A generic ADI solver for 3D PDEs in computational finance

Mike Giles

December 15, 2008

1 Model problem

The generic form of many 3D PDE's in computational finance is

$$\begin{aligned}\frac{\partial V}{\partial t} + \mu_1 \frac{\partial V}{\partial x_1} + \mu_2 \frac{\partial V}{\partial x_2} + \mu_3 \frac{\partial V}{\partial x_3} + \sigma_1^2 \frac{\partial^2 V}{\partial x_1^2} + \sigma_2^2 \frac{\partial^2 V}{\partial x_2^2} + \sigma_3^2 \frac{\partial^2 V}{\partial x_3^2} \\ + \beta_1 \sigma_2 \sigma_3 \frac{\partial^2 V}{\partial x_2 \partial x_3} + \beta_2 \sigma_3 \sigma_1 \frac{\partial^2 V}{\partial x_3 \partial x_1} + \beta_3 \sigma_1 \sigma_2 \frac{\partial^2 V}{\partial x_1 \partial x_2} &= s\, V\end{aligned}$$

which is solved backwards in time from some final conditions $V(x_1, x_2, x_3, T)$ until the initial time $t = 0$.

Using a standard ADI approach, each timestep can be approximated with Crank-Nicolson time-marching, and the spatial terms can be factored to give discrete equations of the form

$$A_3 A_2 A_1 \Delta V = L V^n$$

with $\Delta V = V^{n-1} - V^n$. Here L is the approximation to the spatial operator, and the matrices A_1, A_2, A_3 each define a set of tri-diagonal equations along lines in the three respective directions.

Thus, the CUDA implementation requires 4 phases:

1. Calculate the r.h.s. $R = L V^n$ and the matrices A_1, A_2, A_3;
2. Solve $A_3 Q = R$;
3. Solve $A_2 P = Q$;
4. Solve $A_1 \Delta V = P$ and add ΔV to V^n.

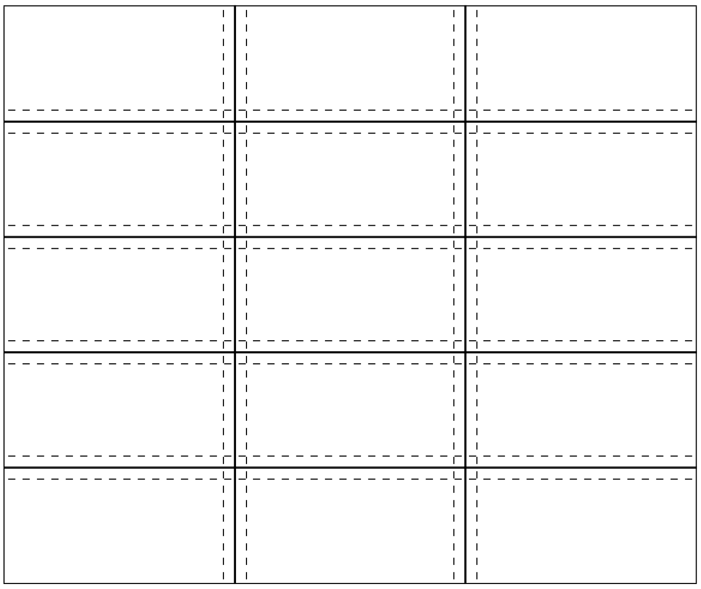

Figure 1: Partition of 2D grid into blocks with overlapping halos

2 Approach for phase 1

As with a Jacobi iteration model problem reported previously, phase 1 is handled through each CUDA block processing a $BX \times BY \times K$ column of the grid. Figure 1 illustrates the 2D partitioning in the i-j plane. This assumes that $BX \times BY$ is much smaller than $I \times J$ so that this approach gives enough blocks to keep all of the multiprocessors busy. Because of the $i \pm 1$ and $j \pm 1$ references, each block has to read in the values at the neighbouring nodes (often referred to as "halo" nodes) on all sides.

Because of the limited amount of shared memory per multiprocessor, each block works with 3 k-planes of data at a time, so there is an outer loop over k, inside which the implementation a) loads in plane $k + 1$, b) calculates and stores new values for plane k.

In optimising the performance, the key concerns are to:

- ensure coalesced global loads/stores as far as possible to minimise the communication time;
- ensure enough overlapping between active warps and/or active blocks to hide the latency on global loads/stores;
- minimise the number of integer operations required for array indexing.

To achieve the highest bandwidth between the global graphics memory and the multiprocessors, it is necessary for memory transfers to be coalesced, which requires that the 16 threads in a half-warp access a contiguous vector of data elements, with the offset for the beginning of the vector being a multiple of 16.

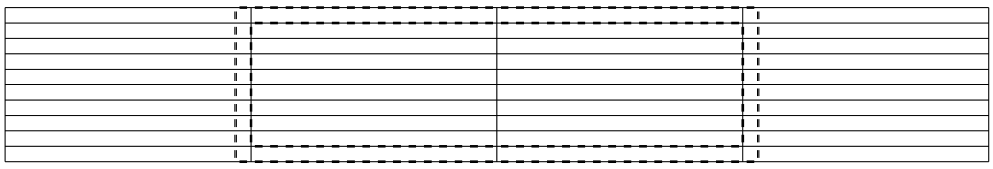

Figure 2: Layout of a block of size 32×8 plus halo

This is achieved to a large extent by using the layout indicated in Figure 2 for a block of size 32×8. The thin "pencils" indicate correctly aligned vectors of length 16, which requires in general that the block size in the i-direction is a multiple of 16, and one uses the CUDA routine `cudaMallocPitch` to allocate memory with padding to ensure that each new row starts at the beginning of one of the vectors. The interior of the block can be read into shared memory by a number of coalesced loads, as can the y-halos above and below. The x-halos on the right and left have to be read in with non-coalesced loads as they require individual elements on different vectors.

3 Approach for phases 2, 3, 4

Unlike the situation in standard distributed-memory MPI implementations of ADI algorithms, since all of the data is held in the device memory between each phase, we are entirely free to use a different data partitioning for each phase.

Accordingly, the simplest approach is to assign each tridiagonal solution to a single thread, so when solving in one direction the domain is partitioned in the other two directions.

Full memory coalescence is achieved for the tri-diagonal solves in the y and z directions; the current implementation does not achieve memory coalescence when solving in the x-direction.

4 Prototype generic solver

One of the objectives of this exercise was to see the extent to which a generic solver could be written to enable users to obtain the benefit of NVIDIA GPUs without requiring any CUDA programming.

In the demo program, the file `generic_3d_test.cu` is the only one which has to be written by the user. In this file, the user supplies a routine `model` which calculates for any given coordinates the parameters μ, σ, β, s required by the solver. The user's program also calls the routine `generic_3d_GPU` which performs the GPU calculation, giving it through the argument list information on the computational grid to be used, the timestep, etc.

5 Future developments

Although the current program has achieved its original goals of achieving good performance (factor 25x speedup using a 9800GT card) in a generic solver which can be used without any knowledge of CUDA programming, there are a number of features which need to be added:

- Improve the memory coalescence in the tridiagonal solution in the x-direction;
- Add support for non-uniform grids;
- Add a Craig-Sneyd correction to improve the accuracy to second order;
- Add an option for Rannacher time-stepping, initially using two or more timesteps of Backward Euler time integration to achieve a smoother solution;
- Add options for other boundary conditions in addition to the current b.c. which assumes zero second derivative normal to each boundary;
- Add the capability of handling American and Bermudan options;
- Write some proper documentation.