GPUs – the next big advance in HPC?

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford-Man Institute for Quantative Finance

Oxford eResearch Centre

Acknowledgments: computations by Xiaoke Su and Gerd Heber

and research funding from Microsoft and EPSRC

Overview

- trends in mainstream HPC
- the co-processor alternatives
- NVIDIA graphics cards
  - hardware
  - software
  - applications
- will they really have an impact?

Computing – Recent Past

- driven by the cost benefits of massive economies of scale, specialised chips (e.g. CRAY vector chips) died out, leaving Intel/AMD dominant
- Intel/AMD chips designed for office/domestic use, not for high performance computing
- increased speed through higher clock frequencies, and complex parallelism within each CPU
- PC clusters provided the high-end compute power, initially in universities and then in industry

- at same time, NVIDIA and ATI grew big on graphics chip sales driven by computer games

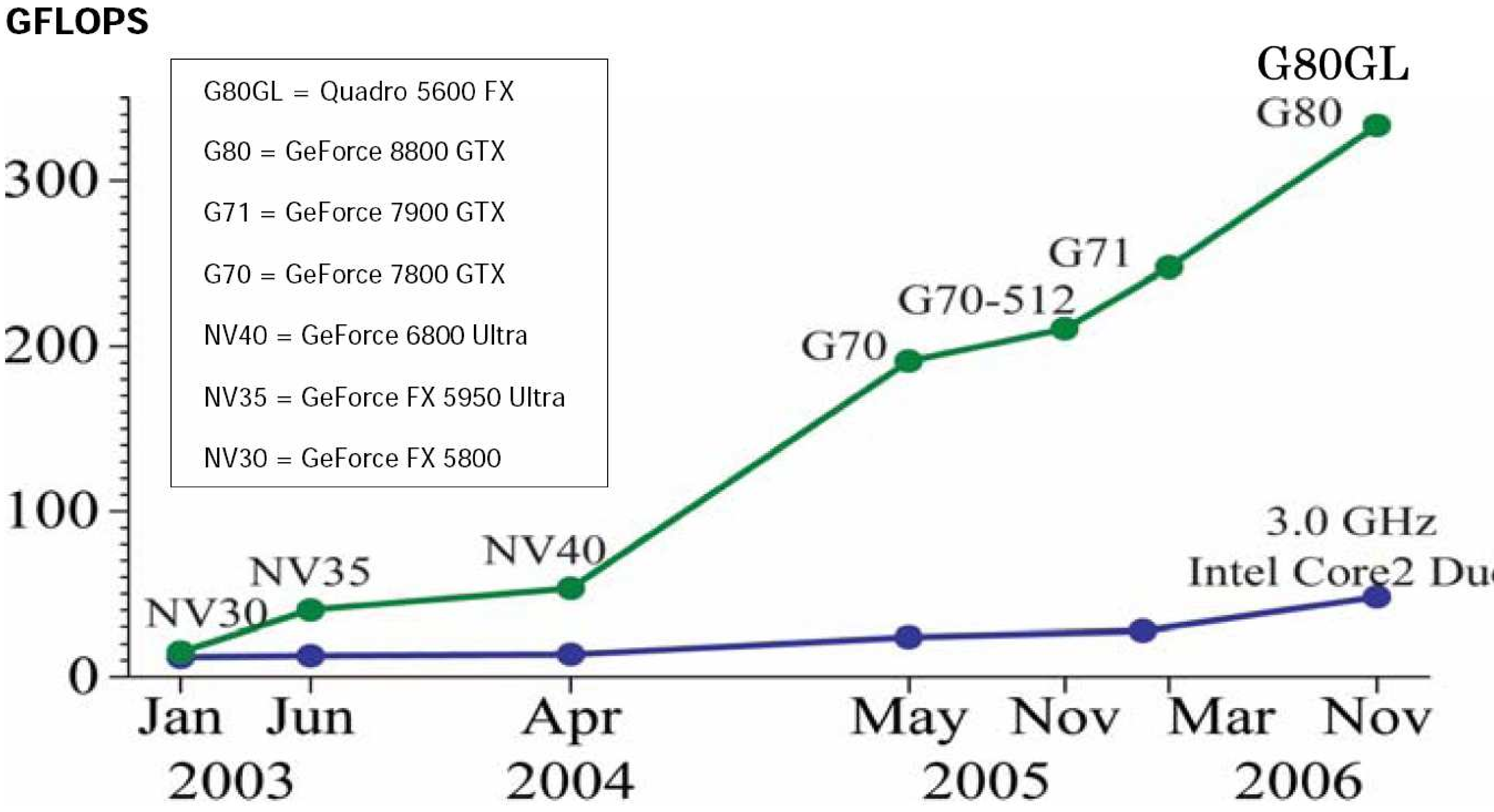
Computing – Present/Future

- move to faster clock frequencies stopped due to high power consumption
- big push now is to multicore (multiple processing units within a single chip) at (slightly) reduced clock frequencies

- graphics chips have even more cores (up to 128) – big new development here is a more general purpose programming environment

  Why? At least partly because computer games do increasing amounts of “physics” simulation

CPUs and GPUs



Copyright NVIDIA 2006/7

Mainstream CPUs

- currently up to 4 cores – 16 cores likely within 5 years?
- intended for general applications
- MIMD (Multiple Instruction / Multiple Data)
  – each core works independently of the others, executing different instructions, often for different processes
- specialised vector capabilities (SSE2/SSE3) for vectors of length 4 (s.p.) or 2 (d.p.) – motivated by graphics requirements but rarely used for scientific applications?

Mainstream CPUs

How does one exploit all of these cores?

