GPU implementation of explicit and implicit finite difference methods in Finance

Mike Giles

Mathematical Institute, Oxford University
Oxford-Man Institute of Quantitative Finance
Oxford e-Research Centre

Endre László, István Reguly (Oxford)
Julien Demouth, Jeremy Appleyard (NVIDIA)

expanded version of presentation at GTC 2014

July 25th, 2014

GPUs

In the last 6 years, GPUs have emerged as a major new technology in computational finance, as well as other areas in HPC:

- over 1000 GPUs at JP Morgan, and also used at a number of other Tier 1 banks and financial institutions
- use is driven by both energy efficiency and price/performance, with main concern the level of programming effort required
- Monte Carlo simulations are naturally parallel, so ideally suited to GPU execution:
  - averaging of path payoff values using binary tree reduction
  - key requirement is parallel random number generation, and that is addressed by libraries such as CURAND

Finite Difference calculations

Focus of this work is finite difference methods for approximating Black-Scholes and other related multi-factor PDEs

- explicit time-marching methods are naturally parallel – again a good target for GPU acceleration
- implicit time-marching methods usually require the solution of lots of tridiagonal systems of equations – not so clear how to parallelise this
- key observation is that cost of moving lots of data to/from the main graphics memory can exceed cost of floating point computations
  - 288 GB/s bandwidth
  - 4.3 TFlops (single precision) / 1.4 TFlops (double precision)

  $\Longrightarrow$ should try to avoid this data movement

1D Finite Difference calculations

In 1D, a simple explicit finite difference equation takes the form

$$u_j^{n+1} = a_j\, u_{j-1}^n + b_j\, u_j^n + c_j\, u_{j+1}^n$$

while an implicit finite difference equation takes the form

$$a_j\, u_{j-1}^{n+1} + b_j\, u_j^{n+1} + c_j\, u_{j+1}^{n+1} = u_j^n$$

requiring the solution of a tridiagonal set of equations.

What performance can be achieved?

