Financial Computing on GPUs

Lecture 1: CPUs and GPUs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford-Man Institute for Quantitative Finance

Oxford University Mathematical Institute

Economics

Money drives computing, as much as technology.

If there's a big enough market, someone will develop the product.

Need economies of scale to make chips cheaply, so very few companies and competing products.

To anticipate computing trends, look at market drivers, key applications.

CPUs

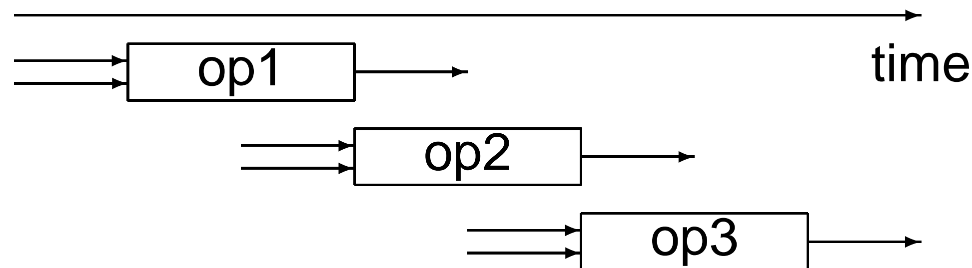
- chip size/speed continues to doubles every 18-24 months (Moore's Law)
- similar growth in other hardware aspects, but memory bandwidth struggles to keep up
- safe to assume that this will continue for at least the next 10 years, driven by:
  - multimedia applications (e.g. streaming video, HD)
  - image processing
  - "intelligent" software

Multilevel Parallelism

- instruction parallelism (e.g. addition)
- pipeline parallelism, overlapping different instructions
- multiple pipelines, each with own capabilities
- multiple cores (CPUs) within a single chip
- multiple chips within a single shared-memory computer
- multiple computers within a distributed-memory system
- multiple systems within an organisation

Ideal Von Neumann Processor

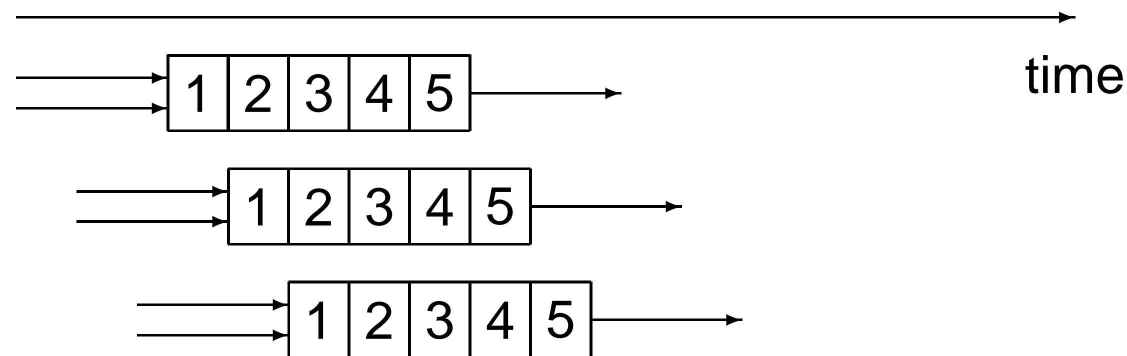
- each cycle, CPU takes data from registers, does an operation, and puts the result back
- load/store operations (memory $\longleftrightarrow$ registers) also take one cycle
- CPU can do different operations each cycle
- output of one operation can be input to next



CPU's haven't been this simple for a long time!

Pipelining

Pipelining is a technique in which multiple instructions are overlapped in execution.



- 1 result per cycle after pipeline fills up
- improved utilisation of hardware
- major complication – an output can only be used as input for an operation starting later

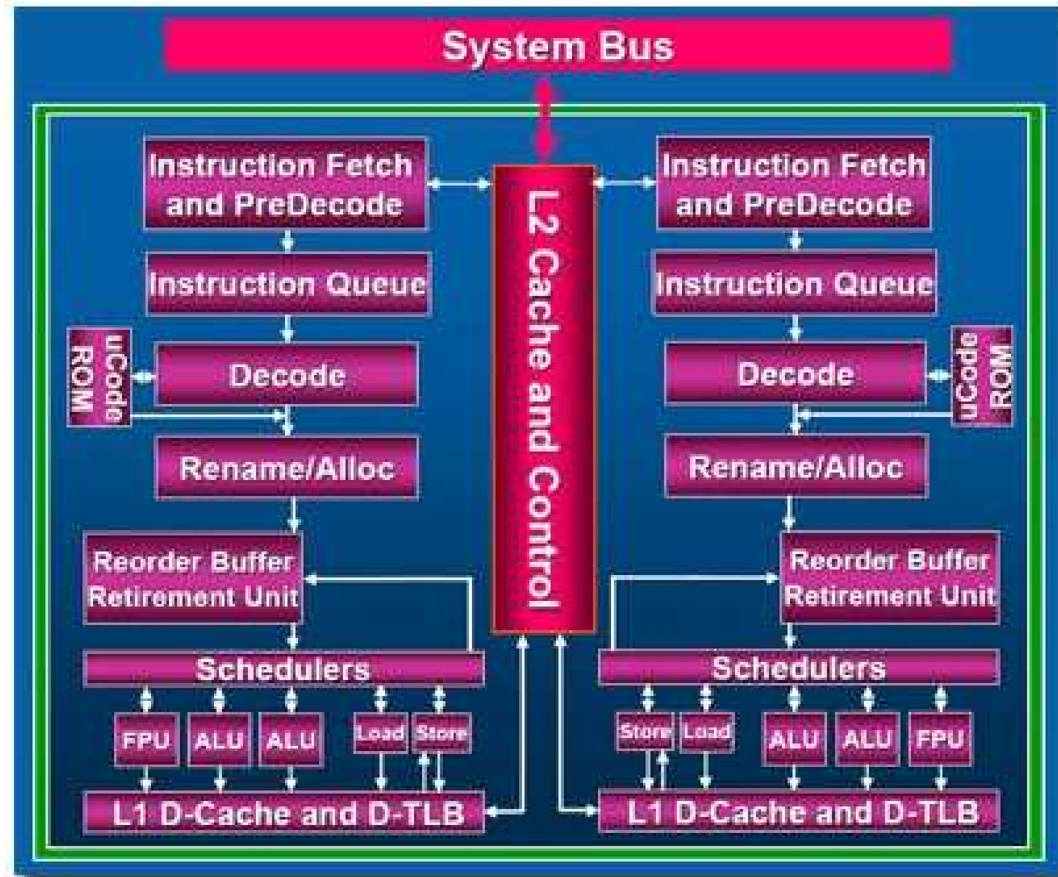
Superscalar Processors

Most processors have multiple pipelines for different tasks, and can start a number of different operations each cycle.

