Using GPUs for Computational Finance

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford-Man Institute of Quantitative Finance

Oxford University Mathematical Institute

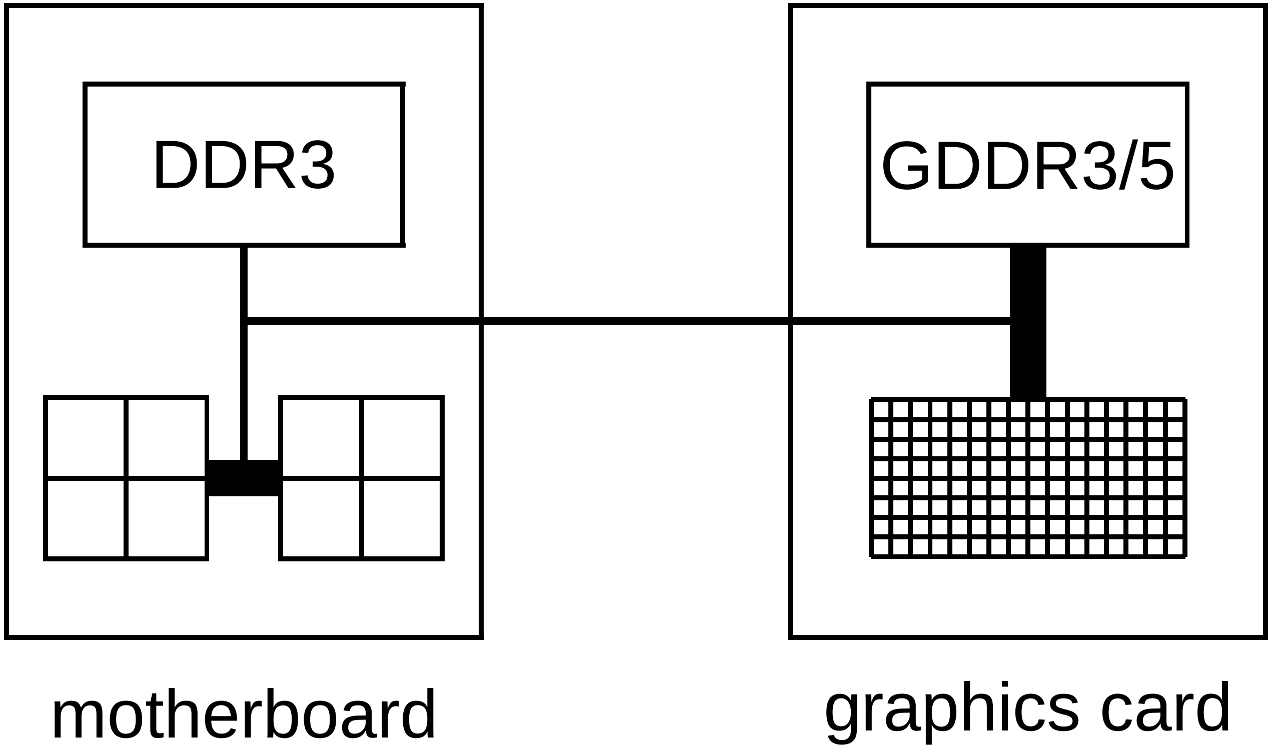
NVIDIA CUDA Fellow for Computational Finance

Fields Institute Workshop on Computational Methods in Finance

Toronto, March 22–24, 2010

CPUs and GPUs

Typically, a PCIe graphics card with a many-core GPU sits inside a PC/server with one or two multicore CPUs:



CPUs and GPUs

- CPUs have up to 6 cores (each with a SSE vector unit) and 10-30 GB/s bandwidth to main system memory
- NVIDIA GPUs have up to 30×8 cores on a single chip and 100+ GB/s bandwidth to graphics memory
- offer 50–100× speedup relative to a single CPU core
- roughly 10× speedup relative to two quad-core Xeons
- also 10× improvement in price/performance and energy efficiency

How is this possible? Much simpler cores (SIMD units, no out-of-order execution or branch prediction) designed for vector computing, not general purpose

CPUs and GPUs

Is this GPU advantage sustainable? Yes!