- OpenMP multithreading for shared-memory parallelism
  - easy to get parallel code running
  - can be harder to get good parallel performance
  - degree of difficulty: 2/10
- MPI message-passing for distributed-memory parallelism
  - hard to get started, need to partition data and programming is low-level and tedious
  - generally easier to get good parallel performance
  - degree of difficulty: 6/10

Mainstream CPUs

Importance of standards:

- makes it possible to write portable code to run on any hardware
- encourages developers to work on code optimisation
- encourages academic/commercial development of tools and libraries to assist application developers

Co-processor alternatives

Graphics chips:

- Cell processor, developed by IBM/Sony/Toshiba for Sony Playstation 3
- NVIDIA GeForce 8 and 9 series GPUs, developed primarily for high-end computer games market
- new AMD/ATI Firestream 9170 (still not shipping?)

Clearspeed card:

- PCI-X/PCIe card with 2 multicore chips, developed specifically for scientific computing applications

Chip Comparison

chip / type	cores	Gflops	cost	watts
MIMD				
Intel Xeon	2-4	10-20	400	80-100
SUN T2	8	25?	1000?	50-100?
IBM Cell	1+8	25-250(sp)	4000	85
SIMD				
Clearspeed	2×96	2×25	4000	25
NVIDIA 8800	112-128	250-500(sp)	140-400	100-200
FPGA				
Xilinx	N/A	50-500(sp)?	200-2000?	50-100?

Does single precision (sp) matter?

Chip Comparison

Intel Core 2 / Xeon:

- 2 or 4 MIMD cores
- few registers, multilevel caches
- 5-20 GB/s bandwidth to main memory
- double precision floating point arithmetic

NVIDIA 8800 GPUs:

- up to 128 SIMD cores
- lots of registers, no caches
- 60-100 GB/s bandwidth to graphics memory
- single precision floating point arithmetic

NVIDIA GeForce 8 and 9 series

- basic building block is a “multiprocessor” with 8 cores, 8192 registers and a small amount of shared memory
- different chips have different numbers of these:

product	multiprocessors	bandwidth	cost
9600 GT	8	58GB/s	£100
8800 GT	14	58GB/s	£140
8800 GTX	16	86GB/s	£250
8800 Ultra	16	104GB/s	£400