Example: each core in an Intel Core 2 Duo chip

- 14-stage pipeline
- 3 integer units (ALU)
- 1 floating-point addition unit (FPU)
- 1 floating-point multiplication unit (FPU)
- 2 load/store units
- in principle, capable of producing 3 integer and 2 FP results per cycle
- FP division is very slow

Intel Core Architecture



Technical Challenges

- compiler to extract best performance, reordering instructions if necessary
- out-of-order CPU execution to avoid delays waiting for read/write or earlier operations
- branch prediction to minimise delays due to conditional branching (loops, if-then-else)
- memory hierarchy to deliver data to registers fast enough to feed the processor

These all limit the number of pipelines that can be used, and increase the chip complexity;
90% of Intel chip devoted to control and data?

Current Trends

- clock cycle no longer reducing, due to problems with power consumption (up to 130W per chip)
- gates/chip still doubling every 24 months
  - $\Rightarrow$ more on-chip memory and MMU (memory management units)
  - $\Rightarrow$ specialised hardware (e.g. multimedia, encryption)
  - $\Rightarrow$ multi-core (multiple CPU's on one chip)
- peak performance of chip still doubling every 12-18 months

Intel chips

Core 2 Quad:

- four cores, up to 2.4GHz
- dynamic power-down of unused cores

Core i7 (Nehalem):

- four cores, up to 3.06GHz in standard models
- each core can run 2 threads
- integrated memory-management unit (QuickPath Interconnect)

Future:

- “Westmere” – 6-10 cores in 2010
- 4-way SMP system (40 cores, 80 threads)

Intel chips

All current chips support SSE vector instructions

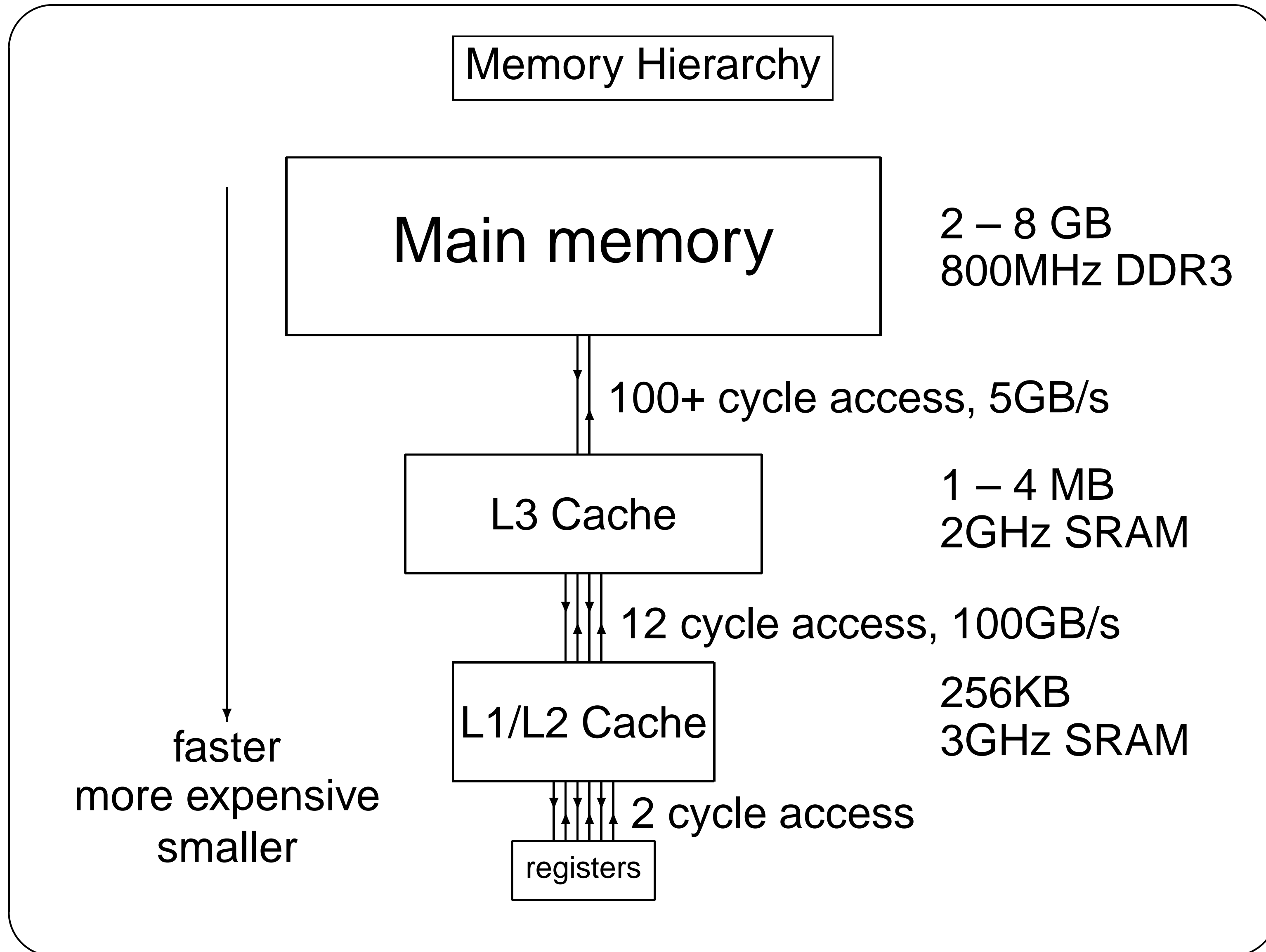
- added for graphics on systems without a GPU
- mini-vector of 4 `floats` or 2 `doubles`

The next generation will support AVX vector instructions:

- mini-vector of 8 `floats` or 4 `doubles`

and later products may increase the vector length.

