Jacobi iteration for a Laplace discretisation on a 3D structured grid

Mike Giles

April 24, 2008

1 Model problem

The application is Jacobi iteration of a Laplace discretisation on a uniform 3D grid, defined by

$$u^{(n+1)}_{i,j,k} = \frac{1}{6}\left(u^{(n)}_{i-1,j,k} + u^{(n)}_{i+1,j,k} + u^{(n)}_{i,j-1,k} + u^{(n)}_{i,j+1,k} + u^{(n)}_{i,j,k-1} + u^{(n)}_{i,j,k+1}\right),$$

for $0 \leq i < I, 0 \leq j < J, 0 \leq k < K$, and with initial data $u^{(0)}_{i,j,k}$, and fixed boundary data for $i = 0, I-1, \;\; j = 0, J-1, \quad k = 0, K-1$.

This is a good model for various iterative solvers, as well as for explicit time-marching methods. It is clearly parallelisable since all grid nodes can be updated simultaneously/independently.

Because it is such a simple model problem, it has very few operations per variable, and so the speedup is likely to be limited by the bandwidth to the global memory. The speedup is likely to be better on real applications which require a lot more floating point operations.

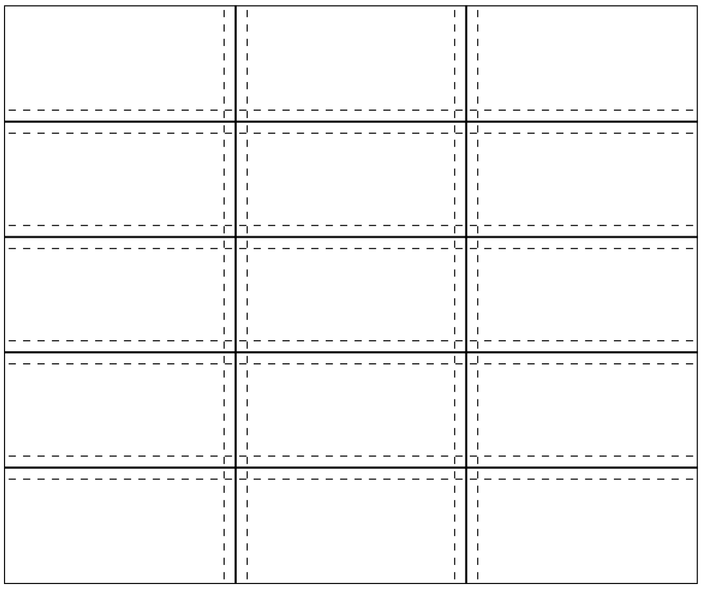

Figure 1: Partition of 2D grid into blocks with overlapping halos

2 Approach

Each CUDA block processes 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.

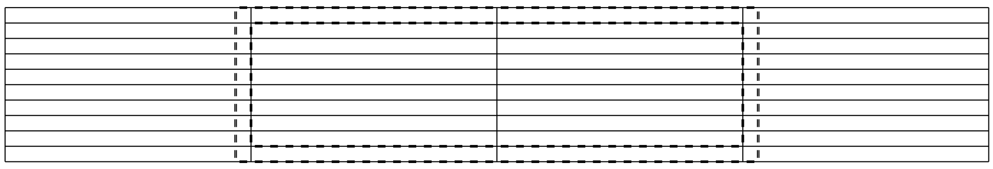

Figure 2: Layout of a block of size 32×8 plus halo

3 Array storage

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.

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.

The computation will compute new values in the interior; these can then be written to global memory by coalesced stores.

4 Other implementation details

The 2D arrangement of threads which is used matches the 2D block size. Using just one thread per block element minimises the number of integer index operations which are required after an initialisation phase. With a block size of 32×4 (which is found to be optimal on a 8800GT) there are 128 threads per block, which ensures lots of active threads for hiding global memory latency. The register and shared memory requirements are low enough to allow four blocks to be active at a time, giving additional scope for hiding memory latency, and an occupancy count of 2/3 on current hardware.

Some of the threads are also responsible for loading in the required halo elements. It is assumed that the block size is large enough that the number of halo nodes is less than the number of threads.

An integer multiply-add macro is defined to ensure that integer multiplications are performed using the more efficient `__mul24` intrinsic.

The block size is defined as a pair of integer values so that the compiler can evaluate various expressions at compile time, putting the constant values into the execution code and not into registers.

5 Performance estimate

A "back-of-the-envelope" calculation shows the extent to which performance is limited by the available bandwidth to the global device memory.

When using a block of size 32×4, each x-halo node requires the loading of a full vector of length 16, even though just one element is actually used. Taking this into account, loading in a block plus its halos requires the loading of 3 nodes of data for each interior node within the block. Hence, each iteration requires three loads and one store per interior node.

On a grid of size 256^3 with approximately 17M nodes, this equates to 67M float load/store operations, and 270MB of data.

On a 8800GT card with a bandwidth of 57.6GB/s, this corresponds to 4.7ms per iteration, whereas the actual execution time is 6.2ms. Thus the sustained bandwidth is 75% of the peak achievable.

The corresponding "Gold" code executing on a single Xeon core takes 310ms so the speedup factor is $50\times$.