Daniel Egloff

Part I: High-Performance Tridiagonal Solvers on GPUs

In a series of notes we would like to bring massively parallel GPU computing to a broader audience in the financial quant world. The financial industry is starting to adopt GPUs more and more. The main application fields as of today are Monte Carlo simulations. They fall into the category of embarrassingly parallel problems and can be implemented on a GPU in a relatively straightforward manner, once a good random number generator is available. Speedups of a factor of 50 to 100 are within reach.

Much less work has been done to speed up PDE solvers with GPUs. Developing efficient PDE solvers on GPUs requires a more detailed understanding of GPUs, their hardware architecture, the different memory types, and the thread execution model. Hence, going through the exercise of writing a PDE solver on a GPU is a very good education in GPU computing.

We therefore dedicate our first series of articles to PDE solvers on GPUs. In the first part we give a quick introduction to one-dimensional finite difference schemes with a view towards GPUs. We then show how to implement a highly efficient parallel tridiagonal solver on a modern GPU based on the parallel cyclic reduction algorithm. This algorithm will be the backbone solver in the second part, where we discuss the final one-dimensional PDE solver and give performance benchmarks.

Introduction

Graphics processing units (GPUs) are slowly being adopted by the financial services industry and are mainly applied to speed up Monte Carlo simulations. Implementing Monte Carlo simulations on GPUs is relatively straightforward. They fall into the category of embarrassingly parallel problems and can be parallelized on a per sample basis. 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 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 in [3], a more recent work for Asian options is [11].

Designing an efficient GPU solver for PDEs based on finite difference schemes or finite elements is significantly harder. Every time step consists of constructing and solving a linear system of equations. The dependence in the calculation of solutions on linear systems on the one hand and the hierarchical architecture of modern GPUs with several hardware limitations on the other hand, makes the development of general purpose linear system solvers very challenging.

In this first part we concentrate on solving tridiagonal systems very efficiently based on a fine-grained parallel algorithm. In a series of follow-up articles we show how to apply it to simultaneously price a large collection of similar or related derivatives, in parallel local volatility calibration, and in higher dimensional PDEs with the method of alternating directions.

Derivatives Pricing based on Partial Differential Equations

We first introduce some background on partial differential equations and finite difference schemes, which is useful to understand the numerical implementation of GPUs.

No-Arbitrage Pricing PDE

It is well known that under suitable assumptions, the Feynman–Kac theorem links the arbitrage-free price of a derivative, expressed as a conditional expectation under the risk-neutral probability measure, to a parabolic convection diffusion equation

$$\frac{\partial u}{\partial t} = L_t u(t,x), \tag{1}$$

where, in case of a single risk factor, L_t is the differential operator

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

The partial differential equation (1) covers several important single-factor financial models in different asset classes. In the equity domain, it includes the local volatility model

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

with risk-free rate r_t, continuous dividend yield q_t, discrete cash and proportional dividends a_i, b_i at times t_i. Between the dividend dates t_i, the arbitrage-free price $V_{(t,s)}$ of a derivative security with payoff $g(s)$ at maturity T satisfies the PDE

WILM_Egloff_v2_2010.indd 32 8/26/10 7:32:51 PM

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

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 date t_i.

Other important cases are given by single-factor convertible bond models, as for instance studied in [1,2], and one-factor interest rate models, such as the Hull–White, Vasiçek, Cox–Ingersoll–Ross, or Black–Derman–Toy model [4]. From now on we will concentrate on the local volatility model because it covers the general case with state-dependent coefficients and jump singularities across discrete dividend dates.

Coordinate Transforms

The PDE (4) directly represents the risk-neutral dynamics (3). To get a strongly parabolic PDE one can transform the equation to log-spot coordinates. From an implementation point of view both coordinate representations have the drawback that for every dividend date a jump condition has to be implemented, which requires an interpolation of the value function on the finite difference grid; this slows the computation down and reduces the convergence accuracy. To avoid this we apply the transformation introduced in [14] and further elaborated in [5]. More precisely, let

$$F_t = E_Q[S_t] = R_t\left(S_0 - \sum_{j:0<t_j\le t}\frac{a_j}{R_{t_j}}\right) \tag{6}$$

be the forward price of S_t, with

$$R_t = e^{\int_0^t (r_s - q_s)ds}\prod_{j:0<t_j\le t}(1-b_j) \tag{7}$$

the proportional growth factor of S_t. It is shown in [5] that the affine transformed process