Code vectorisation will become essential for good performance


Memory Hierarchy
Main memory
2 – 8 GB
800MHz DDR3
100+ cycle access, 5GB/s
L3 Cache
1 – 4 MB
2GHz SRAM
12 cycle access, 100GB/s
L1/L2 Cache
256KB
3GHz SRAM
2 cycle access
registers
faster
more expensive
smaller

Memory Hierarchy

Execution speed relies on exploiting data *locality*

- temporal locality: a data item just accessed is likely to be used again in the near future, so keep it in the cache
- spatial locality: neighbouring data is also likely to be used soon, so load them into the cache at the same time using a 'wide' bus (like a multi-lane motorway)
- from a programmer point of view, all handled automatically – good for simplicity but maybe not for best performance

GPUs – the big development

Economics is again the key:

- produced in vast numbers for computer graphics
- increasingly being used for
  - computer games "physics"
  - video (e.g. HD video decoding)
  - audio (e.g. MP3 encoding)
  - multimedia (e.g. Adobe software)

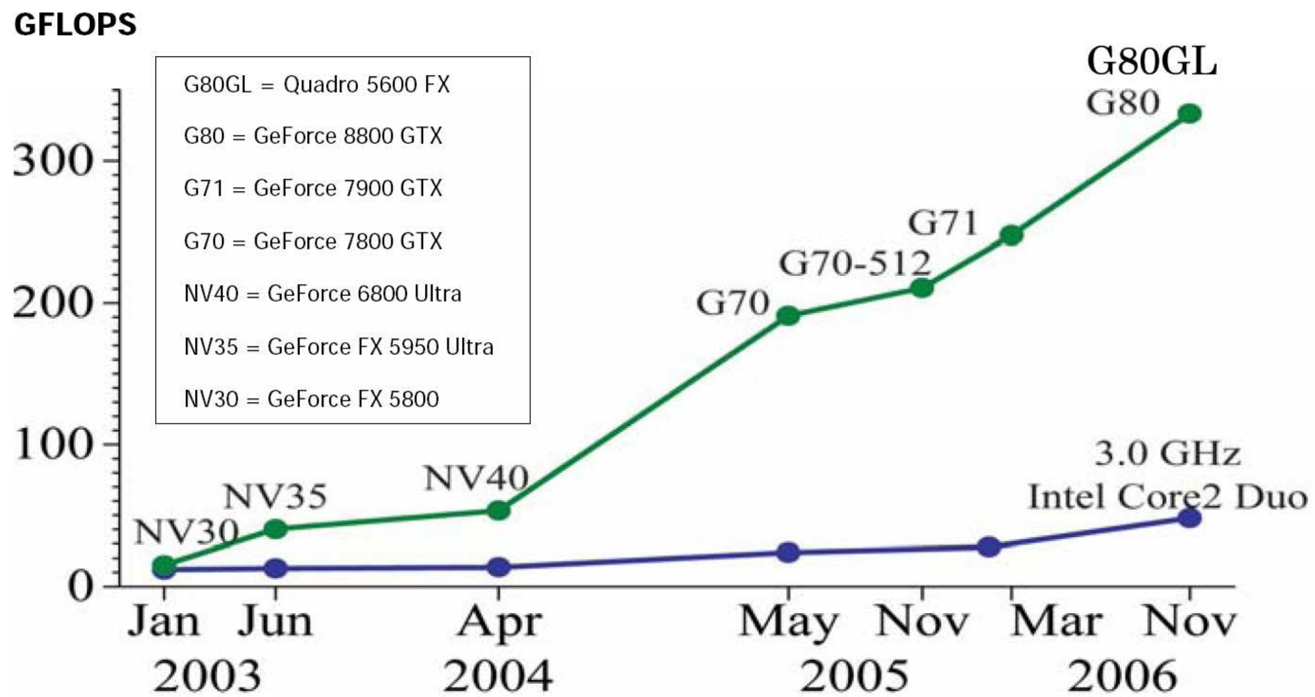
  - computational finance
  - oil and gas
  - medical imaging
  - computational science

GPUs – the big development

4 major vendors:

- NVIDIA
  - turnover about 10% of Intel's
- AMD
  - bought ATI several years ago
- IBM
  - co-developed Cell processor with Sony and Toshiba for Sony Playstation
- Intel
  - developing "Larrabee" GPU, due to ship in early 2010

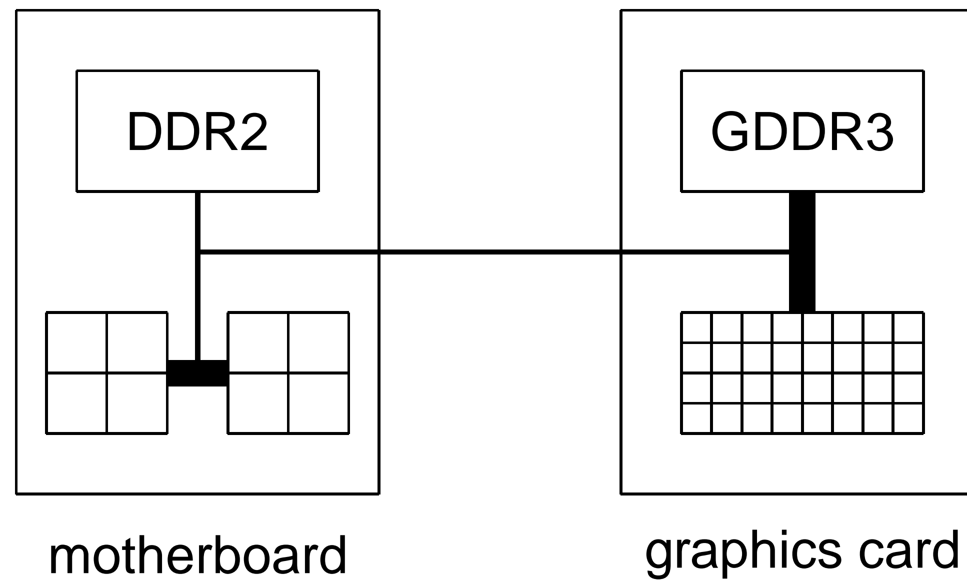
CPUs and GPUs



Copyright NVIDIA 2006/7

GPUs

The GPU sits on a PCIe graphics card inside a standard PC/server with one or two multicore CPUs:



