Computational Finance using GPUs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford-Man Institute of Quantitative Finance

Oxford e-Research Centre

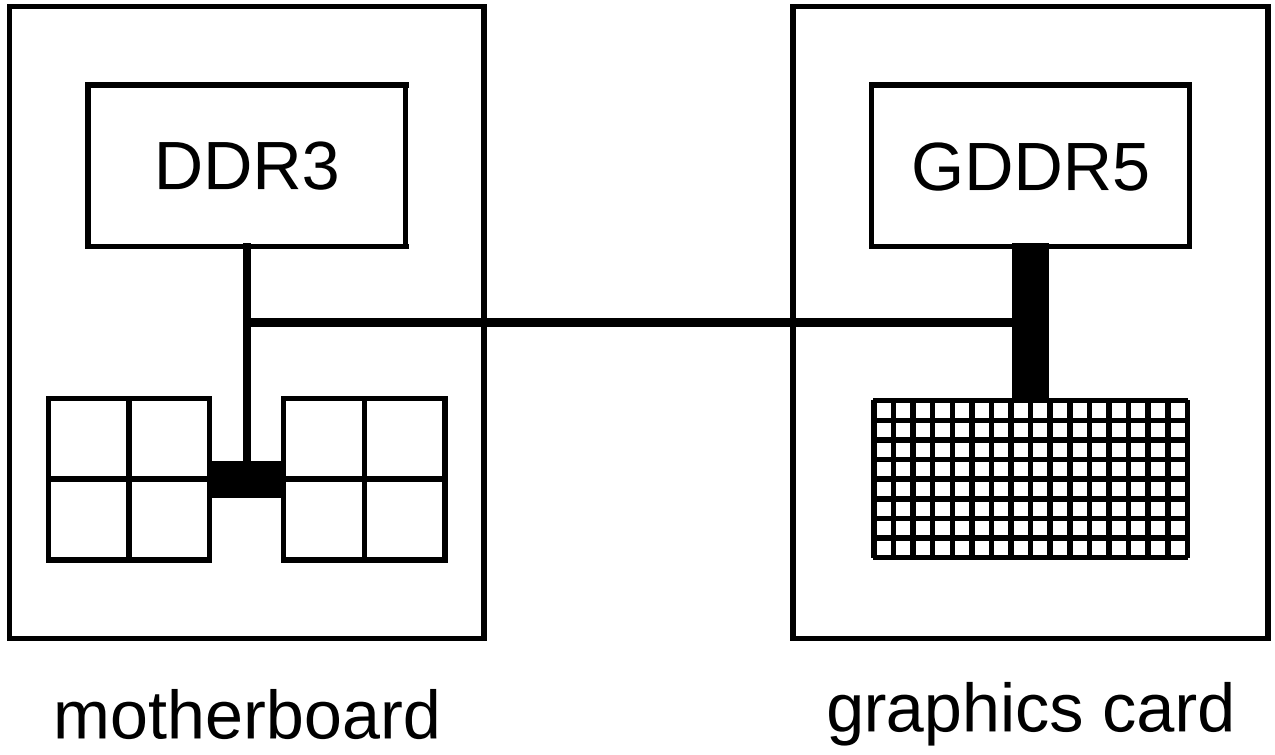
NVIDIA CUDA Fellow for Computational Finance

Workshop at Stefan Banach International Mathematical Center

Warsaw, August 18, 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 12 cores (each with a SSE vector unit) and up to 30 GB/s bandwidth to main system memory
- NVIDIA Fermi GPUs have up to $14 \times 32 = 448$ cores on a single chip and up to 150 GB/s bandwidth to graphics memory
- offer 100$\times$ speedup relative to a single CPU core
- roughly 10$\times$ speedup relative to two Xeons
- also 10$\times$ 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!