$$X_t = \frac{S_t - D_t}{F_t - D_t}, \tag{8}$$

is a non-negative (local) martingale, where

$$D_t = R_t\sum_{j:t_j>t}\frac{a_j}{R_{t_j}} \tag{9}$$

is the so-called dividend floor, defined as the lowest value below which the asset price S_t cannot fall because of deterministic cash payment in the future. Its risk-neutral dynamics is given by

$$\frac{dX_t}{X_t} = \zeta_{\text{loc}}(t, X_t)dW_t, \quad X_0 = 1, \tag{10}$$

where the local volatility $\zeta_{\text{loc}}(t,x)$ of the pure price process X_t is related to the local volatility $\sigma_{\text{loc}}(t,s)$ of the original dividend-paying asset process S_t by the relation

$$\sigma_{\text{loc}}(t,s) = \begin{cases} \frac{s - D_{t-}}{s}\zeta_{\text{loc}}\left(t, \frac{s - D_{t-}}{F_{t-} - D_{t-}}\right) & s > D_{t-}, \\ 0 & s \le D_{t-}. \end{cases} \tag{11}$$

Because X_t has no jumps caused by discrete dividends, we may assume that the local volatility surface $\zeta_{\text{loc}}(t,x)$ is smooth as a bivariate function of t and x. We can exploit the smoothness of $\zeta_{\text{loc}}(t,x)$ to calibrate the local volatility surface against observed quotes of vanilla option prices. More importantly, if we apply the time-dependent coordinate transform

$$x = \frac{s - D_t}{F_t - D_t}, \tag{12}$$

we will simplify the PDE (4) as follows: let

$$G(t,x) = V(t,s) = V(t, (F_t - D_t)x + D_t). \tag{13}$$

Then, a simple calculation shows that the function $G(t, x)$ satisfies the PDE

$$\frac{\partial G}{\partial t} - \frac{1}{2}x^2\zeta_{\text{loc}}^2(t,x)\frac{\partial^2 G}{\partial x^2} = 0, \tag{14}$$

with terminal condition $G(T, x) = g((F_T - D_T)x + D_T)$. For a GPU implementation, the representation (14) has a significant additional advantage: only one state-dependent coefficient must be looked up and possibly interpolated, in case the calibration grid does not match the calculation grid. Because such coefficients are usually stored in GPU device memory, we can save the high cost of a global memory-reading operation, which may exhibit a latency up to between 450 and 600 cycles.

Discretization by Finite Differences

To maximize GPU performance we are going to implement our solver also in single precision. At the scale of single precision, stability and accuracy are even more important. For instance, payoff discontinuities and steep gradients lead to spurious oscillations and a significant loss in accuracy. To deal with these problems, 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. A suitable grid concentration and alignment policy can usually be derived ex-ante from the option payoff function. 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 and different grid generation techniques, we refer to [16].

WILM_Egloff_v2_2010.indd 33 8/26/10 7:32:55 PM

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

$$\mathbf{T} = \{t_0, \ldots, t_{N+1}\}, \quad \mathbf{X} = \{x_0, \ldots, x_{I+1}\}, \tag{15}$$

possibly inhomogeneous grids for the time respectively state variable and

$$\Delta_t^n = t_{n+1} - t_n, \quad \Delta_x^i = x_{i+1} - x_i. \tag{16}$$

For any function $a(t, x)$ let $a_i^n = a(t_n, x_i)$ be the value at the grid point (t_n, x_i). Likewise, let u_i^n denote an approximate solution of (1) at (t_n, x_i).

The finite difference approximations on $\mathbf{T} \times \mathbf{X}$ can be conveniently derived by using Fornberg's algorithm introduced in [6] and [7]. Fornberg's algorithm allows us to derive the finite difference weights. A version for symbolic calculus is implemented in the Mathematica function "FiniteDiff erenceDerivative". For instance, the second-order accurate finite difference approximation of the first derivative is given by

$$\frac{\partial u}{\partial x}(t_n, x_i) \approx -\frac{\Delta_x^i}{\Delta_x^{i-1}\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i-1}^n + \left(\frac{1}{\Delta_x^{i-1}} - \frac{1}{\Delta_x^i}\right) u_i^n + \frac{\Delta_x^{i-1}}{\Delta_x^i\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i+1}^n, \tag{17}$$

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

$$\frac{\partial^2 u}{\partial x^2}(t_n, x_i) \approx \frac{2}{\Delta_x^{i-1}\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i-1}^n - \frac{2}{\Delta_x^{i-1}\Delta_x^i} u_i^n + \frac{2}{\Delta_x^i\left(\Delta_x^{i-1} + \Delta_x^i\right)} u_{i+1}^n. \tag{18}$$

