Monte Carlo and finite difference computations on GPUs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford-Man Institute for Quantitative Finance

Oxford University Mathematical Institute

Thalesians Workshop: GPUs in Finance

Imperial College, Sept 11, 2009

Random number generation

Main challenge for Monte Carlo simulation is parallel random number generation

- want to generate same random numbers as in sequential single-thread implementation
- two key steps:
  - generation of $[0,1]$ uniform random number
  - conversion to other output distributions (e.g. unit Normal)
- many of these problems are already faced with multi-core CPUs and cluster computing
- NVIDIA does not provide a RNG library, so I developed one with NAG

Random number generation

Key issue in uniform random number generation:

- when generating 10M random numbers, might have 5000 threads and want each one to compute 2000 random numbers
- need a “skip-ahead” capability so that thread n can jump to the start of its “block” efficiently
  (usually $\log N$ cost to jump N elements)

Random number generation

mrg32k3a (Pierre l'Ecuyer, '99, '02)

- popular generator in Intel MKL and ACML libraries
- pseudo-uniform $(0,1)$ output is

$$(x_{n,1} - x_{n,2} \mod m_1) \, / \, m_1$$

where integers $x_{n,1}$, $x_{n,2}$ are defined by recurrences

$$\begin{aligned} x_{n,1} &= a_1 \, x_{n-2,1} - b_1 \, x_{n-3,1} \mod m_1 \\ x_{n,2} &= a_2 \, x_{n-1,2} - b_2 \, x_{n-3,2} \mod m_2 \end{aligned}$$

$a_1 = 1403580$, $b_1 = 810728$, $m_1 = 2^{32} - 209$,
$a_2 = 527612$, $b_2 = 1370589$, $m_2 = 2^{32} - 22853$.

Random number generation

- Both recurrences are of the form

$$y_n = A\, y_{n-1} \mod m$$

where y_n is a vector $y_n = (x_n,\ x_{n-1},\ x_{n-2})^T$ and A is a 3×3 matrix. Hence

$$y_{n+2^k} = A^{2^k} y_n \mod m = A_k\, y_n \mod m$$

where A_k is defined by repeated squaring as

$$A_{k+1} = A_k\, A_k \mod m, \qquad A_0 \equiv A.$$

Can generalise this to jump N places in $O(\log N)$ operations.

Random number generation

- output distributions:
  - uniform
  - exponential: trivial
  - Normal: Box-Muller or inverse CDF
  - Gamma: using “rejection” methods which require a varying number of uniforms and Normals to generate 1 Gamma variable
- producing Normals with **mrg32k3a**:
  - 2400M values/sec on a 216-core GTX260
  - 70M values/sec on a Xeon using Intel’s VSL
- have also implemented a **Sobol** generator to produce quasi-random numbers

Monte Carlo simulation

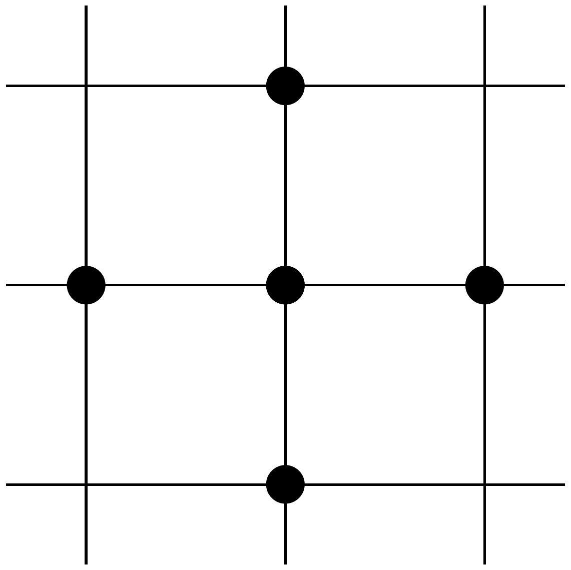
Other challenges in Monte Carlo simulation:

- be careful to ensure coalesced memory transfers to maximise memory bandwidth
- keep constants in special constant memory, as far as possible
- local volatility surface – use texture mapping for efficiency?
- Longstaff-Schwartz – need to combine regression matrix contributions from each path using a global reduction
- complex scripting of payoffs – transfer path values back to CPU for payoff evaluation?

