motion and Stochastic Calculus*, volume 113 of *Graduate Texts in Math.* Springer, second ed. edition, 1999.

[5] Bernt Oksendal. *Stochastic Differential Equations, An Introduction with Applications*. Springer, fifth edition edition, 2000.

[6] D. Tavella and C. Randall. *Pricing finanical instruments – The finite difference method*. Wiley, 2000.

[7] J. Douglas. Alternating direction methods for three space variables. *Numer. Math.*, 4:41–63, 1962.

[8] I. J. D. Craig and A. D. Sneyd. An alternating-direction implicit scheme for parabolic equations with mixed derivative terms. *Comp. Math. Appl.*, 16:341–350, 1988.

[9] K. J. in 't Hout and B. D. Welfert. Stability of ADI schemes applied to convection diffusion equations with mixed derivative terms. *Appl. Numer. Math.*, 57:19–35, 2007.

[10] K. J. in 't Hout and B. D. Welfert. Unconditional stability of second order ADI schemes applied to multi-dimensional diffusion equations with mixed derivative terms. *Appl. Numer. Math.*, 59:677–692, 2009.

[11] NVIDIA. CUDA^{TM} Compute Unified Device Architecture Programming Guide. Technical Report 3.1, NVIDIA Corporation `http://www.nvidia.com/object/cuda_develop.html`, 2010.

[12] D. Egloff. GPUs in financial computing: High performance tridiagonal solvers on GPUs for partial differential equations. *Wilmott Magazine*, September, 2010.

[13] E. Ayache, P. A. Forsyth, and K. R. Vetzal. Next generation models for convertible bonds with credit risk. *Wilmott Magazine*, 11:68–77, 2002.

[14] L. Andersen and D. Buffum. Calibration and implementation of convertible bond models. *Journal of Computational Finance*, 7(2):1–34, 2004.

[15] D. Brigo and F. Mercurio. *Interest Rate Models - Theory and Practice*. Springer-Verlag, 2006.

[16] S. L. Heston. A closed-form solution for options with stochastic volatility with application to bond and currency options. *The Review of Financial Studies*, 9(2):327–343, 1993.

[17] B. Fornberg. Fast generation of weights in finite difference formulas. In G. D. Byrne and W. E. Schiesser, editors, *Recent Developments in Numerical Methods and Software for ODEs/DAEs/PDEs*, pages 97–123. World Scientific, Singapore, 1992.

[18] B. Fornberg. Calculation of weights in finite difference formulas. *SIAM Review*, 40(3):685–691, 1998.

[19] H. Windcliff, P. A. Forsyth, and K. R. Vetzal. Analysis of the stability of the linear boundary condition for the Black-Scholes equation. *Computational Finance*, 8(1), August 2004.

[20] K. J. in 't Hout and S. Foulon. ADI finite difference schemes for option pricing in the Heston model with correlation. *Int. J. Numer. Anal. Mod.*, 7(2):302–320, 2010.

[21] R. W. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM*, 12(1):95–113, 1965.

[22] H. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20(1):27–38, 1973.

[23] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: architecture, programming,*

and algorithms. Institute of Physics Publishing, second edition, 1988.

[24] W. Gander and G. H. Golub. Cyclic reduction - history and applications. In Gene Howard Golub, editor, *Proceedings of the Workshop on Scientific Computing*. Springer Verlag, 1997.

[25] Yao Zhang, Jonathan Cohen, and John D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*, page 10, January 2010.

[26] D. Egloff. GPUs in financial computing: Massively parallel PDE solvers on gpus. *Wilmott Magazine*, November, 2010.

[27] L. B. G. Andersen and R. Brotherton-Ratcliffe. The equity option volatility smile: An implicit finite-difference approach. *Journal of Computational Finance*, 1(2):5–37, 1997.

[28] D. M. Dang, C. C. Christara, and K. R. Jackson. Parallel implementation on GPUs of ADI finite difference methods for parabolic PDEs with applications in finance. Technical report, Department of Computer Science, University of Toronto, 2010.