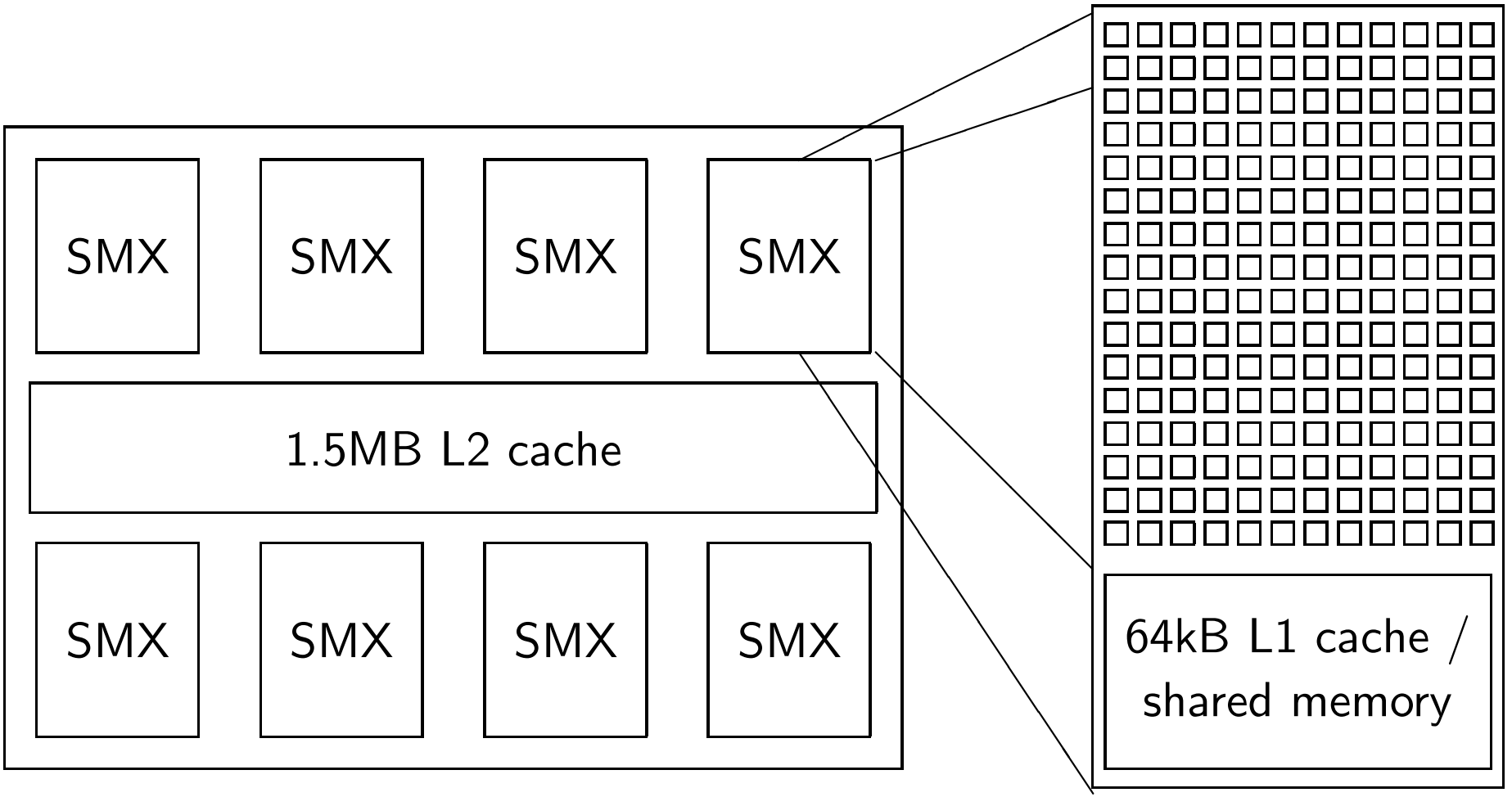
1D Finite Difference calculations

- grid size: 256 points
- number of options: 2048
- number of timesteps: 50000 (explicit), 2500 (implicit)
- K20 capable of 3.5 TFlops (single prec.), 1.2 TFlops (double prec.)

	single prec.			double prec.		
	msec	GFinsts	GFlops	msec	GFinsts	GFlops
explicit1	347	227	454	412	191	382
explicit2	89	882	1763	160	490	980
implicit1	28	892	1308	80	401	637
implicit2	33	948	1377	88	441	685
implicit3	14	643	1103	30	294	505

How is this performance achieved?

NVIDIA Kepler GPU



1D Finite Difference calculations

Approach for explicit time-marching:

- each thread block (256 threads) does one or more options
- 3 FMA (fused multiply-add) operations per grid point per timestep
- doing an option calculation within one thread block means no need to transfer data to/from graphics memory – can hold all data in SMX

1D Finite Difference calculations

- `explicit1` holds data in shared memory
- each thread handles one grid point
- performance is limited by speed of shared memory access, and cost of synchronisation

```
__shared__ REAL u[258];
...
utmp = u[i];

for (int n=0; n<N; n++) {
  utmp = utmp + a*u[i-1] + b*utmp + c*u[i+1];
  __syncthreads();
  u[i] = utmp;
  __syncthreads();
}
```

1D Finite Difference calculations

`explicit2` holds all data in registers

- each thread handles 8 grid points, so each warp (32 threads which act in unison) handles one option
- no block synchronisation required
- data exchange with neighbouring threads uses shuffle instructions (special hardware feature for data exchange within a warp)
- 64-bit shuffles performed using in software (Julian Demouth, GTC 2013)

1D Finite Difference calculations

```
for (int n=0; n<N; n++) {
  um = __shfl_up(u[7], 1);
  up = __shfl_down(u[0], 1);

  for (int i=0; i<7; i++) {
    u0   = u[i];
    u[i] = u[i] + a[i]*um + b[i]*u0 + c[i]*u[i+1];
    um   = u0;
  }

  u[7] = u[7] + a[7]*um + b[7]*u[7] + c[7]*up;
}
```

1D Finite Difference calculations

Bigger challenge is how to solve tridiagonal systems for implicit solvers.

- want to keep computation within an SMX and avoid data transfer to/from graphics memory
- prepared to do more floating point operations if necessary to avoid the data transfer
- need lots of parallelism to achieve good performance

Solving Tridiagonal Systems

On a CPU, the tridiagonal equations

$$a_i\, u_{i-1} + b_i\, u_i + c_i\, u_{i+1} = d_i, \quad i = 0, 1, \ldots, N-1$$

would usually be solved using the Thomas algorithm – essentially just standard Gaussian elimination exploiting all of the zeros.

- inherently sequential algorithm, with a forward sweep and then a backward sweep
- would require each thread to handle separate option
- threads don't have enough registers to store the required data – would require data transfer to/from graphics memory to hold / recover data from forward sweep
- not a good choice – want an alternative with reduced data transfer, even if it requires more floating point ops.