Finite Difference Model Problem

Jacobi iteration to solve discretisation of Laplace equation

$$V_{i,j}^{n+1} = \frac{1}{4}\left(V_{i+1,j}^{n} + V_{i-1,j}^{n} + V_{i,j+1}^{n} + V_{i,j-1}^{n}\right)$$



Finite Difference Model Problem

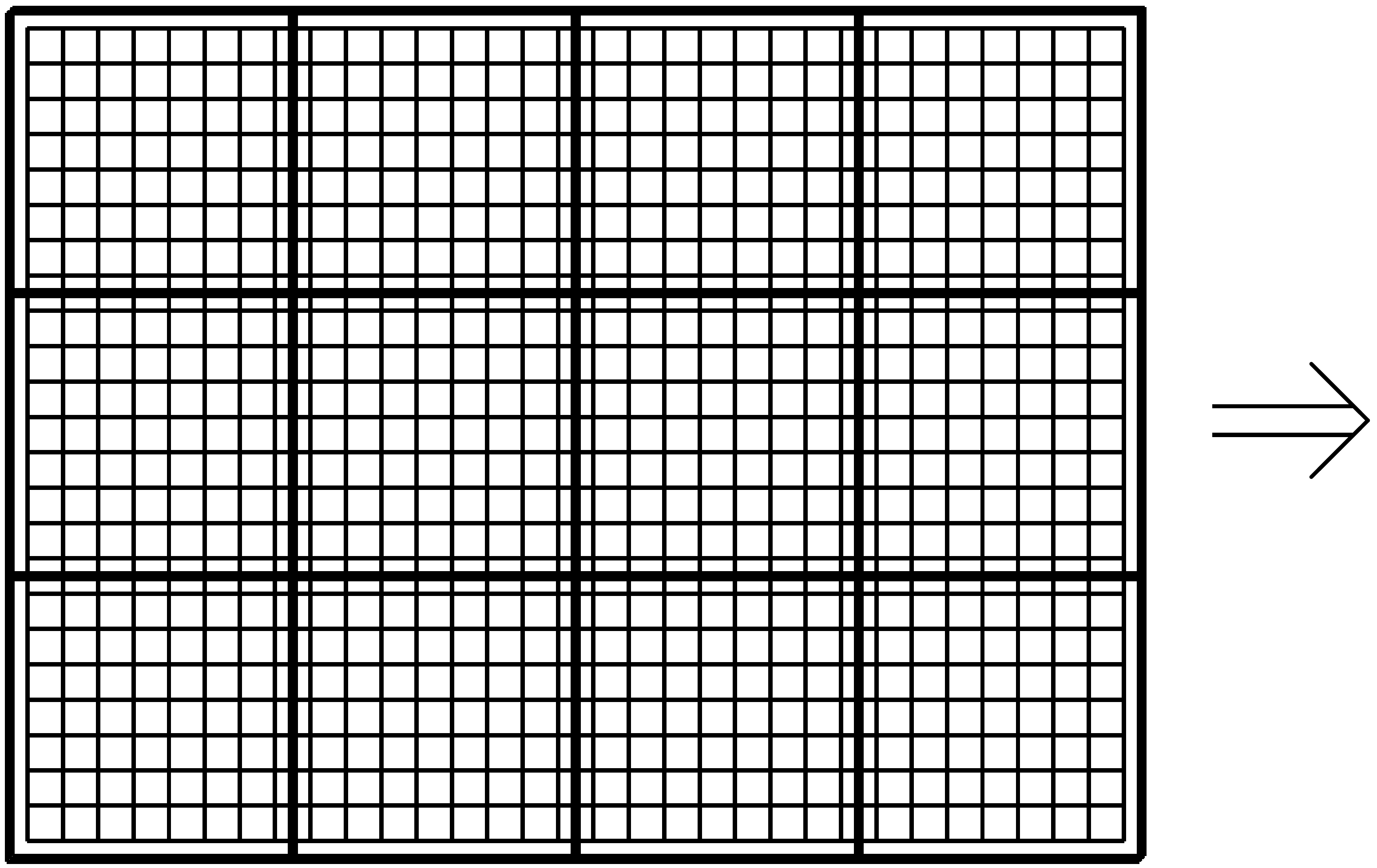
How should this be programmed?

First idea: each thread does one grid point, reading in directly from graphics memory the old values at the 4 neighbours (6 in 3D).