GPUs

- up to 240 cores on a single chip
- simplified logic (minimal caching, no out-of-order execution, no branch prediction) means most of the chip is devoted to floating-point computation
- usually arranged as multiple units with each unit being effectively a vector unit
- very high bandwidth (up to 140GB/s) to graphics memory (up to 4GB)
- not general purpose – for parallel applications like graphics, and Monte Carlo simulations

GPUs

Can also build big clusters out of GPUs:

- NVIDIA S1070 1U server holds four GPUs which can be connected to a 1U twin-node Intel server
- Supermicro produces a 1U server which is a dual-CPU single node plus two GPUs
- similar products coming from other vendors
- cluster management software still evolving to support such servers

High-end HPC

- RoadRunner system at Los Alamos in US
  - first Petaflop supercomputer
  - IBM system based on Cell processors
- TSUBAME system at Tokyo Institute of Technology
  - 170 NVIDIA Tesla servers, each with 4 GPUs
- GENCI / CEA in France
  - Bull system with 48 NVIDIA Tesla servers
- within UK
  - Cambridge is getting a cluster with 32 Teslas
  - other universities are getting smaller clusters

GPUs in computational finance

- BNP Paribas has announced production use of a small cluster
  - 8 NVIDIA GPUs, each with 240 cores
  - replacing 250 dual-core CPUs
  - factor 10x savings in power (2kW vs. 25kW)
- lots of other banks doing proof-of-concept studies
  - I think IT groups are keen, but quants are concerned about effort involved
- I'm working with NAG to provide a random number generation library to simplify the task

GPUs in computational finance

Several ISV's now offer software based on NVIDIA's CUDA development environment:

- SciComp
- Quant Catalyst
- UnRisk
- Hanweck Associates
- Level 3 Finance
- others listed on NVIDIA CUDA website

Many of these are small, but it indicates the rapid take-up of this new technology

Chip Comparison

Intel Core 2 / Xeon / i7

- 4 MIMD cores
- few registers, multilevel caches
- 5-10 GB/s bandwidth to main memory

NVIDIA GTX280:

- 240 cores, arranged as 30 units each with 8 SIMD cores
- lots of registers, almost no cache
- 5 GB/s bandwidth to host processor (PCIe x16 gen 2)
- 140 GB/s bandwidth to graphics memory

Chip Comparison

MIMD (Multiple Instruction / Multiple Data)

- each core operates independently
- each can be working with a different code, performing different operations with entirely different data

SIMD (Single Instruction / Multiple Data)

- all cores executing the same instruction at the same time, but working on different data
- only one instruction de-coder needed to control all cores
- functions like a vector unit

Chip Comparison

One issue with SIMD / vector execution: what happens with if-then-else branches?

Standard vector treatment: execute both branches but only store the results for the correct branch.

NVIDIA treatment: works with sub-vectors (called warps) of length 32. If a particular warp only goes one way, then that is all that is executed.

Chip Comparison

How do NVIDIA GPUs deal with same architectural challenges as CPUs?

Very heavily multi-threaded, with at least 8 threads per core, and I often use 40 or more:

- solves pipeline problem, because output of one calculation can be used an input for next calculation *for same thread*
- switch from one set of threads to another when waiting for data from graphics memory

GPU Programming

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

- initially driven by needs of computer games developers
- now being driven by new markets (e.g. HD video decoding)
- C plus some extensions and some C++ features (e.g. templates)

GPU Programming

- 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

GPU Programming

Next major step is development of OpenCL standard

- pushed strongly by Apple, which now has NVIDIA GPUs in its entire product range, but doesn't want to be tied to them forever
- drivers are computer games physics, MP3 encoding, HD video decoding and other multimedia applications
- multi-platform standard will encourage 3rd party developers (including ISVs and banks)

GPU Programming

- based on CUDA and supported by NVIDIA, AMD, Intel, IBM and others, so developers can write their code once for all platforms
- Imagination Technologies (which developed chip for iPhone) is also assembling an OpenCL compiler team
- Microsoft is the one big name not involved
- first OpenCL compilers likely later this year
- will need to re-compile on each new platform, and also re-optimise the code
  – auto-tuning is another of the big trends in scientific computing

My experience

- Random number generation (mrg32k3a/Normal):
  - 2000M values/sec on GTX 280
  - 70M values/sec on Xeon using Intel's VSL library
- LIBOR Monte Carlo testcase:
  - 180x speedup on GTX 280 compared to single thread on Xeon

My experience

- 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 up to 4 times slower, probably factor 2 in future.

Why GPUs will stay ahead

Technical reasons:

- SIMD cores (instead of MIMD cores) means larger proportion of chip devoted to floating point computation
- tightly-coupled fast graphics memory means much higher bandwidth

Why GPUs will stay ahead

Economic 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

Will GPUs have impact in finance?

- I think they're the most exciting development in last 10 years
- Have generated a lot of interest/excitement in academia, being used by application scientists,not just computer scientists
- Gives at least $10\times$ improvement in energy efficiency and price / performance compared to $2\times$ quad-core Intel Xeons.
- Effectively a personal cluster in a PC under your desk

Will GPUs have an impact in finance?