- IBM, AMD and Intel all producing or developing GPUs too
- NVIDIA has a good headstart on software side with CUDA environment
- new OpenCL software standard (based on CUDA and pushed by Apple) will probably run on all platforms
- driving applications are:
  - computer games “physics”
  - video (e.g. HD video decoding)
  - computational science
  - computational finance
  - oil and gas

Why GPUs will stay ahead

Technical reasons:

- SIMD units means larger proportion of chip devoted to floating point computation (but CPUs will respond with longer vector units – AVX)
- tightly-coupled fast graphics memory means much higher bandwidth

Commercial reasons:

- CPUs driven by price-sensitive office/home computing; not clear these need vastly more speed
- CPU direction may be towards low cost, low power chips for mobile and embedded applications
- GPUs driven by high-end applications – prepared to pay a premium for high performance

Use in computational finance

- Bloomberg has a large cluster:
  - 48 NVIDIA Tesla units, each with 4 GPUs
  - alternative to buying 2000 CPUs
- BNP Paribas has a small cluster:
  - 2 NVIDIA Tesla units
  - replacing 250 dual-core CPUs
  - factor 10x savings in power (2kW vs. 25kW)
- lots of other banks doing proof-of-concept studies
  - my impression is that IT groups are keen, but quants are concerned about effort involved
- Several ISV's now offer software based on CUDA

Programming

Big breakthrough in GPU computing has been NVIDIA's development of CUDA programming environment

- C plus some extensions and some C++ features
- host code runs on CPU, CUDA code runs on GPU
- explicit movement of data across the PCIe connection
- very straightforward for Monte Carlo applications, once you have a random number generator
- significantly harder for finite difference applications (but will be much easier with next-generation GPU)
- see example codes on my website

My experience

- Random number generation (mrg32k3a/Normal):
  - 2500M values/sec on GTX 280
  - 70M values/sec/core on Xeon using Intel’s VSL
- LIBOR Monte Carlo testcase:
  - 100x speedup on GTX 280 compared to single thread on Xeon
- 3D PDE application:
  - factor 50x speedup on GTX 280 compared to single thread on Xeon
  - factor 10x speedup compared to two quad-core Xeons

GPU results are all single precision – double precision is currently 2-4 times slower, no more than factor 2 in future

Random number generation

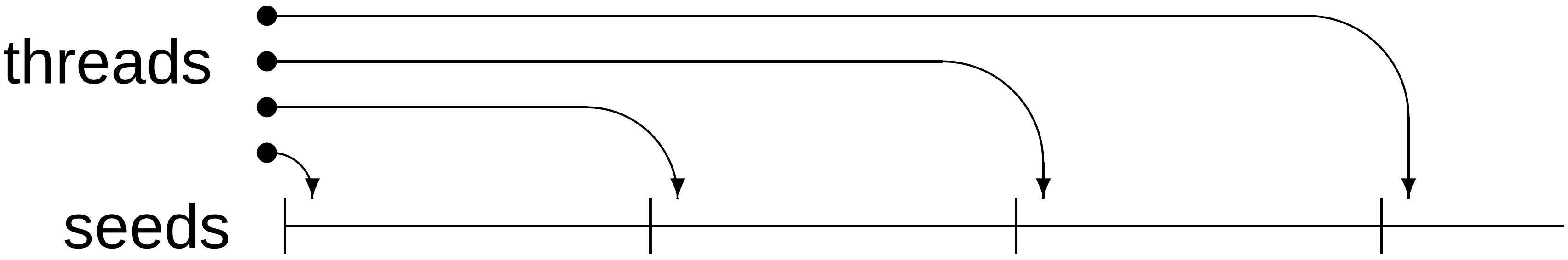
Main challenge for Monte Carlo simulation is parallel random number generation

- 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 have 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.

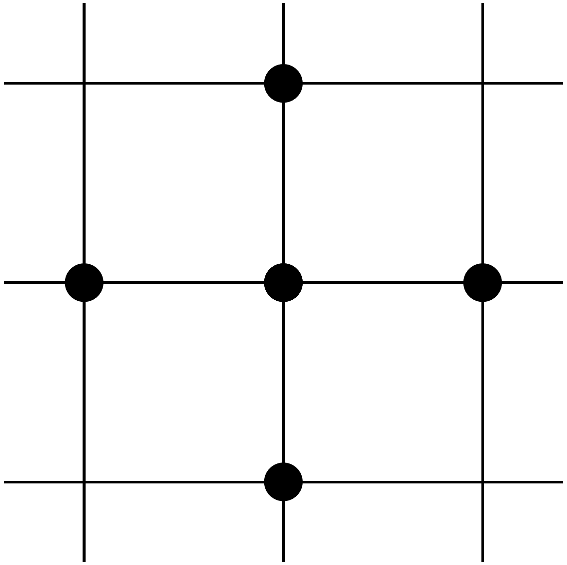
Random number generation

- 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
  - 6500M Normals/sec on a 216-core GTX260 using an inverse CDF implementation