Both approximations are centered finite differences. This yields for all interior nodes $x_1, \ldots, x_I$ the approximation

$$L_{t_n} u(t_n, x_i) \approx (L^n \mathbf{u}^n)_i = a_i^n \sum_{k=-1}^{1} w_k^2 u_{i+k}^n + b_i^n \sum_{k=-1}^{1} w_k^1 u_{i+k}^n + c_i^n u_i^n + d_i^n, \tag{19}$$

with weights w_k^l determined from (17), (18) and $\mathbf{u}^n = (u_0^n, \ldots, u_{I+1}^n)^T$. Equation (19) defines a tridiagonal operator on all interior nodes. To complete the system we must supply boundary conditions. Depending on the payoff we choose either Dirichlet, Neumann or asymptotically linear boundary conditions

$$s^2 \frac{\partial^2 V}{\partial s^2} = 0 \quad \text{for } s \to \infty. \tag{20}$$

We refer to [17] and also [16, p. 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, such as for example in log-spot or in the coordinates of (12).

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

$$\frac{u_i^{n+1} - u_i^n}{\Delta_t^n} = (1-\theta)(L^n \mathbf{u}^n)_i + \theta(L^{n+1}\mathbf{u}^{n+1})_i. \tag{21}$$

If we solve for $\mathbf{u}^n$, we obtain a tridiagonal system

$$\left(\mathrm{id} + (1-\theta)\Delta_t^n L^n\right)\mathbf{u}^n = \left(\mathrm{id} - \theta\Delta_t^n L^{n+1}\right)\mathbf{u}^{n+1}, \tag{22}$$

for every time step $n = N, \ldots, 0$. For a terminal payoff $\mathbf{f}$ the algorithm to find the solution $\mathbf{u}^0$ takes the form of the following PDE solver pseudo code, henceforth referred to as Algorithm 1.

```
assemble matrix A_r ← L^{N+1}
set u^{N+1} ← f
for n ← N to 0 do
    calculate right-hand side v ← (id − θΔ_t^n A_r) u^{n+1}
    assemble matrix A_l ← L^n
    u^n ← solution of (id + (1 − θ)Δ_t^n A_l) u^n = v
    swap A_r ← A_l
end
```

Once the matrices L^n are assembled, the most performance-critical section of Algorithm 1 is to solve the tridiagonal system (22). On a CPU, this can be achieved very efficiently with Gaussian elimination, as implemented for instance in the Lapack functions sgtsv and dgtsv, as provided in [12]. Partial pivoting may be applied to increase the accuracy of ill-conditioned systems, but in most practical situations this is not necessary, except possibly for very low volatility regimes.

A Gaussian elimination algorithm without pivoting solves a diagonally dominant tridiagonal system of n linear equations with $8n$ arithmetic operations. Because the algorithm minimizes the number of arithmetic operations and all vector elements are accessed with unit stride, it is optimal for a serial computer. On the other hand, it displays no parallelism at all because all loops are serial recurrences. It is highly unsuitable for a parallel architecture with more than one processor. Therefore, the crucial step to move from CPU to GPU is a tridiagonal solver, which is capable of exploiting the parallel hardware architecture of a GPU.

A Highly Efficient Tridiagonal GPU Solver

Solving tridiagonal systems efficiently on parallel computers is challenging, because of the inherent dependency between the rows of the system and the low computation-to-communication ratio. The first parallel algorithm for the solution of tridiagonal systems was cyclic reduction developed by Hockney [9], which is also known as the odd–even reduction method. Later, Stone introduced in [15] 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 [10] and [8].

WILM_Egloff_v2_2010.indd 34 8/26/10 7:32:56 PM

Let us first look at the cyclic reduction algorithm in more detail and consider the tridiagonal system

$$Ax = \begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & 0 & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & \ddots & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ \vdots \\ h_{n-1} \\ h_n \end{pmatrix} = h \tag{23}$$

The idea of cyclic reduction is to eliminate variables from adjacent equations and reduce the system recursively until a single equation or a two-by-two system remains. Consider equation *i* with its upper and lower equations

$$\begin{array}{ccccccccc} a_{i-1}x_{i-2} & + & b_{i-1}x_{i-1} & + & c_{i-1}x_i & & & = & h_{i-1} \\ & & a_i x_{i-1} & + & b_i x_i & + & c_i x_{i+1} & = & h_i \\ & & & & a_{i+1}x_i & + & b_{i+1}x_{i+1} + c_{i+1}x_{i+2} & = & h_{i+1} \end{array} . \tag{24}$$