- each card has fast graphics memory which is used for:
  - global memory accessible by all multiprocessors
  - special read-only constant memory
  - additional local memory for each multiprocessor

NVIDIA GeForce 8 and 9 series

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

- all cores execute the same instructions simultaneously
- vector style of programming harks back to CRAY vector supercomputing
- natural for graphics processing and much scientific computing
- SIMD is also a natural choice for massively multicore to simplify each core
- requires specialised programming (no standard)

CUDA programming

CUDA is NVIDIA’s program development environment:

- based on C with some extensions
- lots of example code and good documentation
  – 2-4 week learning curve for those with experience of OpenMP and MPI programming
- growing user community active on NVIDIA forum
- main process runs on host system (Intel/AMD CPU) and launches multiple copies of “kernel” process on graphics card
- communication is through data transfers to/from graphics memory
- minimum of 4 threads per core, but more is better

CUDA programming

How hard is it to program?

Needs combination of skills:

- splitting the application between the multiple multiprocessors is similar to MPI programming,
  but no need to split data – it all resides in main graphics memory
- SIMD CUDA programming within each multiprocessor is a bit like OpenMP programming – needs good understanding of memory operation
- difficulty also depends a lot on application

CUDA programming

One option is to use linear algebra libraries to off-load parts of a calculation:

- libraries for BLAS, LAPACK and FFTs
- performance potentially restricted by limited 2GB/s bandwidth of PCIe-2 link between host and graphics card
- still, quick easy win for some applications (e.g. solving 10,000 simultaneous linear equations)
- spectral CFD testcase from Univ. of Washington gets $20\times$ speedup using MATLAB/CUDA interface
- degree of difficulty (2/10)

CUDA programming

Monte Carlo application:

- ideal because it is trivially parallel – each path calculation is independent of the others
- degree of difficulty (4/10)
- Xiaoke Su and I obtained excellent results for a financial application called a LIBOR interest rate model
- timings in seconds for 96,000 paths, with 40 active threads per core, each thread doing just one path
- remember: CUDA results are for single precision

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

Original LIBOR code

```
void path_calc(int N, int Nmat, double delta,
               double L[], double lambda[], double z[])
{
  int    i, n;
  double sqez, lam, con1, v, vrat;

  for(n=0; n<Nmat; n++) {
    sqez = sqrt(delta)*z[n];
    v = 0.0;
    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += (con1*L[i])/(1.0+delta*L[i]);
      vrat = exp(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
  }
}
```

CUDA LIBOR code

```
__constant__ int    N, Nmat, Nopt, maturities[NOPT];
__constant__ float  delta, swaprates[NOPT], lambda[NN];

__device__ void path_calc(float *L, float *z)
{
  int   i, n;
  float sqez, lam, con1, v, vrat;

  for(n=0; n<Nmat; n++) {
    sqez = sqrtf(delta)*z[n];
    v    = 0.0;
    for (i=n+1; i<N; i++) {
      lam  = lambda[i-n-1];
      con1 = delta*lam;
      v   += __fdividef(con1*L[i],1.0+delta*L[i]);
      vrat = __expf(con1*v + lam*(sqez-0.5*con1));
      L[i] = L[i]*vrat;
    }
  }
}
```

CUDA LIBOR code

The main code performs the following steps:

- initialises card
- allocates memory in host and on device
- copies constants from host to device memory
- launches multiple copies of execution kernel on device
- copies back results from device memory
- de-allocates memory and terminates