Finite Difference Model Problem

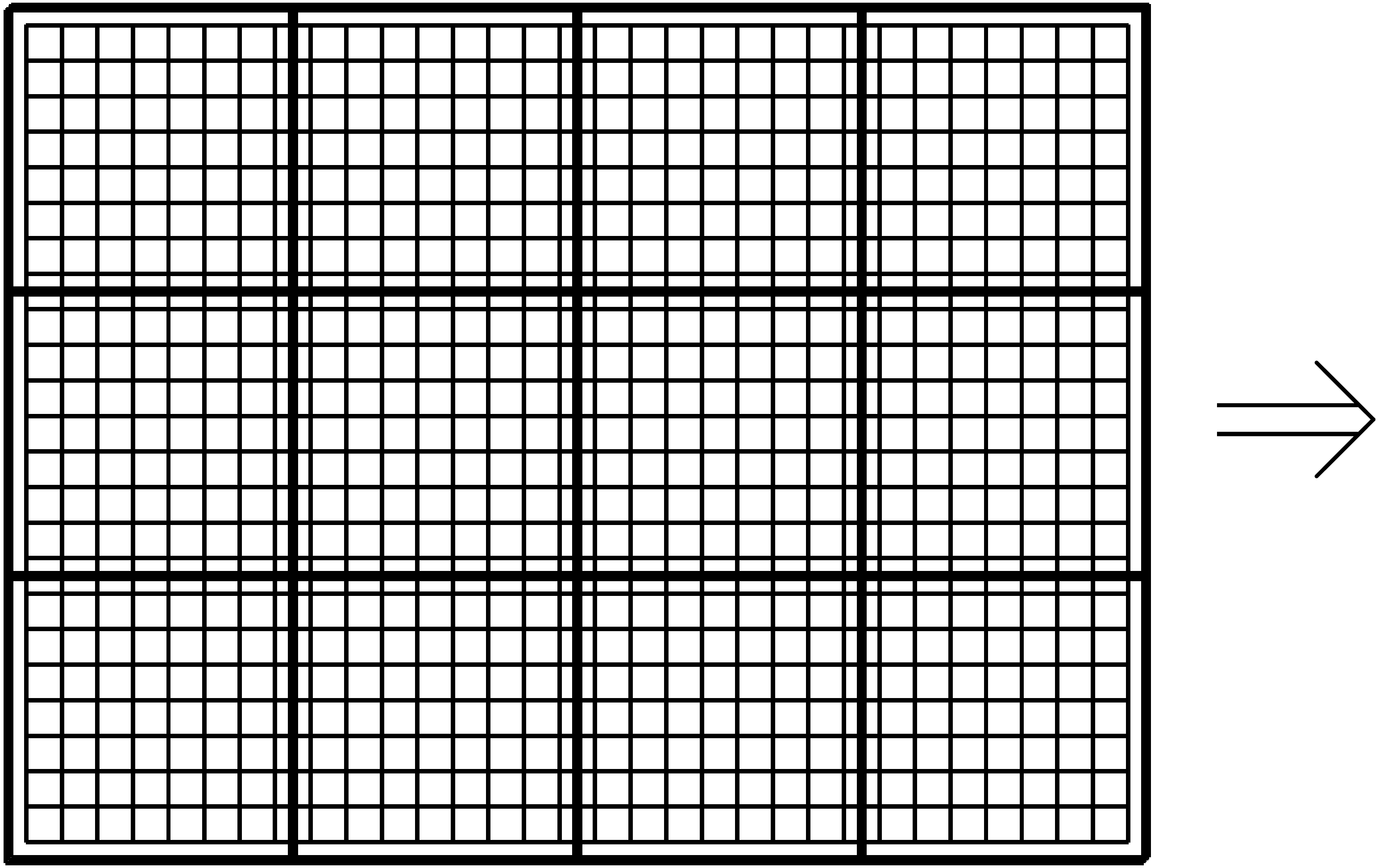
Jacobi iteration to solve discretisation of Laplace equation