To eliminate x_{i-1} and x_{i+1} we multiply the first equation by $\alpha_i = -a_i / b_{i-1}$ and the last one by $\gamma_i = -c_i / b_{i+1}$ and sum up the three equations to get

$$a_i^{(1)} x_{i-2} + b_i^{(1)} x_i + c_i^{(1)} x_{i+2} = h_i^{(1)}, \tag{25}$$

where the coefficients are given by

$$\begin{aligned} a_i^{(1)} &= \alpha_i a_{i-1}, \\ b_i^{(1)} &= b_i + \alpha_i c_{i-1} + \gamma_i a_{i+1}, \\ c_i^{(1)} &= \gamma_i c_{i+1}, \\ h_i^{(1)} &= h_i + \alpha_i h_{i-1} + \gamma_i h_{i+1}. \end{aligned} \tag{26}$$

It is important to note that equation (25) only refers to every second variable.

Cyclic reduction solves the tridiagonal system (23) in two phases. For explanation purposes we assume that $n = 2^q - 1$ with $q \in \mathbb{N}$. The forward reduction phase starts from the original coefficients and the right-hand side and recursively calculates new coefficients and right-hand sides for reduction levels $l = 1, \ldots, q-1$ by

$$\begin{aligned} a_i^{(l)} &= \alpha_i a_{i-2^{l-1}}^{(l-1)}, \\ b_i^{(l)} &= b_i^{(l-1)} + \alpha_i c_{i-2^{l-1}}^{(l-1)} + \gamma_i a_{i+2^{l-1}}^{(l-1)}, \\ c_i^{(l)} &= \gamma_i c_{i+2^{l-1}}^{(l-1)}, \\ h_i^{(l)} &= h_i^{(l-1)} + \alpha_i h_{i-2^{l-1}}^{(l-1)} + \gamma_i h_{i+2^{l-1}}^{(l-1)}, \end{aligned} \tag{27}$$

with

$$\alpha_i = -a_i / b_{i-2^{l-1}}, \quad \gamma_i = -c_i / b_{i+2^{l-1}}, \tag{28}$$

and index *i* running through

$$i \in \{k2^l \mid k = 1, \ldots, 2^{q-2l}\}. \tag{29}$$

Each reduction roughly halves the number of equations. At level q–1 only one equation remains and we can start the second phase, which performs a backward substitution to find the solution recursively for the levels $l = q, q-1, \ldots, 1$

$$x_i = \frac{1}{b_i^{(l-1)}} \left(h_i^{(l-1)} - a_i^{(l-1)} x_{i-2^{l-1}} - c_i^{(l-1)} x_{i+2^{l-1}} \right), \tag{30}$$

where *i* runs through

$$i \in \{2^{l-1} + k2^l \mid k = 1, \ldots, 2^{q-l} - 1\}, \tag{31}$$

and $x_0 = 0, x_{2^q} = 0$ whenever used. If we run cyclic reduction on an array of *n* processors we see from (27) that 2(q–1) nearest-neighbor communications are required. Additional storage requirements can be held to a minimum by overwriting the original tridiagonal system with all reduced systems of equations. Note however, because equations (27) are solved in parallel, we must use temporary storage to make sure that we always use the right values from the previous reduction level.

The optimal algorithm on a serial computer must minimize the number of arithmetic operations. On a parallel architecture multiple aspects have to be considered and properly balanced. The optimal parallel algorithm depends on the amount of hardware parallelism of the computer. We can trade computation cost against parallelism to enhance the overall efficiency by better using the hardware infrastructure. If we look at the above cyclic reduction algorithm, the amount of parallelism deteriorates in the forward reduction and vice versa increases in the backward substitution, with every level *l*. The parallel cyclic reduction algorithm is a slight variation, which applies the reduction (27) simultaneously to all *n* equations. The resulting algorithm requires more computations but shows significantly higher parallelism. Another advantage of the parallel cyclic reduction algorithm is a better memory access pattern for GPU devices. The basic cyclic reduction algorithm has a strided access pattern, the stride being doubled in every forward reduction, respectively halved in a backward reduction step. This leads to bank conflicts very soon. The same observation has also been made independently by Zhang *et al.* in [18]. They also developed a combination of cyclic reduction and the parallel cyclic reduction algorithm, which they call the hybrid algorithm, with a further performance benefit of roughly 20% over the pure parallel cyclic reduction algorithm.