Performance would be awful:

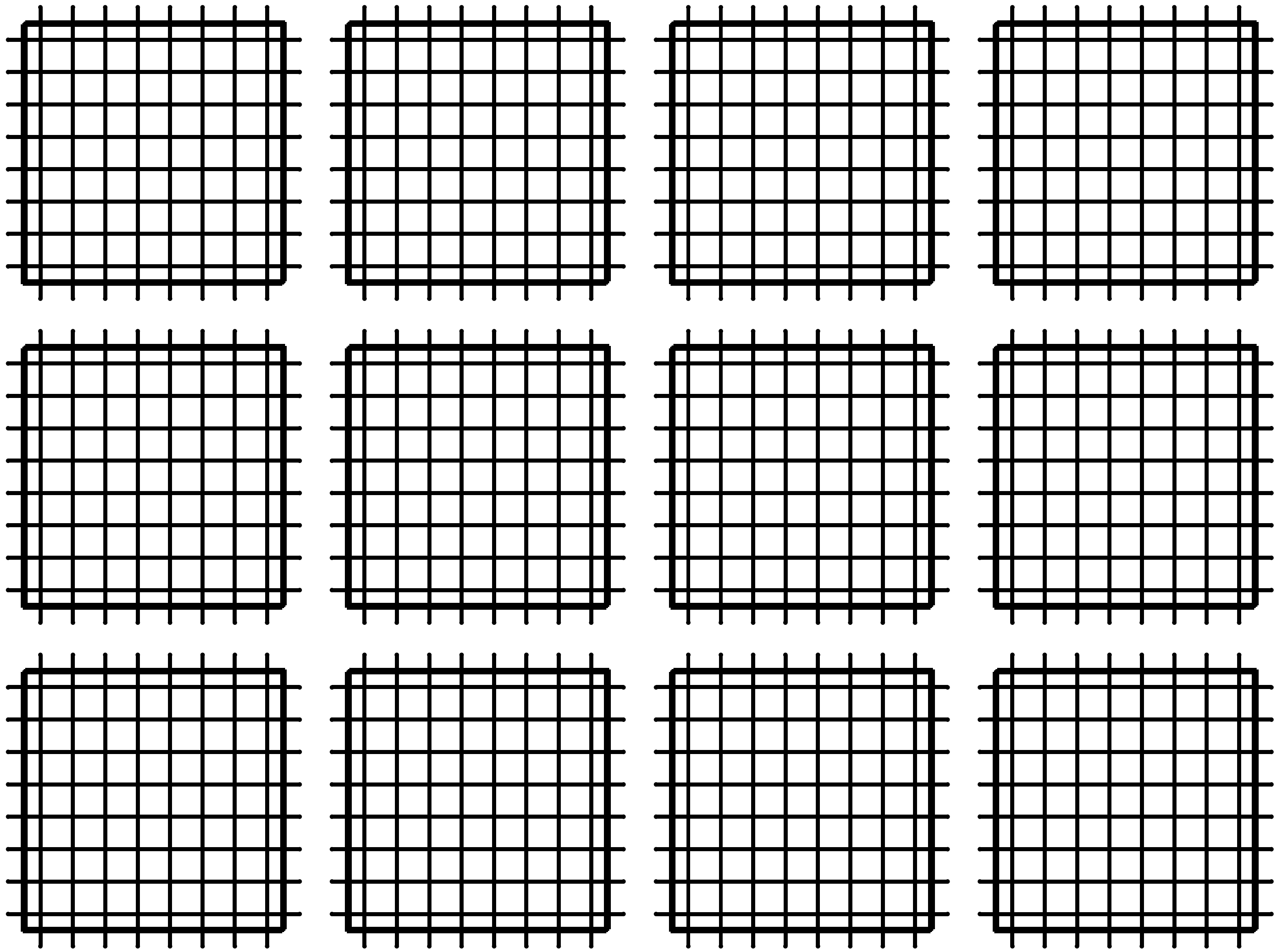
- each old value read in 4 times (6 in 3D)
- although reads would be contiguous (all read from the left, then right, etc.) they wouldn't have the correct alignment (factor $2\times$ penalty on new hardware, even worse on old)
- overall a factor $10\times$ reduction in effective bandwidth (or $10\times$ increase in read time)