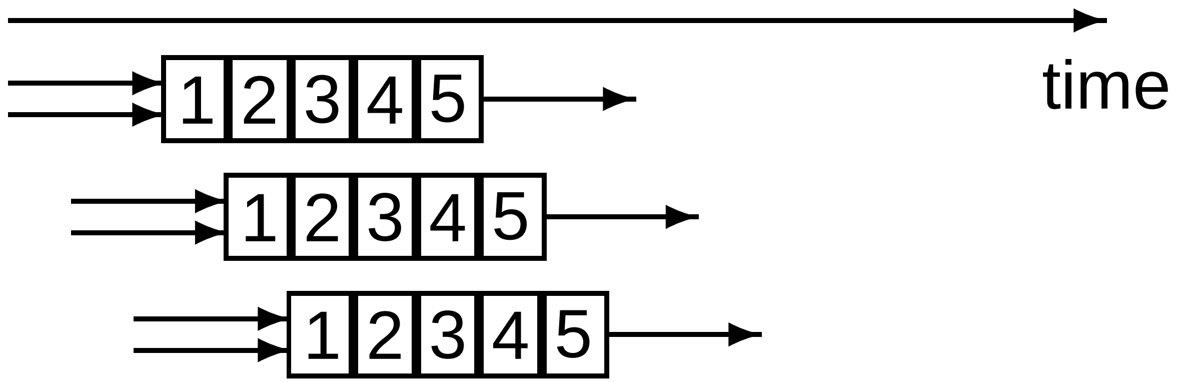
NVIDIA multithreading

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

CUDA programming

Finite difference application:

- recently started work on very simple 2D/3D finite difference applications – Jacobi iteration for solving discrete Laplace equation
- conceptually straightforward for someone who is used to partitioning grids for MPI implementations
  - each multiprocessor works on a block of the grid
  - threads within each block read data into local shared memory, do the calculations in parallel and write new data back to main device memory
- degree of difficulty: 6/10 (going up to 8/10 for implicit solver?)

CUDA programming

2D finite difference implementation:

- key steps in kernel code:
  - load in block of data from graphics memory, with halo of depth 2
  - perform 2 Jacobi iterations
  - store new data back into graphics memory
- $30\times$ speedup relative to Xeon single core, compared to $4.5\times$ speedup using OpenMP with 8 cores

The speedup is comparable to results obtained by Graham Pullan and Tobias Brandvik at Cambridge for a compressible flow CFD code.

CUDA programming

3D finite difference implementation:

- insufficient shared memory to hold whole 3D block, so hold 3 working planes at a time (halo depth of 1, just one Jacobi iteration 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 (over-writing $k-2$ plane)
    - process k z-plane
    - store new k z-plane
- $20\times$ speedup relative to Xeon single core – no OpenMP comparison yet

Will GPUs have real impact in HPC?

- I think they're the most exciting development since initial development of PVM and Beowulf clusters
- Have generated a lot of interest/excitement in academia, being used by application scientists, not just computer scientists
- Potential for $10-100\times$ speedup and improvement in GFLOPS/£ and GFLOPS/watt
- Effectively a personal cluster in a PC under your desk

Will GPUs have real impact in HPC?

What’s needed to make it mainstream?

- training to educate potential users
- even more example codes, relevant to different application areas
- work on tools and libraries to simplify development effort
- student projects to get more people exposed to the possibilities
- more conferences like this one to build the user community in the UK

Will GPUs have real impact in HPC?

What are we doing in Oxford?

- bought several PCs with different NVIDIA cards
- Oxford Supercomputing Centre is installing a 4-card Tesla 1U server (1.5GB per card)
- building community of local users to help newcomers get started
- starting to work on library development
- adding 3 lectures to my MSc course on Numerically Intensive Computing for Finance

New EPSRC Cluster

Funded by EPSRC for 1 year to promote use of FPGA and GPU technologies:

- extensive travel budget
- workshops:
  - use of FPGAs
  - use of GPUs
  - challenges in bio-informatics
- a number of pilot investigations (e.g. issues associated with single precision arithmetic)
- one key outcome is proposals for further research
- another is community building so we share our experience and expertise, and connect computing experts to application experts

Webpages

Wikipedia overviews of GeForce cards:
`en.wikipedia.org/wiki/GeForce_8_Series`
`en.wikipedia.org/wiki/GeForce_9_Series`

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

Microprocessor Report article:
`www.nvidia.com/docs/IO/47906/220401_Reprint.pdf`

Information on chips and parallel computing:
`www.comlab.ox.ac.uk/mike.giles/nicf.html`