- AMD producing similar GPUs too, and Intel designing similar new chips (Knights Ferry)
- 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
  - some were waiting for new NVIDIA Fermi GPUs with ECC memory and faster double precision
- 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
- was harder for finite difference applications but now a lot easier because of caches on Fermi GPUs
- 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 half the speed, as with CPU SSE instructions

Random number generation

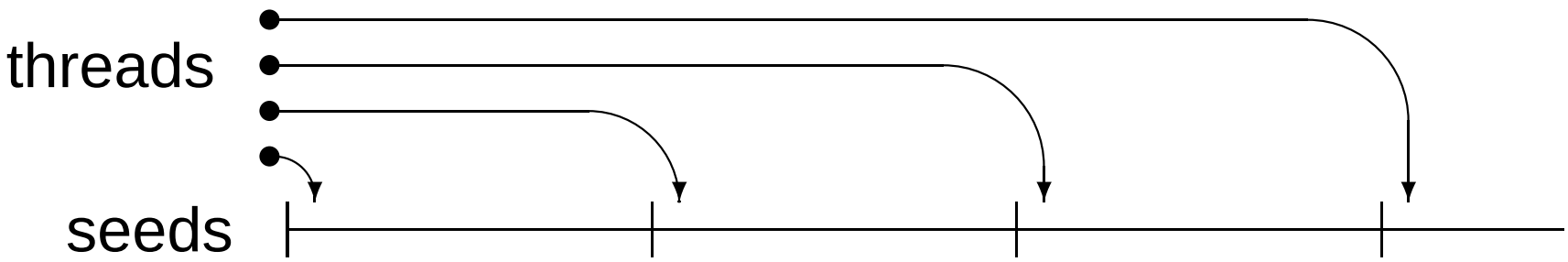
Main challenge for Monte Carlo simulation is parallel random number generation

- 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 – freely available to academics

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

Random number generation

Library provides routines at two levels:

- host level
  routine is called by a user's code on the host, but executes on the GPU and puts output into arrays in the graphics memory
- device level
  routine is called by uesr's code on the GPU, produce number "on-the-fly" ready for immediate use
- the device level is usually a little more efficient because there's no data transfer, but it uses up some registers and that might be a prob;em in some cases

Random number generation

Sobol generator for quasi-random numbers:

- same approach as **mrg32k3a**, very efficient skip-ahead
- conversion of quasi-uniforms into quasi-Normals uses a novel implementation of the inverse error function
- also implemented Brownian Bridge construction
- 6500M Normals/sec on a 216-core GTX260

Mersenne twister:

- very popular in the banks because of its extremely long period
- large “state” makes it more difficult to parallelise (not many registers per thread in a GPU)
- also, skip-ahead is $O(\log N)$ but much more expensive

Monte Carlo simulation

Given a library for parallel random number generation, the rest of a Monte Carlo simulation is straightforward:

- within the GPU, each thread performs one or more path calculation
- each thread works with a different set of random numbers, so each path simulation is independent
- results from different threads need to be averaged; easiest way is to transfer the outputs back to the CPU
- averaging on the GPU is trickier – needs cooperation / communication between different threads
- for an example, see practical 2 from my CUDA course

CUDA programming

At the top level, we have a master process which runs on the CPU and performs the following steps:

1. initialises card
2. allocates memory in host and on device
3. copies data from host to device memory
4. launches multiple copies of execution “kernel” on device
5. copies data from device memory to host
6. repeats 3-5 as needed
7. de-allocates all memory and terminates

CUDA programming

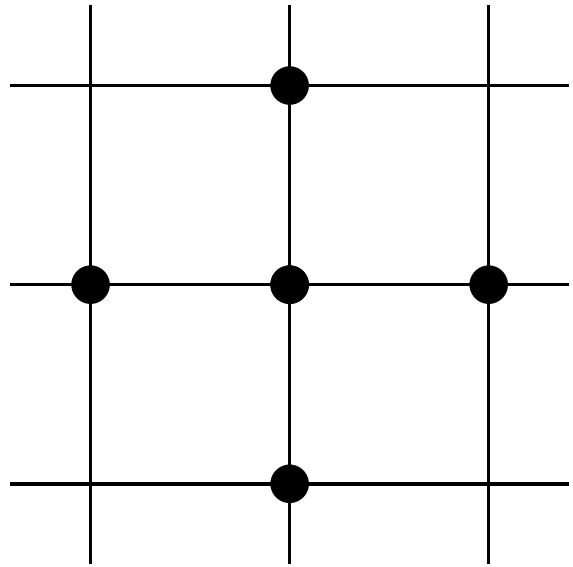
At a lower level, within the GPU:

- each copy of the execution kernel executes on an SM
- if the number of copies exceeds the number of SMs, then more than one will run at a time on each SM if there are enough registers and shared memory, and the others will wait in a queue and execute later
- all threads within one copy can access local shared memory but can’t see what the other copies are doing (even if they are on the same SM)
- there are no guarantees on the order in which the copies will execute

Finite Difference Model Problem

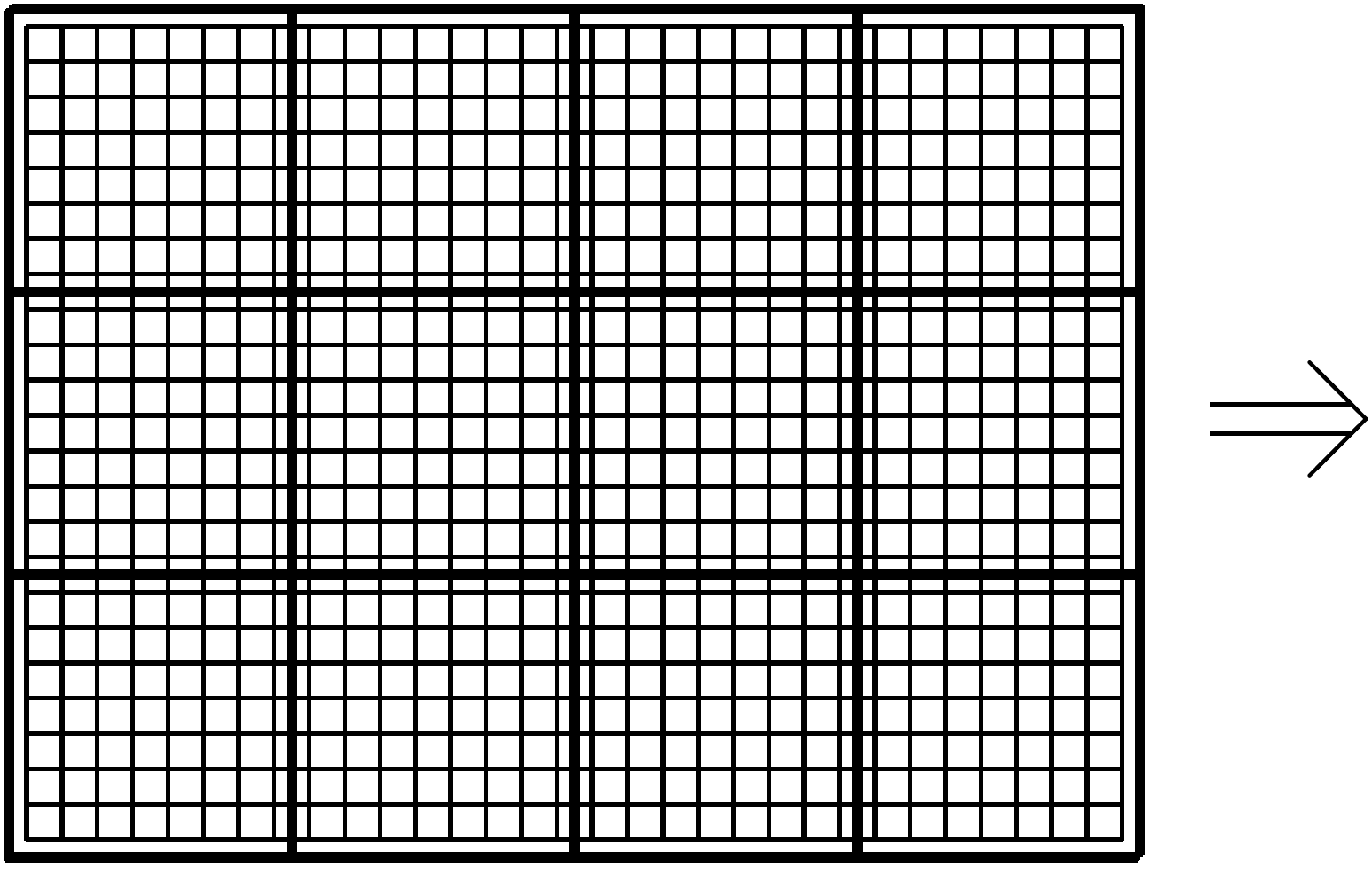
Jacobi iteration to solve discretisation of Laplace equation