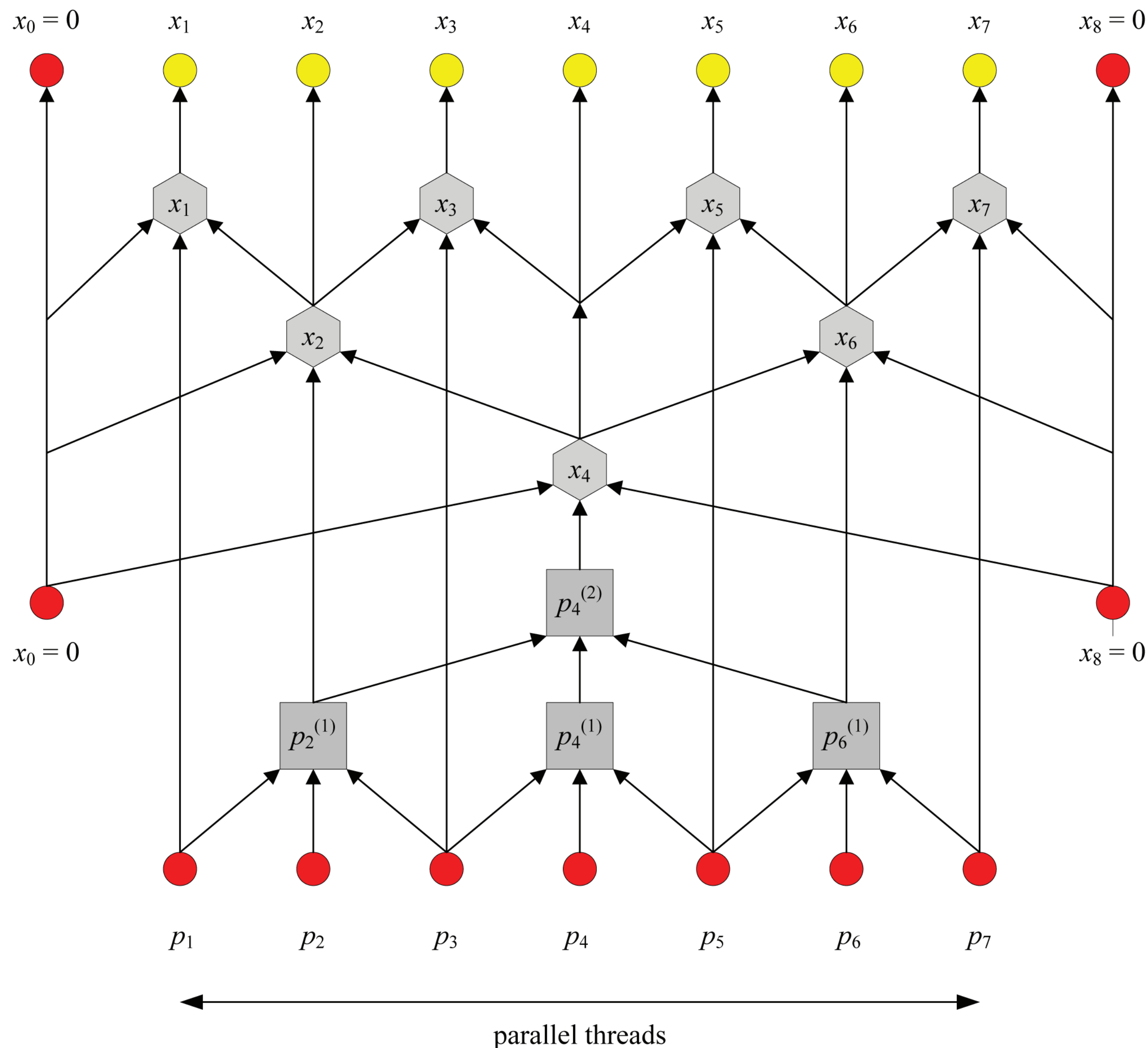
Concrete GPU Implementation

Modern GPUs provide special hardware features such as close to the ALU shared memory, which allow us to efficiently implement parallel tridiagonal solvers, based on cyclic reduction or recursive doubling. For a concrete GPU implementation we target the parallel computing architecture CUDA of NVIDIA, see [13]. The basic units of executable code in CUDA are so-called kernels. An application may consist of several kernels, written in a specific dialect of the C language, enriched with additional keywords to express parallelism. Whenever a kernel is executed by the host (CPU), an index space is