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



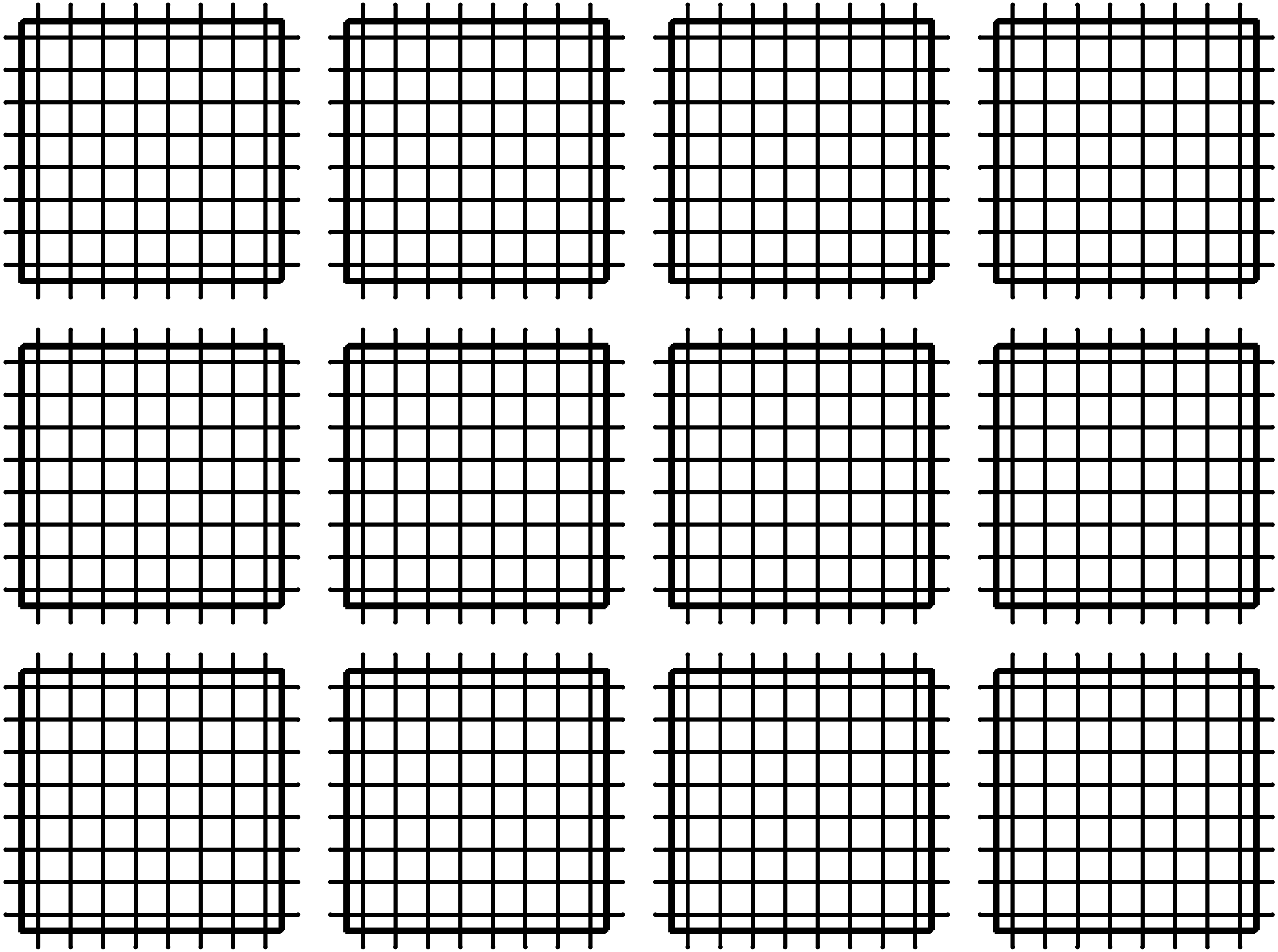
How should this be programmed?