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



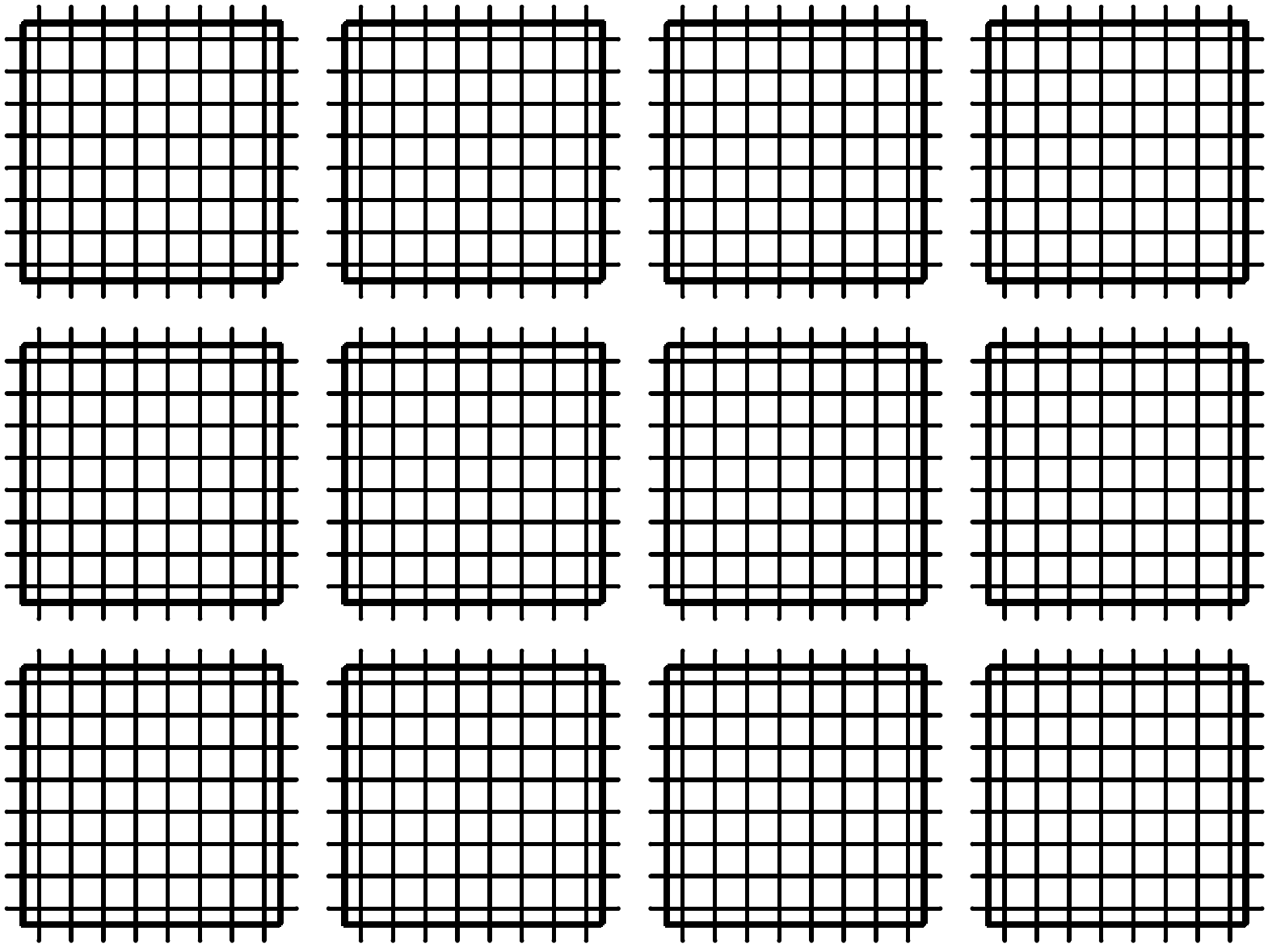
How should this be programmed?