Solving Tridiagonal Systems

PCR (parallel cyclic reduction) is a highly parallel algorithm.

Starting with

$$a_i\, u_{i-1} + u_i + c_i\, u_{i+1} = d_i, \qquad i = 0, 1, \ldots, N-1,$$

where $u_j = 0$ for $j < 0, j \geq N$, can subtract multiples of rows $i \pm 1$, and re-normalise, to get

$$a'_i\, u_{i-2} + u_i + c'_i\, u_{i+2} = d'_i, \qquad i = 0, 1, \ldots, N-1,$$

Repeating with rows $i \pm 2$ gives

$$a''_i\, u_{i-4} + u_i + c''_i\, u_{i+4} = d''_i, \qquad i = 0, 1, \ldots, N-1,$$

and after $\log_2 N$ repetitions end up with solution because $u_{i \pm N} = 0$.

Solving Tridiagonal Systems

```
template <typename REAL> __forceinline__ __device__
REAL trid1_warp(REAL a, REAL c, REAL d){
  REAL b;
  uint s=1;
#pragma unroll
  for (int n=0; n<5; n++) {
    b = __rcp( 1.0f - a*__shfl_up(c,s)
                    - c*__shfl_down(a,s) );
    d = ( d  - a*__shfl_up(d,s)
             - c*__shfl_down(d,s) ) * b;
    a =      - a*__shfl_up(a,s)     * b;
    c =      - c*__shfl_down(c,s)   * b;
    s = s<<1;
  }
  return d;
}
```

1D Finite Difference calculations

Using a naive implementation of PCR we would have:

- 1 grid point per thread
- multiple warps for each option, so data exchange via shared memory, and synchronisation required – not ideal
- $O(N \log_2 N)$ floating point operations – quite a bit more than Thomas algorithm

1D Finite Difference calculations

This leads us to a hybrid algorithm: `implicit1`.

- follows data layout of `explicit2` with each thread handling 8 grid points – means data exchanges can be performed by shuffles
- each thread uses Thomas algorithm to obtain middle values as a linear function of two (not yet known) "end" values

$$u_{J+j} = A_{J+j} + B_{J+j}\, u_J + C_{J+j}\, u_{J+7}, \quad 0 < j < 7$$

- the reduced tridiagonal system of size 2×32 for the "end" values is solved using PCR
- total number of floating point operations is approximately double what would be needed on a CPU using the Thomas algorithm (but CPU division is more expensive, so similar Flop count overall?)

1D Finite Difference calculations

`implicit2` is very similar to `implicit1`, but instead of solving

$$a_j\, u_{j-1}^{n+1} + b_j\, u_j^{n+1} + c_j\, u_{j+1}^{n+1} = u_j^n$$

it instead computes the change $\Delta u_j \equiv u_j^{n+1} - u_j^n$ by solving

$$a_j\, \Delta u_{j-1} + b_j\, \Delta u_j + c_j\, \Delta u_{j+1} = d_j^n$$

and then updates u_j.

This gives better accuracy, which might be important if working in single precision.

1D Finite Difference calculations

Errors:

- `explicit1, explicit2` SP errors: 1e-5

  could improve a little by changing to

$$u_j^{n+1} = u_j^n + (a_j\, u_{j-1}^n + b_j\, u_j^n + c_j\, u_{j+1}^n)$$

- `implicit1` SP errors: 5e-5
- `implicit2` SP errors: 1e-6
- discretisation errors: 1e-4
- model errors (wrong PDE, wrong coefficients): MUCH larger

Personally, I think single precision is perfectly sufficient, but the banks still prefer double precision.

1D Finite Difference calculations

If the matrices do not change each timestep, then some parts of the tridiagonal solution do not need to be repeated each time.

Impressively, the compiler noticed this in the original version of `implicit1`, and pre-computed as much as it could, at the cost of some additional registers.

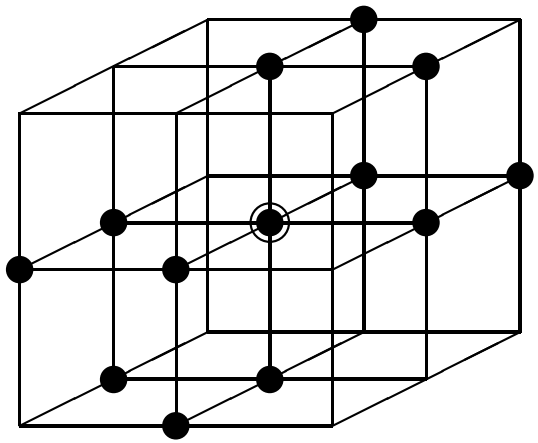
For meaningful performance results for real time-dependent matrices, I stopped this by adding a (zero) time-dependent term on the main diagonal.