Finite Difference Model Problem

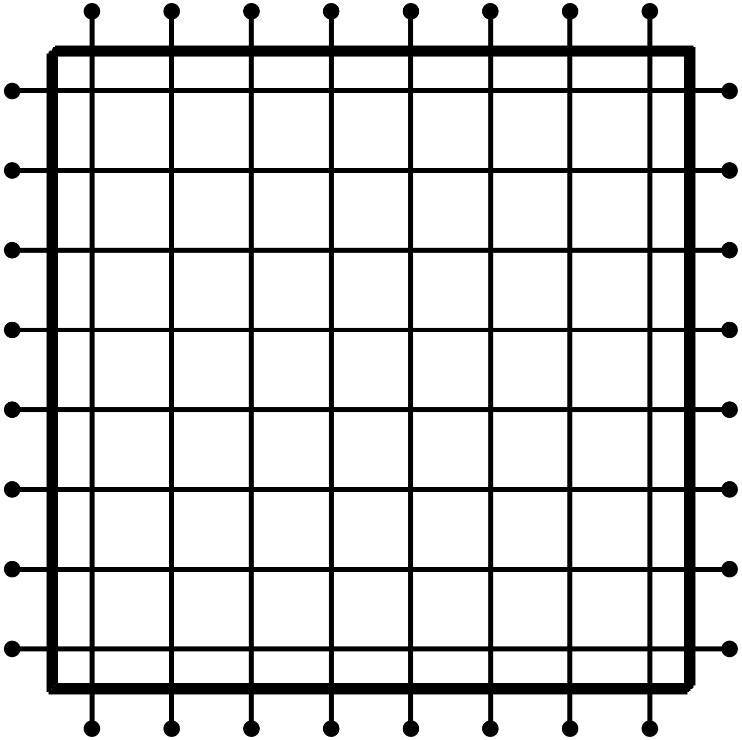


Key idea: take ideas from distributed-memory parallel computing and partition grid into pieces

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

Old data is loaded into shared memory:

- each thread loads in the data for its grid point and maybe one halo point
- data is then available for neighbouring threads when they need it
- each thread computed its new value and writes it to graphics memory
- this is slightly tedious, manually programming to duplicate what is done in a cache in a CPU; will be much simpler on new GPUs which have a cache

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.

More on Finite Differences

ADI implicit time-marching:

- each thread handles tri-diagonal solution along a line in one direction
- again roughly 10× speedup compared to two quad-core Xeons

Implicit time-marching with iterative solvers:

- BiCGStab: each iteration similar to Jacobi iteration except for need for global dot-product (see “reduction” example in CUDA SDK)
- 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

Programming

Software alternatives:

- OpenCL
  - no personal experience
  - looks similar to the lower-level CUDA device API
  - I'm waiting for
    - simpler higher-level layer
    - experience of others on pros/cons versus CUDA
    - competitive hardware from other vendors

Programming

Software alternatives:

- Microsoft's DX Compute
  - unlikely to be used for scientific computing, but maybe for games and multimedia applications
- Intel: Ct, TBB, SSE/AVX vectors, `icc`, OpenCL
  - I find range of alternatives confusing – look to Intel for clear guidance on pros and cons
  - I think SSE/AVX vectors may offer best performance but programming is tedious (worse than CUDA?)
  - I hope OpenCL support is good (should map very naturally to SSE/AVX vectors)

Current developments

NVIDIA: new “Fermi” GPUs shipping in April

- 512 SP cores (1.5 TFlops), 256 DP cores (750 GFlops)
- 3GB ECC memory
- L1 / L2 cache – will simplify programming

AMD: new GPUs out now with OpenCL support, but focus is on consumer products

IBM: Cell hard to program – terminating further development for scientific computing

Intel: Larrabee GPU badly delayed (2011 / 2012 / never?) but watch AVX vectors for mainstream CPUs though performance maybe limited by available bandwidth

What is needed now?

Skilled manpower, training:

- 50+ on Oxford CUDA mailing list: students and post-docs in almost all science departments
- 1-week CUDA course last summer, to be repeated ths summer
- in 3 years time, many PhDs in computational science will have these skills

More development of libraries, high-level packages:

- Monte Carlo simulation
- PDE solvers
- . . .

Further information

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

5 day CUDA course, and 1/2 day introduction
www.maths.ox.ac.uk/∼gilesm/cuda/
www.maths.ox.ac.uk/∼gilesm/pp10/

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