>

WILM_Egloff_v2_2010.indd 35 8/26/10 7:32:57 PM

Figure 1: Routing diagram of the cyclic reduction algorithm. We denote by $p_i=(a_i, b_i, c_i, h_i)$ the coefficients of a single row i in (23) and let $p_0 = (0, 1, 0, 0)$ be the values to handle the boundary condition. Similarly, $p_i^{(l)} = (a_i^{(l)}, b_i^{(l)}, c_i^{(l)}, h_i^{(l)})$ represents the updated coefficients of row i at reduction level l. Squares represent the evaluation of equations (27) in the forward reduction phase, hexagons correspond to the evaluation of (30) during backward substitution.

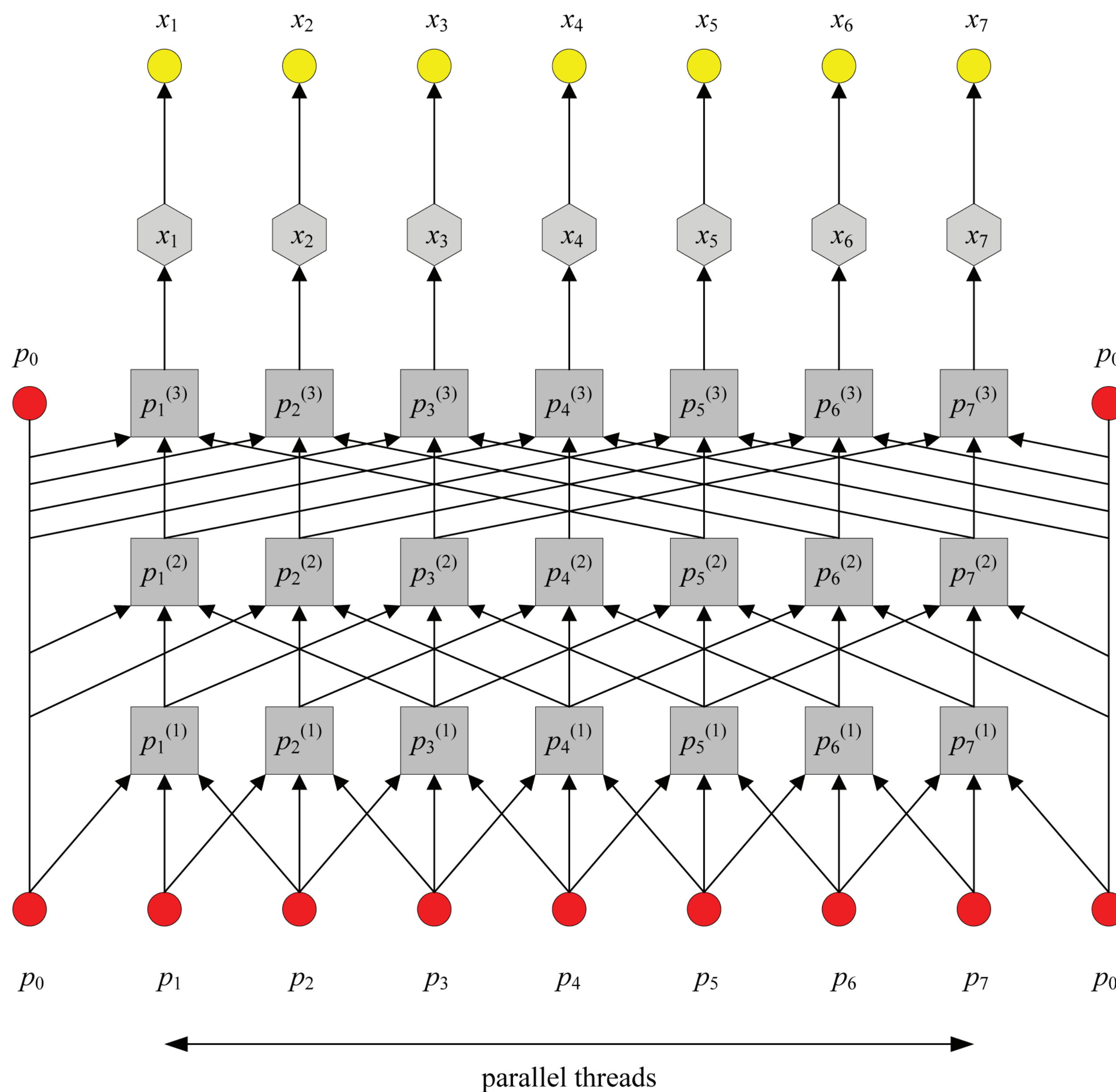


defined and a separate instance of the kernel executes for every point in this index space. Each kernel instance is called a thread and is uniquely identified by its point in the index space. A thread executes the same code but the specific execution path through the code, and the data operated upon, can vary from thread to thread.