What's needed to break through in finance?

- random number generation library from NAG (and others?)
- even more financial example codes
- work on tools and libraries to simplify development effort – especially for banks where quants have enough to do without having to think about CUDA programming
- training to educate potential users

More information

Wikipedia overviews of GeForce cards:

en.wikipedia.org/wiki/GeForce_9_Series

en.wikipedia.org/wiki/GeForce_200_Series

NVIDIA's CUDA homepage:

www.nvidia.com/object/cuda_home.html

Microprocessor Report article:

www.nvidia.com/docs/IO/47906/220401_Reprint.pdf

My webpages:

www.maths.ox.ac.uk/~gilesm/

www.maths.ox.ac.uk/~gilesm/hpc.html

Computational Finance on GPUs

Lecture 2: CUDA programming
and Monte Carlo applications

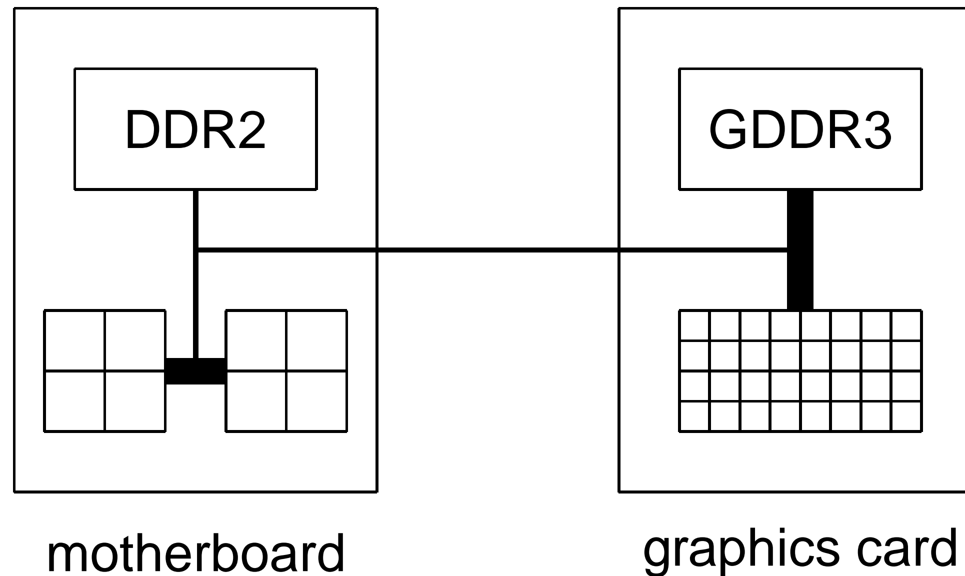
Mike Giles

mike.giles@maths.ox.ac.uk

Oxford-Man Institute for Quantitative Finance
Oxford University Mathematical Institute

Hardware view

At the top-level, there is a PCIe graphics card with a many-core GPU sitting inside a standard PC/server with one or two multicore CPUs:



Hardware view

At the GPU level:

- basic building block is a “multiprocessor” with
  - 8 cores
  - 16384 registers (on newest chips)
  - 16KB of shared memory
  - 8KB cache for constants held in graphics memory
  - 8KB cache for textures held in graphics memory

Hardware view

- different GPUs have different numbers of these:

product	multiprocs	bandwidth	cost
9800 GT	14	58GB/s	100 €
GTX 260	27	112GB/s	200 €
GTX 295	2×30	2×112GB/s	400 €

Hardware view

Key hardware feature is that the 8 cores in a multiprocessor are SIMD (Single Instruction Multiple Data) cores:

- all cores execute the same instructions simultaneously, but with different data
- similar to vector computing on CRAY supercomputers
- minimum of 4 threads per core, so end up with a minimum of 32 threads all doing the same thing at (almost) the same time
- natural for graphics processing and much scientific computing

Software view

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 2-4 as needed
7. de-allocates all memory and terminates

Software view

At a lower level, within the GPU:

- each copy of the execution kernel executes on one of the “multiprocessors”
- if the number of copies exceeds the number of multiprocessors, then more than one will run at a time on each multiprocessor if there are enough registers and shared memory
- other copies will wait in a queue and execute later

Software view

- 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 multiprocessor)
- there are no guarantees on the order in which the copies will execute

CUDA programming

CUDA is NVIDIA's program development environment:

- based on C with some extensions
- SDK has lots of example code and good documentation – 2-4 week learning curve for those with experience of OpenMP and MPI programming
- software freely available
- well integrated into Visual Studio for Windows users
- large user community active on NVIDIA forums

CUDA programming

At the host code level, there are library routines for:

- memory allocation on graphics card
- data transfer to/from graphics memory
  - constants
  - texture arrays (useful for lookup tables)
  - ordinary data
- error-checking
- timing

There is also a special syntax for launching multiple copies of the kernel process on the GPU.

CUDA programming

In its simplest form it looks like:

```
routine <<<gridDim,blockDim>>>(args);
```

- `gridDim` is the number of copies of the kernel (the "grid" size)
- `blockDim` is the number of threads within each copy (the "block" size)
- `args` is a limited number of arguments, usually mainly pointers to arrays in graphics memory

More general form allows `gridDim, blockDim` to be 2D or 3D to simplify application programs

CUDA programming

At the lower level, when one copy of the kernel is started on a multiprocessor it is executed by a number of threads, each of which knows about:

- some variables passed as arguments
- pointers to arrays in device memory (also arguments)
- constants held in device memory
- uninitialised shared memory
- local variables in private registers

CUDA programming

- some special variables:
  - `gridDim`
    size (or dimensions) of grid of blocks
  - `blockIdx`
    index (or 2D/3D indices) of block
  - `blockDim`
    size (or dimensions) of each block
  - `threadIdx`
    index (or 2D/3D indices) of thread

CUDA programming

The kernel code involves operations such as:

- read from/write to arrays in device memory
- write to/read from arrays in shared memory
- read constants
- use a texture array for a lookup table
- perform integer and floating point operations on data held in registers

The reading and writing is done implicitly – data is automatically read into a register when needed for an operation, and is automatically written when there is an assignment.