However, for applications with fixed matrices, `implicit3` exploits this to pre-compute as much as possible.

3D Finite Difference calculations

What about a 3D extension on a 256^3 grid?

- memory requirements imply one kernel with multiple thread blocks to handle a single option
- kernel will need to be called for each timestep, to ensure that the entire grid is updated before the next timestep starts
- 13-point stencil for explicit time-marching



- implementation uses a separate thread for each grid point in 2D x-y plane, then marches in z-direction

3D Finite Difference calculations

- grid size: 256^3 points
- number of timesteps: 500 (explicit), 100 (implicit)
- K40 capable of 4.3 TFlops (single prec.), 1.4 TFlops (double prec.) and 288 GB/s

	single prec.			double prec.		
	msec	GFlops	GB/s	msec	GFlops	GB/s
explicit1	747	597	100	1200	367	127
explicit2	600	760	132	923	487	144
implicit1	505	360	130	921	235	139

Performance as reported by nvprof, the NVIDIA Visual Profiler

3D Finite Difference calculations

`explicit1` relies on L1/L2 caches for data reuse – compiler does an excellent job of optimising loop invariant operations

```
u2[indg] = t23                                        * u1[indg-KOFF-JOFF]
         +  t13                                       * u1[indg-KOFF-IOFF]
         + (c1_3*S3*S3 - c2_3*S3 - t13 - t23)         * u1[indg-KOFF]
         +  t12                                       * u1[indg-JOFF-IOFF]
         + (c1_2*S2*S2 - c2_2*S2 - t12 - t23)         * u1[indg-JOFF]
         + (c1_1*S1*S1 - c2_1*S1 - t12 - t13)         * u1[indg-IOFF]
         + (1.0f - c3 - 2.0f*( c1_1*S1*S1 + c1_2*S2*S2 + c1_3*S3*S3
                             - t12 - t13 - t23 ) )    * u1[indg]
         + (c1_1*S1*S1 + c2_1*S1 - t12 - t13)         * u1[indg+IOFF]
         + (c1_2*S2*S2 + c2_2*S2 - t12 - t23)         * u1[indg+JOFF]
         +  t12                                       * u1[indg+JOFF+IOFF]
         + (c1_3*S3*S3 + c2_3*S3 - t13 - t23)         * u1[indg+KOFF]
         +  t13                                       * u1[indg+KOFF+IOFF]
         +  t23                                       * u1[indg+KOFF+JOFF];
```

3D Finite Difference calculations

`explicit2` uses extra registers to hold values which will be needed again

```
u =        t23                                  * u1_om
       +   t13                                  * u1_mo
       + (c1_3*S3*S3 - c2_3*S3 - t13 - t23) * u1_m;

u1_mm = u1[indg-JOFF-IOFF];
u1_om = u1[indg-JOFF];
u1_mo = u1[indg-IOFF];
u1_pp = u1[indg+IOFF+JOFF];

u = u   +   t12                                  * u1_mm
       + (c1_2*S2*S2 - c2_2*S2 - t12 - t23) * u1_om
       + (c1_1*S1*S1 - c2_1*S1 - t12 - t13) * u1_mo
       + (1.0f - c3 - 2.0f*( c1_1*S1*S1 + c1_2*S2*S2 + c1_3*S3*S3
                     - t12 - t13 - t23 ) ) * u1_oo
       + (c1_1*S1*S1 + c2_1*S1 - t12 - t13) * u1_po
       + (c1_2*S2*S2 + c2_2*S2 - t12 - t23) * u1_op
       +   t12                                  * u1_pp;

indg += KOFF;
u1_m  = u1_oo;
u1_oo = u1[indg];
u1_po = u1[indg+IOFF];
u1_op = u1[indg+JOFF];

u = u   + (c1_3*S3*S3 + c2_3*S3 - t13 - t23) * u1_oo
       +   t13                                  * u1_po
       +   t23                                  * u1_op;
```

3D Finite Difference calculations

For implicit time-marching, the ADI discretisation requires the solution of a tridiagonal equations along each line in the x-direction, and then the same in the y- and z-directions.

