

Financial computing on NVIDIA GPUs

Mike Giles

`mike.giles@maths.ox.ac.uk`

Oxford University Mathematical Institute
Oxford-Man Institute for Quantative Finance
Oxford eResearch Centre

Acknowledgments: Gerd Heber, Abinash Pati, Vignesh Sundaresh, Xiaoke Su
and funding from Microsoft, EPSRC, TCS/CRL

Overview

- trends in mainstream HPC
- the co-processor alternatives
- NVIDIA graphics cards
- CUDA programming
- LIBOR Monte Carlo application
- finite difference PDE applications

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

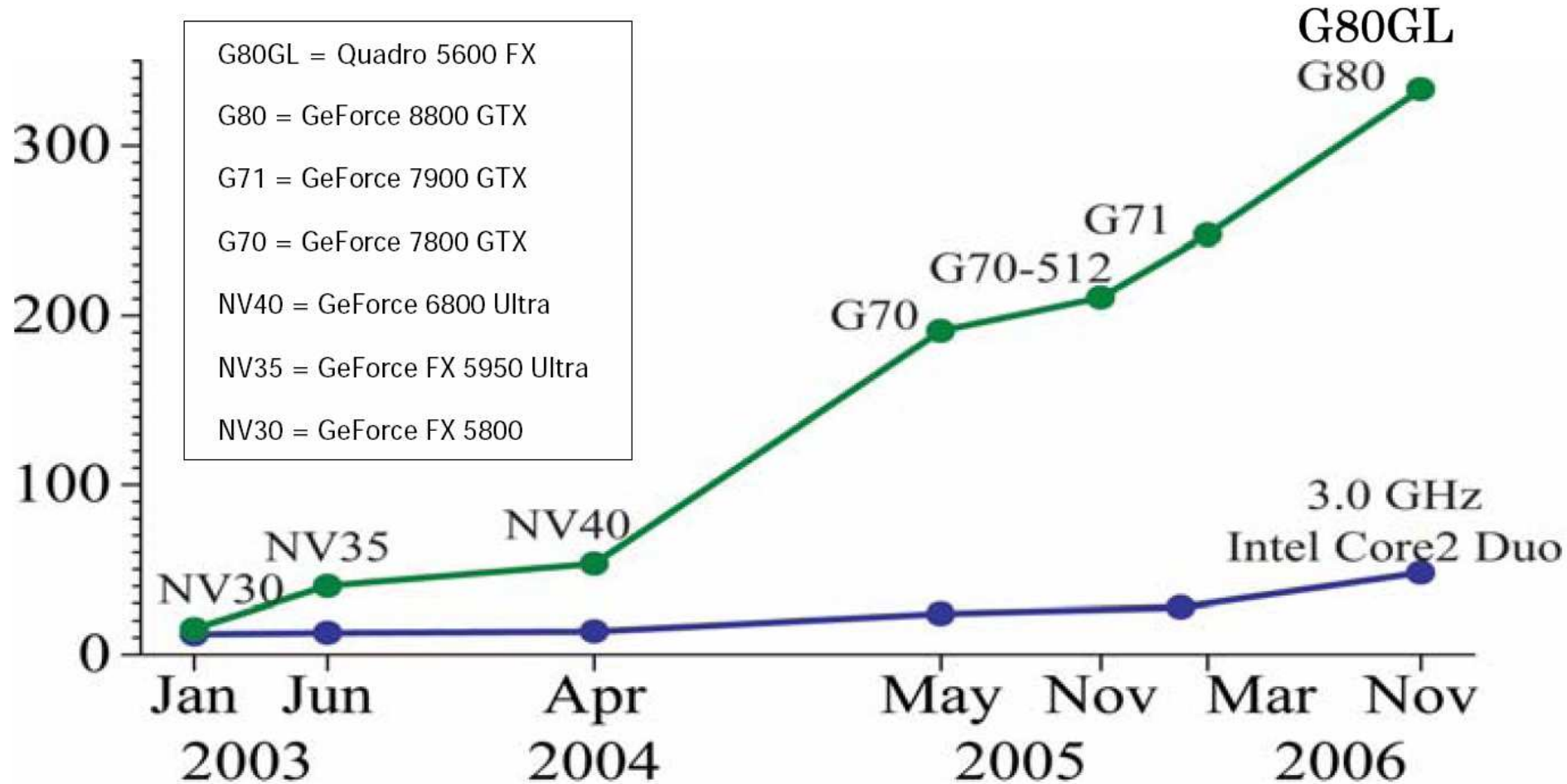
Computing – Present/Future

- move to faster clock frequencies stopped due to high power consumption (proportional to f^2 ?)
- 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 240 on NVIDIA GPUs)
 - 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