CUDA programming

First example: **prac4_1.cu**

Has elements above plus:

- memory allocation

```
cudaMalloc((void **)&d_x,
                     nbytes);
```

- data copying

```
cudaMemcpy(h_x,d_x,nbytes,
                 cudaMemcpyDeviceToHost);
```

Notice how kernel routine is declared by `__global__` prefix, and is written from point of view of a single thread.

CUDA programming

Second example: **prac4_2.cu**

Very similar to first, but using CUDA SDK toolkit for various safety checks – gives useful feedback in the event of errors.

- check for error return codes:

  ```
  cutilSafeCall( ...  );
  ```

- check for failure messages:

  ```
  cutilCheckMsg( ...  );
  ```

Third example: **prac4_3.cu**

A very simple Monte Carlo example:

- two asset, geometric Brownian motion
- European digital option
- `pathcalc` does the path calculation and evaluates the payoff
- main host code copies the results back and averages the payoff

Third example: **prac4_3.cu**

New CUDA bits:

- declaration of constant data for kernels

```
__constant__ int N;
```

- initialisation of constant data by host code

```
cudaMemcpyToSymbol("N",&N,
                        sizeof(N));
```

  first `"N"` refers to constant data on GPU;
  second `&N` referes to value in main code

CUDA programming

Third example: **prac4_3.cu**

Also demonstrates use of random number generation library developed with NAG:

- `gpu_mrg32k3a_init(V1, V2, 0);`
  initialises seeds (without an offset)
- ```
  gpu_mrg32k3a_normal(NPATH/64, 64,
                                2*N, d_z);
  ```
  each block has 64 threads, and each thread generates $2N$ Normals for a single path on the GPU and puts them into the graphics memory
  ```

CUDA programming

Final example: Monte Carlo LIBOR application

- timings in seconds for 96,000 paths, with 1500 blocks each with 64 threads, so one thread per path
- executed 5 blocks at a time on each multiprocessor, so 40 active threads per core
- CUDA results are for single precision

	time
original code (VS C++)	26.9
CUDA code (8800GTX)	0.2

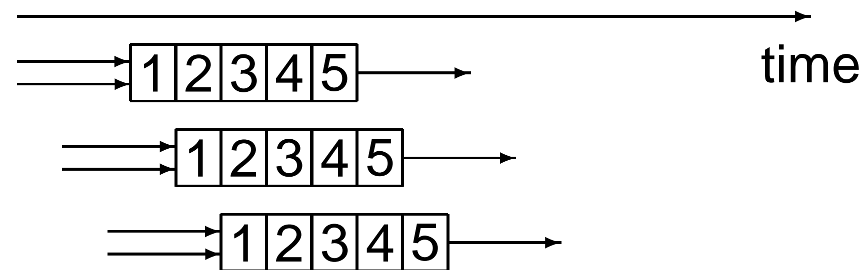
NVIDIA multithreading

Lots of active threads is the key to high performance:

- no “context switching”; each thread has its own registers, which limits the number of active threads
- threads execute in “warps” of 32 threads per multiprocessor (4 per core) – execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

NVIDIA multithreading

- for each thread, one operation completes long before the next starts – avoids the complexity of pipeline overlaps which can limit the performance of modern processors



- memory access from device memory has a delay of 400-600 cycles; with 40 threads this is equivalent to 10-15 operations and can be managed by the compiler

Random number generation

Main challenge with Monte Carlo 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'm developing 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 output is $(x_{n,1} - x_{n,2} \mod m_1) \,/\, m_1$ where integers $x_{n,1}$, $x_{n,2}$ are defined by

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

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

Random number generation

- Both recurrences are of the form

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

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

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

where A_k is defined by repeated squaring as

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

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

Random number generation

- **mrg32k3a** speed-up is $100\times$ on 216-core GTX260 compared to a single Athlon core
- have also implemented a **Sobol** generator to produce quasi-random numbers
- 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

Computational Finance on GPUs

Lecture 3: more CUDA programming
and finite difference application

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford-Man Institute for Quantitative Finance
Oxford University Mathematical Institute

CUDA programming

Coalesced memory transfers are one of the trickiest aspects of CUDA programing

The bandwidth from graphics memory to GPU is 100+GB/s, but only if data is accessed correctly

Key point: when reading (or writing), 32 threads of a warp should address a block of 32 elements of an array in graphics memory

(Minor point: the starting point should be a multiple of 32 from the beginning of the array)

CUDA programming

Example: **prac4_3.cu**

```
__global__ void pathcalc(float *d_z,
                         float *d_v)
d_z = d_z + threadIdx.x
      + 2*N*blockIdx.x*blockDim.x;
d_v = d_v + threadIdx.x
      + blockIdx.x*blockDim.x;
```

- starting addresses for the threads in each block are sequential.
- offset for each block corresponds to total data for each block

Example: **prac4_3.cu**

After each reference to random number array, shift pointer by number of threads in the block to next element for that thread:

```
y1   = (*d_z);
d_z += blockDim.x;
y2   = rho*y1 + alpha*(*d_z);
d_z += blockDim.x;
```

CUDA programming

We haven't yet discussed *shared memory*.

Each multiprocessor has 16kB of shared memory which can be accessed by any thread in the block

Very useful whenever a thread needs to share data with another thread in same block – no sharing with threads in other blocks

It's declared in a device routine like this:

```
__shared__ float u[100];
```

then used in the usual way

CUDA programming

There is also read-only *texture memory* which has a local cache.

I won't give any examples of this, but it is useful for random-access lookup tables (e.g. when computing a local volatility $\sigma(S,t)$ defined by a spline function)

Finally, there is *local memory*, which is poorly named – it's really part of the graphics memory which holds local variables when there are too many to go in registers

Finite Difference Model Problem

The Black-Scholes PDE for a two-asset model problem is

$$V_t + rS_1 V_{S_1} + rS_2 V_{S_2} +$$
$$\sigma^2 \left(\frac{1}{2} S_1^2 V_{S_1 S_1} + \rho S_1 S_2 V_{S_1 S_2} + \frac{1}{2} S_2^2 V_{S_2 S_2} \right) = rV$$

This is solved backwards in time from the final value equal to the payoff function, to get the value at the initial time $t = 0$.

Finite Difference Model Problem

Switching to new variables $\eta = \log S$, $\tau = 1 - t$, and defining

$$r^* = r - \tfrac{1}{2}\sigma^2,$$

the equation becomes

$$\begin{aligned} V_\tau &= r^* \left(V_{\eta_1} + V_{\eta_2}\right) \\ &\quad + \sigma^2 \left(\tfrac{1}{2} V_{\eta_1\eta_1} + \rho V_{\eta_1\eta_2} + \tfrac{1}{2} V_{\eta_2\eta_2}\right) - rV \end{aligned}$$

which is solved forward in time from $\tau = 0$ to $\tau = 1$.

Finite Difference Model Problem

A simple Explicit Euler central space discretisation on a uniform Cartesian grid is

$$V^{n+1} = (1 - r\Delta t)V^n + \frac{r^*\Delta t}{2\Delta\eta}\left(\delta_{2\eta_1} + \delta_{2\eta_2}\right)V^n$$

$$+\frac{\sigma^2\Delta t}{2\Delta\eta^2}\left((1-\rho)\,\delta^2_{\eta_1} + \rho\,\delta^2_{\eta_1\eta_2} + (1-\rho)\,\delta^2_{\eta_2}\right)V^n$$

where

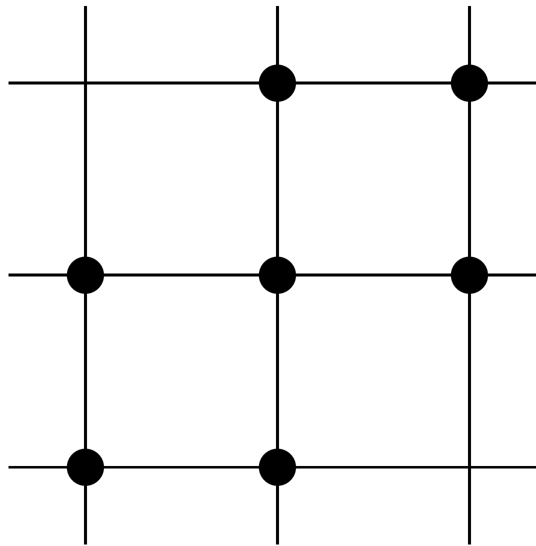