`implicit1` is based on library software being written by Endre László, István Reguly and Jeremy Appleyard (NVIDIA), based on the 1D hybrid PCR code - - better than the Thomas method because it involves much less data transfer to/from graphics memory.

The clever part of the implementation is in the data transpositions required to maximise bandwidth – a bit like transposing a matrix.

3D Finite Difference calculations

The `implicit1` code has the following structure:

- kernel similar to explicit kernel to produce r.h.s.
- separate kernel for tridiagonal solution in each coordinate direction

Fairly balanced between computation and communication, provided attention is paid to maximising data coalescence.

In x-direction:

- each warp handles one tri-diagonal system
- data is contiguous in global memory
- data is loaded in coalesced way, then transposed in shared memory so each thread gets 8 contiguous elements
- care is taken to avoid shared memory bank conflicts
- process is reversed when storing data

Data transposition

What is the problem?

In this application, a warp of 32 threads wants to load in an array of 256 elements:

- natural coalesced load means warp reads in first 32, then next 32, and so on
- thread 0 ends up with elements 0, 32, 64, 96, 128, ...
- but, for hybrid PCR algorithm, thread 0 needs elements 0, 1, 2, 3, 4, 5, 6, 7
- so, how does data get re-arranged?
- more generally, how do we handle it with I elements per thread?
- same problem arises in applications where 32 threads want to load 32 objects (`structs`) each consisting of I contiguous elements

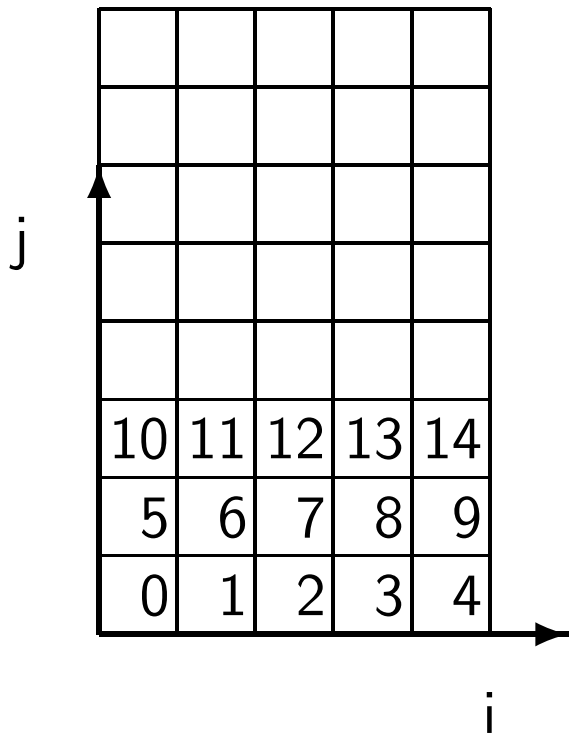
Data transposition

If I is odd, it can be done very simply using shared memory.

If j is the thread index, then

for i=0, I-1 **do**
  load global array element $j + 32 * i$
  write into shared memory $j + 32 * I$
end for

for i=0, I-1 **do**
  read from shared memory $i + j * I$
end for



Key point is that in final read, each thread in the warp is reading from a different shared memory bank – there are 32 of these.

