# Use of GPUs for Explicit and Implicit Finite Difference Methods

Mike Giles

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

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

QuanTech Conference, London

April 22nd, 2016

## GPUs

In the last few 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
  - implementations exist also for Longstaff-Schwartz least squares regression for American options – STAC-A2 testcase
  - 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

  NVIDIA Kepler GPU (current double-precision GPU, about to be superseded by new Pascal GPU)
  - 288 GB/s bandwidth
  - 5.0 TFlops (single precision) / 1.7 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?

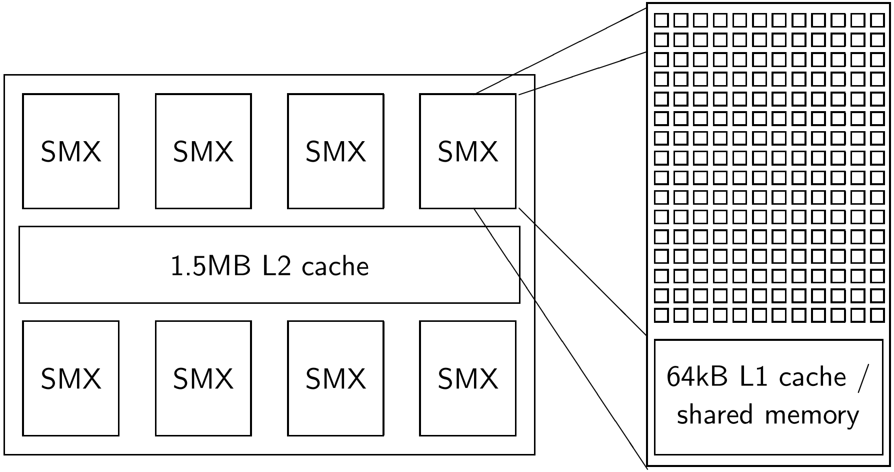
# 1D Finite Difference calculations

- grid size: 256 points
- number of options: 2048
- number of timesteps: 50000 (explicit), 2500 (implicit)
- K40 capable of 5 TFlops (single prec.), 1.7 TFlops (double prec.)

| | single prec. | | double prec. | |
|---|---|---|---|---|
| | msec | GFlops | msec | GFlops |
| explicit1 | 224 | 700 | 258 | 610 |
| explicit2 | 52 | 3029 | 107 | 1463 |
| implicit1 | 19 | 1849 | 57 | 892 |