Threads are organized into thread blocks for synchronization and communication purposes, providing a more coarse-grained decomposition of the index space. The threads in a given thread block execute concurrently on the processing elements of a single compute unit. The scheduling of threads depends on the actual implementation. Detailed knowledge is necessary to optimally utilize the device. For instance, CUDA-enabled GPUs use warps of 32 threads as the fundamental unit for execution on a single streaming multiprocessor. A CUDA GPU performs best if warps of threads execute the same operation in a single

WILM_Egloff_v2_2010.indd 36 8/26/10 7:32:58 PM

Figure 2: Routing diagram of the parallel cyclic reduction algorithm. The symbols have the same meaning as in Figure 1.



instruction multiple data (SIMD) fashion. More details can be found in chapters 4 and 5 of [13].

Threads executing a kernel have access to four distinct memory spaces:

(i) Global memory or device memory. This can be accessed read/write by all threads in all thread blocks, but at the highest access latency.
(ii) Constant memory, which is only readable during kernel execution.
(iii) Local memory or shared memory. This memory space can be used to allocate variables that are shared by all threads in a thread block.
(iv) Private memory or registers. All variables defined in the private memory of a thread are invisible to other threads.

Further specific memory may be available. For instance, CUDA provides so-called texture memory, which is cached and may utilize hardware

WILM_Egloff_v2_2010.indd 37 8/26/10 7:33:08 PM

interpolation capabilities. From now on, let us focus on an implementation of CUDA and accordingly use its terminology such as threads and thread blocks, shared memory, and so on.

On a NVIDIA Tesla GPU there are at most 512 threads per block, which allows us to implement a tridiagonal solver for equations of dimension 512 all in parallel without any further loops, running as a single thread block.

Listing 1 provides an elegant and highly efficient CUDA implementation of a parallel cyclic reduction solver for systems of maximal dimension **dim** ≤ 512. The storage requirements of the solver are three vectors **l**, **d**, **u** for the system and one vector **h** for the right-hand side, which also will hold the result. The solver reuses **l**, **d**, **u** to store the recursively generated coefficients and therefore no additional temporary storage is required. This point is crucial because the solver works best if all the vectors can be stored in shared memory. On the other hand, excessive shared memory and register usage would limit the number of threads per block to a value strictly smaller than 512, or even prevent the kernel from executing.

For systems of dimension **dim** > 512 one thread must process multiple rows, which can be done in a variety of ways. One approach is to replace code sections

if (rank < dim) { ... }

with a loop like

for (int elem = rank; elem < dim; elem += size) { ... }

where size denotes the number of threads per block. Particular attention has to be paid to the temporary variables **lTemp, uTemp, hTemp**. Because the order of execution of the threads is not guaranteed, we must have one group of temporary variables for every iteration in the for loop. In a fully dynamic setting these variables must be allocated in shared memory. Instead of using a loop it is more efficient to use the C++ preprocessor to rule out a sufficiently large number of **if**-blocks

if (rank < dim) { ... } else if (dim <= rank && rank < 2*dim) { ... }

and to have one group of temporary variables for every **if**-block. We can then leave the temporary variables in registers. A downside is that the system dimension becomes a compile-time constant. But this is not a severe restriction because the GPU hardware resource limits can be seen as a compile-time constant.