Shared memory

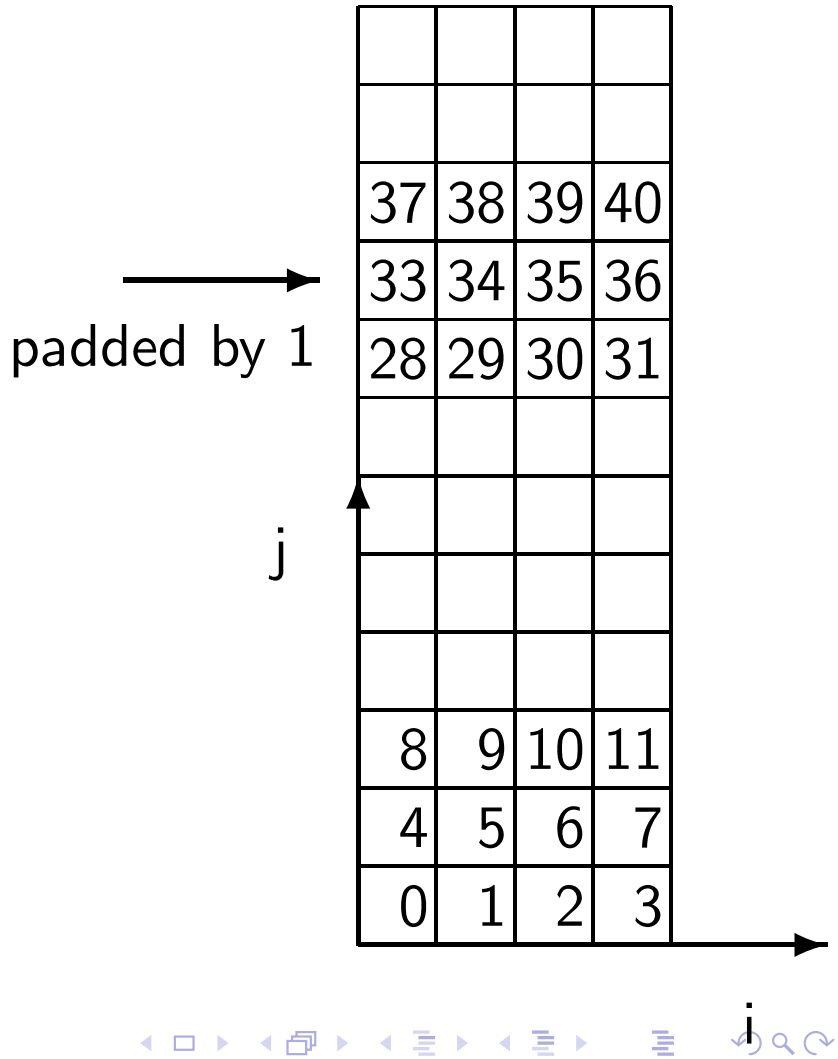
- physically, the shared memory hardware is organised into 32 memory banks
- data for shared memory location k is stored in bank $k \mod 32$
- if two threads in the warp read different data from the same memory bank then it takes longer (the requests are handled sequentially)
- generally not a major concern, but in a worst case it can perform very poorly
- no bank conflict for odd l because $j = 32$ is first positive integer with $j\ l \bmod 32 = 0$ producing a bank conflict with $j = 0$.
- when l is a power of 2, there is a problem since $j = 0, 32/l, 64/l \ldots$ all access the same bank – particularly bad for large l

Data transposition

When I is a power of 2, we avoid bank conflicts by padding the shared memory storage

for i=0, I-1 **do**
  $k = j + 32 * i$
  load global array element k
  write into shared memory $k+k/32$
end for

for i=0, I-1 **do**
  $k = i + j * I$
  read from shared memory $k+k/32$
end for



Data transposition

- this can be extended to general I (neither odd nor a power of 2)
- definitely confusing the first time you learn about it
- not worth worrying about in most applications
- can be important when developing library software to achieve the ultimate in performance

3D Finite Difference calculations

In y and z-directions:

- thread block with 8 warps to handle 8 tri-diagonal systems
- data for 8 systems is loaded simultaneously to maximise coalescence
- each thread gets 8 elements to work on
- data transposition in shared memory so that each warp handles PCR for one tridiagonal system
- then data transposition back to complete the solution and finally store the result
- quite a tricky implementation but it performs very well

Bottom line – distinctly non-trivial, so check out the code on my webpage!

Finite Difference calculations

Other dimensions?

2D:

- if the grid is small (128^2?) one option could fit within a single SMX
  - in this case, could adapt the 1D hybrid PCR method for the 2D ADI solver
  - main complication would be transposing the data between the x-solve and y-solve so that each tridiagonal solution is within a single warp
- otherwise, will have to use the 3D approach, but with solution of multiple 2D problems to provide more parallelism

4D:

- same as 3D, provided data can fit into graphics memory (buy a K40 with 12GB graphics memory!)

Conclusions

- GPUs can deliver excellent performance for financial finite difference calculations, as well as for Monte Carlo
- some parts of the implementation are straightforward, but others require a good understanding of the hardware and parallel algorithms to achieve the best performance
- some of this work will be built into future NVIDIA libraries (CUSPARSE, CUB?)
- we are now working to develop a program generator to generate code for arbitrary financial PDEs, based on an XML specification

For further info, see software and other details at
`http://people.maths.ox.ac.uk/gilesm/codes/BS_1D/`
`http://people.maths.ox.ac.uk/gilesm/codes/BS_3D/`
`http://people.maths.ox.ac.uk/gilesm/cuda_slides.html`