GFLOPS



Copyright NVIDIA 2006/7

Mainstream CPUs

- currently up to 6 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 sometimes 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

GPUs:

- 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
- AMD Firestream 9250
- Intel “Larrabee” GPU due in 2010

FPGAs:

- Xilinx
- Altera

Chip Comparison

chip / type	cores	Gflops	GB/s	watts	cost (£)
<u>MIMD</u>					
Intel Xeon	2-6	10-40	5-20	60-120	100-250
SUN T2	8	11	60	100	1000?
IBM Cell	1+8	250*, 100	25	100	2000?
<u>SIMD</u>					
NVIDIA	112-240	250-1000*, 125	60-140	100-250	100-400
AMD/ATI	similar				
<u>FPGA</u>					
Xilinx	N/A	50-500*?	5-20?	50-100?	200-2000?

Does single precision* matter?

Chip Comparison

Intel Core 2 / Xeon:

- 2-6 MIMD cores
- few registers, multilevel caches
- 5-20 GB/s bandwidth to main memory
- double precision floating point arithmetic

NVIDIA GPUs:

- up to 240 SIMD cores
- lots of registers, no caches
- 5 GB/s bandwidth to host processor (PCIe x16 gen 2)
- 60-140 GB/s bandwidth to graphics memory
- single precision floating point arithmetic

Why GPUs will stay ahead

Technical reasons:

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

Commercial reasons:

- CPUs driven by cost-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

NVIDIA GeForce 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
9800 GT	14	58GB/s	£100
9800 GTX	16	70GB/s	£140
9800 GX2	2×16	128GB/s	£280
GTX280	30	142GB/s	£350

- 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 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 restricted by 5GB/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× 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)
- we obtained excellent results for a LIBOR 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 (8800GTX)	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    += __fdivdef(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

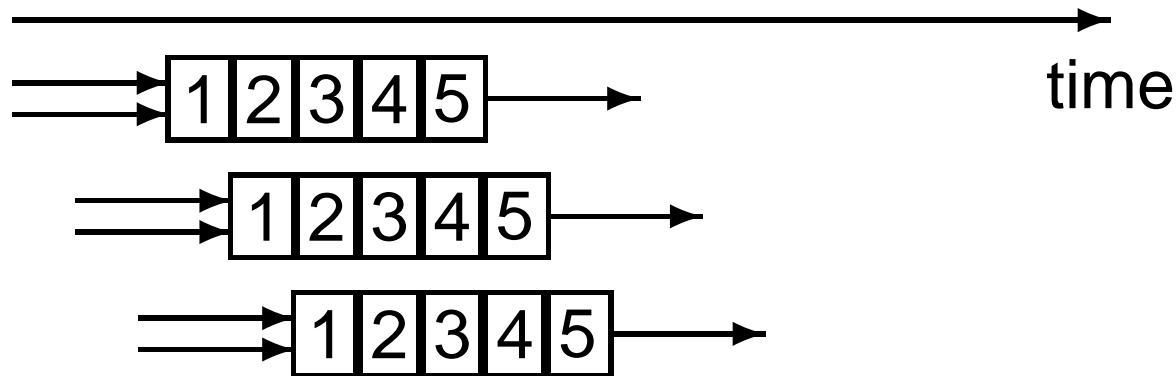
CUDA multithreading

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

CUDA 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

Other Monte Carlo considerations:

- need RNG routines
 - which ones?
 - skip-ahead for multiple threads?
- need to generate correlated streams (a bit tricky due to limited shared-memory in each 8-core multiprocessor)
- QMC much trickier because of memory requirements for BB or PCA construction
- working with NAG to develop a generic Monte Carlo engine

CUDA programming

Finite difference application:

- recently started work on 2D/3D finite difference applications
 - Jacobi iteration for discrete Laplace equation
 - CG iteration for discrete Laplace equation
 - ADI time-marching
- 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 for explicit solvers, 8/10 for ADI solver

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
- $50\times$ speedup relative to Xeon single core, compared to $5\times$ speedup using OpenMP with 8 cores.

CUDA programming

Development of PDE demo codes is being funded by TCS/CRL:

- TCS: Tata Consultancy Services – India's biggest IT services company
- CRL: Computational Research Laboratories – part of Tata group, with an HP supercomputer ranked at #4 in Top 500 six months ago (now #8)
- demo codes will be made freely available on my website
- trying to create generic 3D library/template to enable easy development of new applications
- looking for new test applications

Will GPUs have real impact?

- 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× speedup and improvement in GFLOPS/£ and GFLOPS/watt
- Effectively a personal cluster in a PC under your desk
- Needs work on tools and libraries to simplify development effort

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`

LIBOR test code:

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