Listing 1: Parallel cyclic reduction solver

```
1  template <class RealType>
2  __device__ void triDiagonalSystemSolve(
3      int rank                // thread index, within the block
4    , int dim                 // the dimension of the tridiagonal system
5    , RealType * l            // lower diagonal, destroyed at exit
6    , RealType * d            // diagonal, destroyed at exit
7    , RealType * u            // upper diagonal, destroyed at exit
8    , RealType * h            // right hand side and solution at exit
9    )
10 {
11         RealType lTemp, uTemp, hTemp;
12 
13     for (int span = 1; span < dim; span *= 2) {
14         if (rank < dim) {
15             if (rank - span >= 0)
16                 lTemp = (d[rank - span] != 0) ? -l[rank] / d[rank - span] : 0;
17             else
18                 lTemp = 0;
19             if (rank + span < ns)
20                 uTemp = (d[rank + span] != 0) ? -u[rank] / d[rank + span] : 0;
21             else
22                 uTemp = 0;
23             hTemp = h[rank];
24         }
25         __syncthreads();
26 
27         if (rank < dim) {
28             if (rank - span >= 0) {
29                 d[rank] += lTemp * u[rank - span];
30                 hTemp += lTemp * h[rank - span];
31                 lTemp *= l[rank - span];
32             }
33             if (rank + span < dim) {
34                 d[rank] += uTemp * l[rank + span];
35                 hTemp += uTemp * h[rank + span];
36                 uTemp *= u[rank + span];
37             }
38         }
39         __syncthreads();
40 
41         if (rank < dim) {
42             l[rank] = lTemp;
43             u[rank] = uTemp;
44             h[rank] = hTemp;
45         }
46         __syncthreads();
47     }
48 
49     if (rank < dim)
50         h[rank] /= d[rank];
51     __syncthreads();
52 }
```

Outlook

We have shown how to implement efficient parallel solvers for tridiagonal systems on a GPU. Because of the special hardware structure of a GPU, like close to the ALU shared memory and thread synchronization, the parallel cyclic reduction solver in Listing 1 is confined to run as a single thread block. A modern GPU can execute many thread blocks in parallel. For instance, a Tesla C1060 GPU has 30 multiprocessors, so that 30 thread blocks can run in parallel. Consequently, solving a single tridiagonal system can at best utilize only a small fraction of the GPU capacity. The situation changes radically if many independent tridiagonal systems can be solved in parallel.

In the next issue we will evaluate the performance of the tridiagonal solver and apply it to simultaneously pricing a large collection of similar or related derivatives at once. Such a use case is practically very relevant: a price change in an underlying asset entails the repricing of all options on this underlying. Another application is risk management, where whole books of derivatives have to be priced under multiple scenarios. We end up with enough data-parallel work to fully load a modern GPU.

REFERENCES

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

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

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

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

>

WILM_Egloff_v2_2010.indd 38 8/26/10 7:33:16 PM

[5] Bühler, H. 2009. Volatility and dividends – volatility modelling with cash dividends and simple credit risk, Institut für Mathematik.
[6] Fornberg, B. 1992. Fast generation of weights in finite difference formulas, in G.D. Byrne and W.E. Schiesser (eds), *Recent Developments in Numerical Methods and Software for ODEs/DAEs/PDEs,* World Scientific, pp. 97–123.
[7] Fornberg, B. 1998. Calculation of weights in finite difference formulas, *SIAM Review* **43**(3), 685–691.
[8] Gander, W. and Golub, G.H. 1997. Cyclic reduction – history and applications, in G.H. Golub (ed.), *Proceedings of the Workshop on Scientific Computing,* Springer-Verlag.
[9] Hockney, R.W. 1965. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM* **12**(1), 95–113.
[10] Hockney, R.W. and Jesshope, C.R. 1988. *Parallel Computers 2: Architecture, programming, and algorithms,* 2nd edn, Institute of Physics Publishing.
[11] Joshi, M. 2009. Graphical Asians, Technical report, University of Melbourne – Centre for Actuarial Studies.
[12] LAPACK. 2006. Subroutine dgtsv, http://www.netlib.org/lapack/double/dgtsv.f
[13] NVIDIA. 2009. CUDA™ Compute Unified Device Architecture Programming Guide, Technical Report 2.3.1, NVIDIA Corporation, http://www.nvidia.com/object/cuda_develop.html.
[14] Overhaus, M., Bermudez, A., Bühler, H., Ferraris, A., Jordinson, C. and Lamnouar, A. 2007. *Equity Hybrid Derivatives.* Wiley.
[15] Stone, H. 1973. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM* **20**(1), 27–38.
[16] Tavella, D. and Randall, C. 2000. *Pricing Financial Instruments – The finite difference method.* Wiley.
[17] Windcliff, H., Forsyth, P.A. and Vetzal, K.R. 2004. Analysis of the stability of the linear boundary condition for the Black–Scholes equation, *Computational Finance* **8**(1).
[18] Zhang, Y., Cohen, J. and Owens, J.D. 2010. Fast tridiagonal solvers on the GPU, Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), p. 10.

W

WILM_Egloff_v2_2010.indd Sec1:40 8/26/10 7:33:21 PM