$$\begin{aligned} \delta_{2\eta_1}V_{i,j} &\equiv V_{i+1,j} - V_{i-1,j} \\ \delta_{2\eta_2}V_{i,j} &\equiv V_{i,j+1} - V_{i,j-1} \end{aligned}$$

and ...

Finite Difference Model Problem

$$
\begin{aligned}
\delta^2_{\eta_1} V_{i,j} &\equiv V_{i+1,j} - 2V_{i,j} + V_{i-1,j} \\
\delta^2_{\eta_1\eta_2} V_{i,j} &\equiv V_{i+1,j+1} - 2V_{i,j} + V_{i-1,j-1} \\
\delta^2_{\eta_2} V_{i,j} &\equiv V_{i,j+1} - 2V_{i,j} + V_{i,j-1}
\end{aligned}
$$

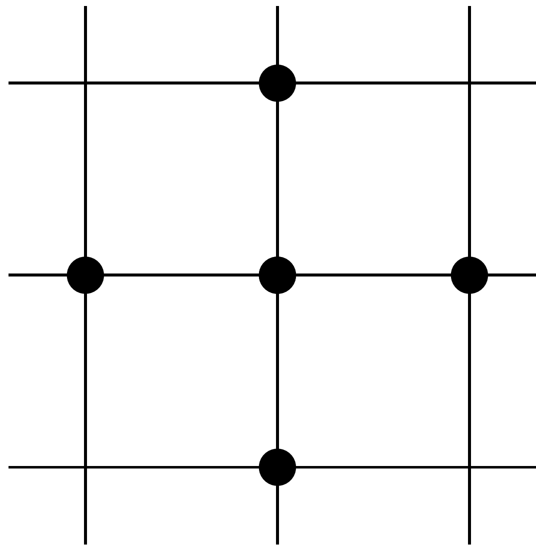
making it a 7-point stencil:



Finite Difference Model Problem

Even simpler 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?

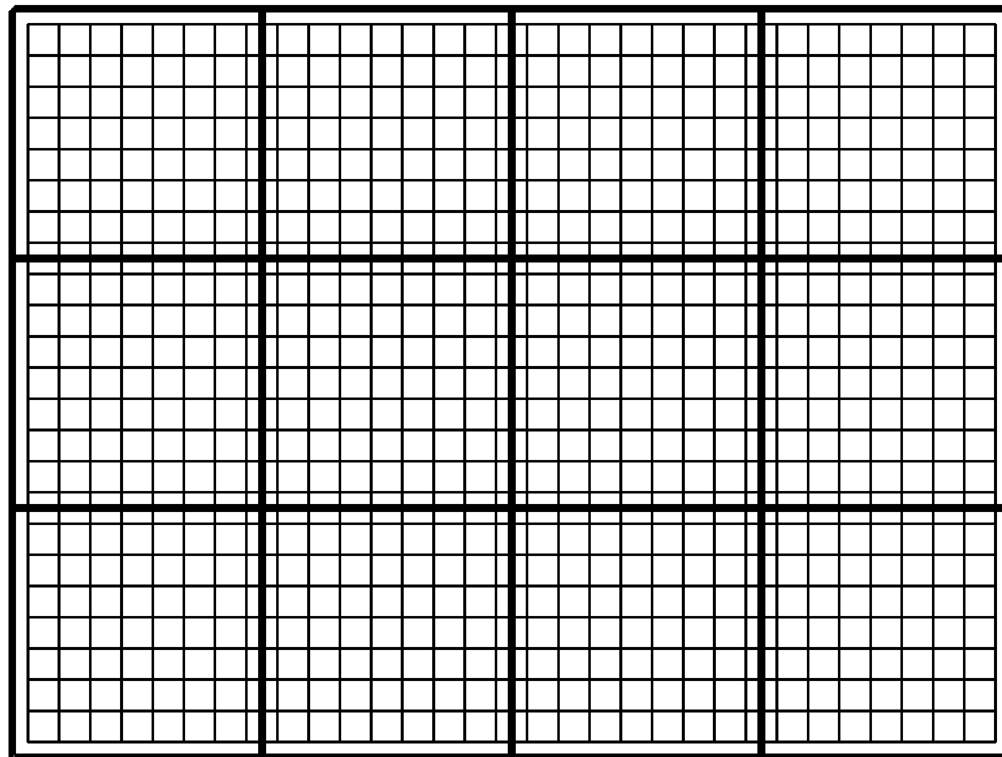
First idea: each thread does one grid point, reading in directly from graphics memory the old values at the 4 neighbours (6 in 3D).

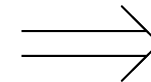
Performance would be awful:

- 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)

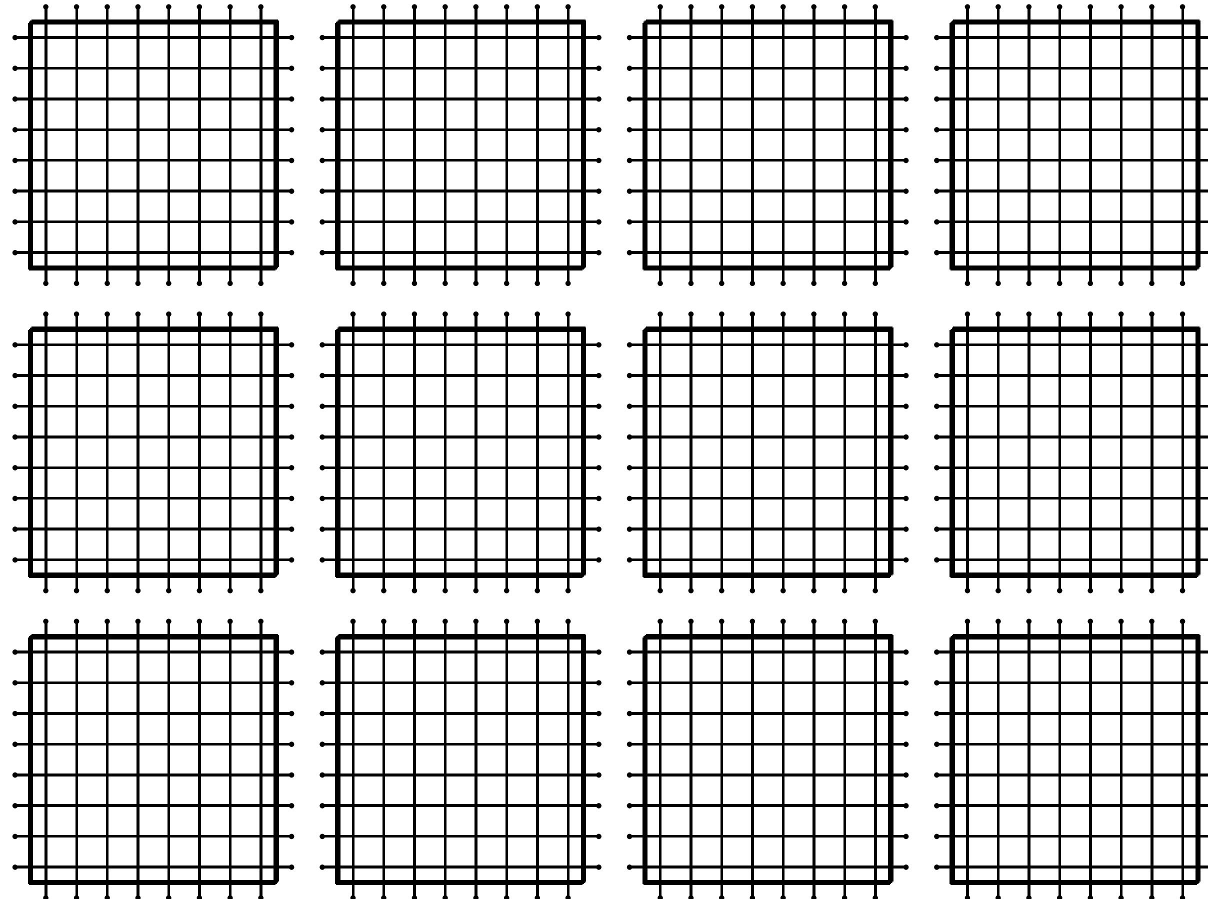
Finite Difference Model Problem

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

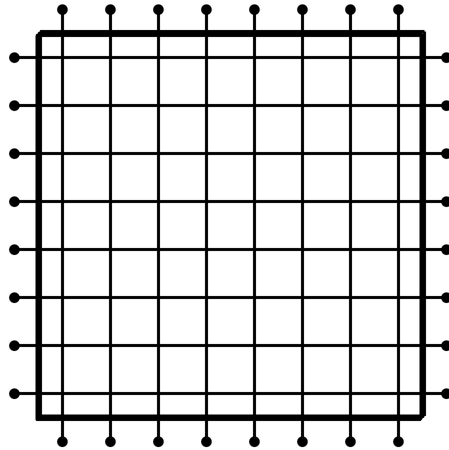




Finite Difference Model Problem



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

Finite Difference Model Problem

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

Finite Difference Model Problem

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

Finite Difference Model Problem

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

Final Words

- GPUs offer $10\times$ improvements in energy efficiency and price / performance compared to standard CPUs
- biggest development in HPC for 10 years, will be important for next 10 years
- (also watch increase in SSE vector length within standard CPUs)
- Monte Carlo applications are fairly straightforward, given a random number generator
- PDE applications are possible but require more effort