Finite Difference Model Problem

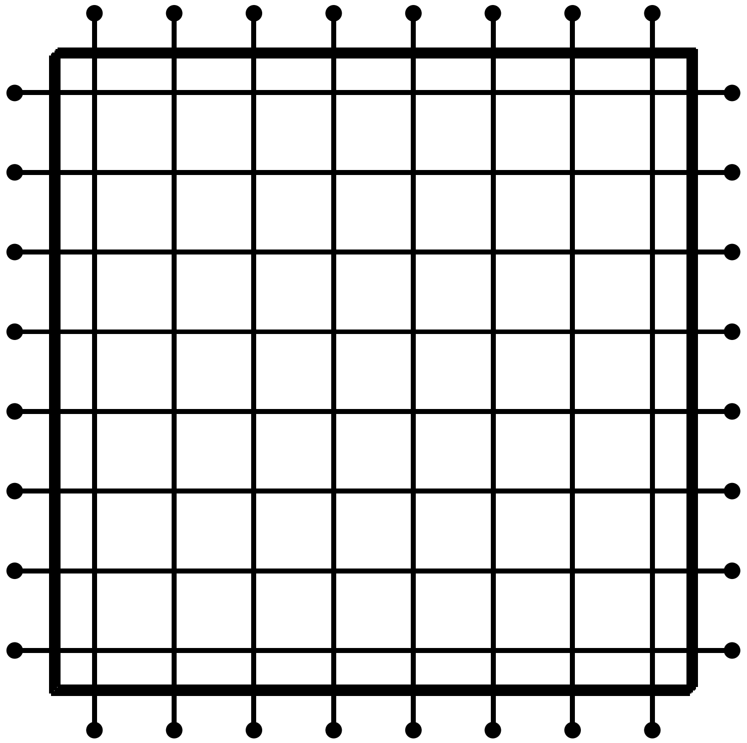


Second idea: take ideas from distributed-memory parallel computing and partition grid into pieces

Finite Difference Model Problem



Finite Difference Model Problem



Each block of threads will work with one of these grid blocks, reading in old values (including the “halo nodes” from adjacent partitions) then computing and writing out new values

Finite Difference Model Problem

Key point: old data is loaded into shared memory:

- each thread loads in the data for its grid point (coalesced) and maybe one halo point (only partially coalesced)
- need a `__syncthreads();` instruction
  to ensure all threads have completed this before any of them access the data
- each thread computed its new value and writes it to graphics memory

Finite Difference Model Problem

2D finite difference implementation:

- good news: $30\times$ speedup relative to Xeon single core, compared to $4.5\times$ speedup using OpenMP with 8 cores
- bad news: grid size has to be 1024^2 to have enough parallel work to do to get this performance
- in a real financial application, more sensible to do several 2D calculations at the same time, perhaps with different payoffs

Finite Difference Model Problem

3D finite difference implementation:

- insufficient shared memory for whole 3D block, so hold 3 working planes at a time
- key steps in kernel code:
  - load in $k=0$ z-plane (inc x and y-halos)
  - loop over all z-planes
    - load $k+1$ z-plane
    - process k z-plane
    - store new k z-plane
- $50\times$ speedup relative to Xeon single core, compared to $5\times$ speedup using OpenMP with 8 cores.

Finite Difference Model Problem

Third idea: use texture memory

- basic approach is the same
- difference is in loading of “old” data using texture mapping
- local texture cache means values are only transferred from graphics memory once (?)
- “cache line” transfer is coalesced as far as possible (?)
- not as fast as hand-coded version but much simpler
- no documentation on cache management, so hard to predict/understand performance

More on Finite Differences

ADI implicit time-marching:

- each thread handles tri-diagonal solution along a line in one direction
- easy to get coalescence in y and z directions, but not in x-direction
- again roughly 10$\times$ speedup compared to two quad-core Xeons

More on Finite Differences

Implicit time-marching with iterative solvers:

- BiCGStab: each iteration similar to Jacobi iteration except for need for global dot-product
- See “reduction” example and documentation in CUDA SDK for how shared memory is used to compute partial sum within each block, and then these are combined at a higher level to get the global sum
- ILU preconditioning could be tougher

More on Finite Differences

Generic 3D financial PDE solver:

- available on my webpages
- development funded by TCS/CRL (leading Indian IT company)
- uses ADI time-marching
- designed for user to specify drift and volatility functions as C code – no need for user to know anything about CUDA programming
- an example of what I think is needed to hide complexities of GPU programing

Further information

LIBOR and finite difference test codes
`www.maths.ox.ac.uk/∼gilesm/hpc/`

NAG numerical routines for GPUs
`www.nag.co.uk/numeric/GPUs/`

NVIDIA's CUDA homepage
`www.nvidia.com/object/cuda_home.html`

NVIDIA's computational finance page
`www.nvidia.com/object/computational_finance.html`