Finite Difference Model Problem

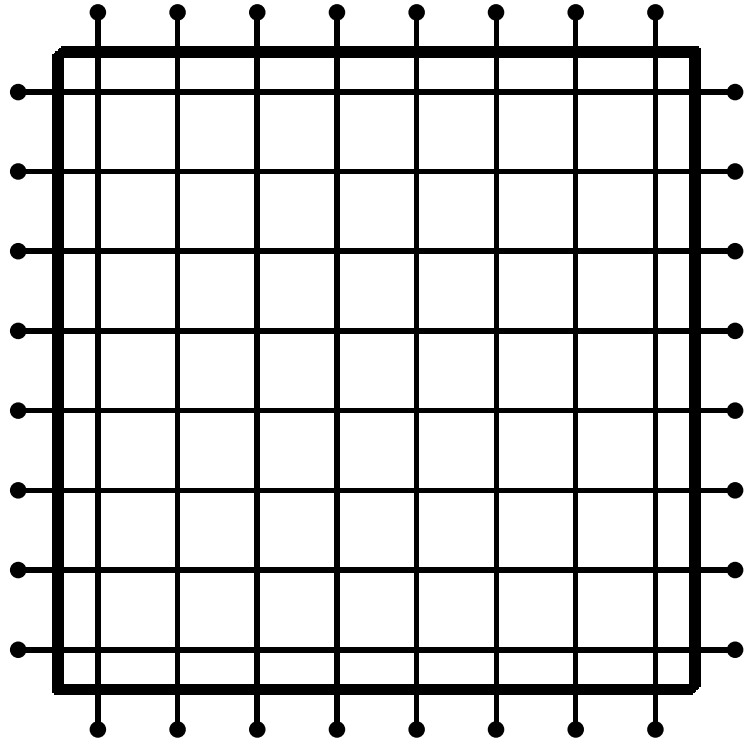


Key idea: take ideas from distributed-memory parallel computing and partition grid into pieces

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

Implementation on old GPUs used local 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 computes 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;
  – much simpler on new Fermi GPUs with a cache
- see code examples on my course webpage

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
- (same comment would apply to 1D simulations – best to do many at the same time)

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
- again much simpler on the new Fermi GPUs
- $50\times$ 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$\times$ 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

- 448 SP cores (1.5 TFlops), 224 DP cores (750 GFlops)
- 3GB ECC memory
- L1 / L2 cache

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

IBM: abandoned development of Cell GPU

Intel: abandoned development of Larrabee GPU, but continuing with Knights Ferry chip for HPC; I've heard its performance is comparable to Fermi

How to get started?

- buy a 1GB GTX 460 graphics cards
  - about 200 euro + tax
  - needs a double-width PCIe slot (preferably $16\times$)
  - uses about 150W, needs two 6-pin power connectors
  - will fit in most big PC cases
- read lecture notes and do practicals from online courses:
  - Wen-mei Hwu (UIUC)
  - me
- look through examples in CUDA SDK
- start experimenting!

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`