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

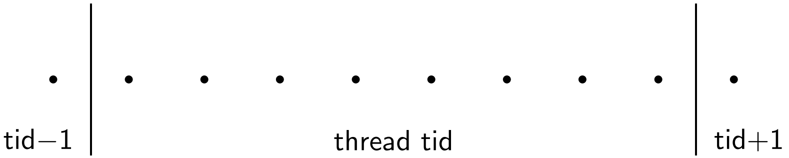
```
__shared__ REAL u[258];
...
utmp = u[i];

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)



## 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 $u_{i\pm N} = 0$.

## 1D Finite Difference calculations

`implicit1` uses a hybrid Thomas / PCR algorithm:

- 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 $2 \times 32$ 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

$$
\left(\begin{array}{c|cccccccc|c}
\hline
x & x & x & & & & & & & \\
 & x & x & x & & & & & & \\
 & & x & x & x & & & & & \\
 & & & x & x & x & & & & \\
 & & & & x & x & x & & & \\
 & & & & & x & x & x & & \\
 & & & & & & x & x & x & \\
 & & & & & & & x & x & x \\
\hline
\end{array}\right)
$$

$$
\Longrightarrow \quad
\left(\begin{array}{c|cccccccc|c}
\hline
x & x & & & & & & & x & \\
 & x & 1 & & & & & & x & \\
 & x & & 1 & & & & & x & \\
 & x & & & 1 & & & & x & \\
 & x & & & & 1 & & & x & \\
 & x & & & & & 1 & & x & \\
 & x & & & & & & 1 & x & \\
 & x & & & & & & & x & x \\
\hline
\end{array}\right)
$$

## 1D Finite Difference calculations

For comparison, we developed an implementation for two 8-core "Sandy Bridge" Xeon E5-2690 CPUs

- OpenMP used for multi-threading
- each core has two 256-bit AVX vector units (ADD and MUL)
- two CPUs are capable of 740 GFlops (single), 370 GFlops (double)
- a variety of possible vectorisation approaches
  - compiler auto-vectorisation
  - low-level vector intrinsics
  - OpenCL
  - Cilk Plus
- each core has a large 256kB L2 cache for temporary variables, so Thomas algorithm best for implicit solver

## 1D Finite Difference calculations

Performance on two 8-core "Sandy Bridge" Xeon E5-2690 CPUs

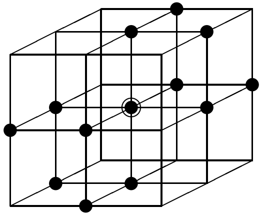
| | single prec. | | double prec. | |
|---|---|---|---|---|
| | msec | GFlop/s | msec | GFlop/s |
| explicit1 | 563 | 279 | 1188 | 132 |
| explicit2 | 398 | 394 | 781 | 201 |
| implicit1 | 187 | 139 | 470 | 48 |
| implicit2 | 157 | 166 | 473 | 47 |

- `explicit1` uses compiler auto-vectorisation (data in L1 cache)
- `explicit2` uses low-level vector intrinsics and a similar approach to the GPU implementation (data in L1 cache)
- `implicit1` uses compiler auto-vectorisation (data in L2 cache)
- `implicit2` uses low-level vector intrinsics (data in L2 cache)

## 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 5.0 TFlops (single prec.), 1.7 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 | 447 | 406 | 146 | 889 | 243 | 144 |

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

The implicit ADI discretisation requires the solution of tridiagonal equations along each coordinate direction.

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
- very important to ensure each warp loads a contiguous vector of data (coalesced read) as much as possible
- requires some careful transposition of data using shared memory

Distinctly non-trivial, so check out the paper and the code on my webpage!

## 3D Finite Difference calculations

Performance on two 8-core "Sandy Bridge" Xeon E5-2690 CPUs, with combined 100 GB/s bandwidth to main memory (66 GB/s observed)

| Dual-socket Intel Xeon E5-2690 | | | | | | |
|---|---|---|---|---|---|---|
| | single prec. | | | double prec. | | |
| | msec | GFlop/s | GB/s | msec | GFlop/s | GB/s |
| explicit1 | 1903 | 233 | 34 | 3911 | 114 | 33 |
| implicit1 | 2561 | 82 | 23 | 4966 | 42 | 23 |

- `explicit1` uses compiler auto-vectorisation
- `implicit1` uses compiler auto-vectorisation

## Final GPU / CPU comparison

Best explicit/implicit one-factor (1F) and three-factor (3F) times (ms) on one K40 GPU versus two Intel E5-2690 Xeon CPUs

| | K40 GPU | | 2 Xeon CPUs | |
|---|---|---|---|---|
| | SP | DP | SP | DP |
| 1F explicit | 52 | 107 | 398 | 781 |
| 1F implicit | 19 | 57 | 157 | 473 |
| 3F explicit | 600 | 923 | 1903 | 3911 |
| 3F implicit | 447 | 889 | 2561 | 4966 |

## 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 NVIDIA CUSPARSE library
- results show one K40 GPU is $7-8\times$ (1F) and $3-5.5\times$ (3F) faster than two 8-core Xeon E5-2690 CPUs

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`

## Kepler versus newly announced Pascal

Biggest innovations in Pascal architecture:

- HBM2 stacked memory – very high memory bandwidth
- 160 GB/s NVLink between GPUs (and IBM Power8+ CPU)
- `https://devblogs.nvidia.com/parallelforall/inside-pascal/`

| card | Tesla K40 | Tesla P100 |
|---|---|---|
| GPU | Kepler GK110B | Pascal GP100 |
| cores | 2880 | 3584 |
| clock | 750 – 875 GHz | 1300 – 1500 GHz |
| memory | 12 GB (GDDR5) | 16 GB (HBM2) |
| bandwidth | 288 GB/s | 720 GB/s |
| registers | 4 MB | 14 MB |
| SP | 4.3 TF | 10.6 TF |
| DP | 1.4 TF | 5.3 TF |