Computational Finance using CUDA on NVIDIA GPUs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford-Man Institute for Quantitative Finance

Oxford eResearch Centre

Acknowledgments: Gerd Heber, Abinash Pati, Vignesh Sundaresh, Xiaoke Su, Chris Yau, Anthony Ng, and support from NVIDIA, Microsoft, TCS/CRL and EPSRC

Overview

- NVIDIA GPU's
- Monte Carlo methods
  - LIBOR model testcase
  - random number generation
  - particle filters
- finite difference methods
  - explicit time-marching
  - implicit time-marching

NVIDIA GPUs

- basic building block is a “multiprocessor” with 8 cores, up to 16384 registers, 16KB shared memory and 8KB caches for texture and constant data
- different chips have different numbers of these:

product	multiprocs	bandwidth	cost
9800 GT	14	58GB/s	$140
GTX 280	30	142GB/s	$350

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

NVIDIA GPUs

For high-end HPC, NVIDIA have Tesla systems:

- C1060 card:
  - PCIe card, plugs into standard PC/workstation
  - single GPU with 240 cores and 4GB graphics memory
- S1070 server:
  - 4 cards packaged in a 1U server
  - connect to 2 external servers, one for each pair of cards
  - each GPU has 240 cores plus 4GB graphics memory
- neither product has any graphics output, intended purely for scientific computing

NVIDIA GPUs

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
- minimum of 4 threads per core, so minimum vector length of 32 – I usually use at least 16 threads per core
- natural for graphics processing and much scientific computing
- natural for massively multicore to simplify each core

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 require high 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

Computational Finance

- biggest growth area in scientific computing, roughly 20% of Top 500 “supercomputers”
- biggest employer of Oxford mathematicians and theoretical physicists, often as “quants”
  - traders – make/lose the money
  - quants – develop the models and codes
  - IT – organise execution on large distributed systems, and connect to data and external world
- two main kinds of computations for options pricing
  - Monte Carlo (60% ?)
  - PDE / finite difference (30% ?)

Monte Carlo methods

Monte Carlo is a trivially parallel application:

- involves 10^4–10^6 independent "path" simulations with different random numbers
- usually just interested in average of a single output (the "payoff") to determine the option value
- it is compute-intensive, with a minimal amount of data for each path (level 1 cache on CPU?)
- ideally suited for GPU implementation, usually very little conditional branching so good for vectorisation

LIBOR testcase

- models behaviour of interest rates to compute prices for lots of products dependent on future interest rates
- lots of computation per random number – for testcase omitted random number generation
- testcase computes price sensitivities using adjoint approach – requires more data storage than usual
- timings in seconds for 96,000 paths, with 40 active threads per core on 128-core 8800GTX

	hardware	cores	time
original code	Intel Xeon	1	26.9
CUDA code	NVIDIA 8800GTX	128	0.2

LIBOR testcase

These CUDA results are for single precision – does it matter?

Compared to modelling, discretisation and Monte Carlo sampling errors, single precision perfectly sufficient provided:

- use binary tree summation when averaging payoffs (natural approach to vectorisation)
- avoid computing sensitivities by finite differencing:

$$\frac{\partial V}{\partial \theta} \approx \frac{V(\theta+\Delta\theta) - V(\theta-\Delta\theta)}{2\,\Delta\theta}$$

Irrelevant in the long-term as double precision becomes available at minimal cost.

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;
    }
  }
}
```

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

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

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

- **mrg32k3a** speed-up is only $6\times$ on 112-core 9800GT compared to a single Athlon core because of extensive use of 64-bit integer arithmetic (implemented in software/firmware on top of 32-bit hardware?)
- **mrg32k3a** speed-up is $13.5\times$ when one includes the conversion to unit Normals
- have also implemented a Sobol generator to produce quasi-random numbers
- Sobol speedup is about $45\times$ because it uses 32-bit arithmetic

Random number generation

Other output distributions:

- exponential: trivial
- Normal: Box-Muller or inverse CDF
- Gamma: only efficient approaches using “rejection” methods which require a varying number of uniforms to generate 1 Gamma variable – this means no efficient skip-ahead, because don’t know how many uniforms will be needed to generate 1000 Gammas

Sequential Monte Carlo

- also known as particle filter
- an alternative to Kalman filters used for estimating some underlying state based on a sequence of observations and a model for the underlying evolution
- lots of applications in finance, economics, signal processing, tracking, statistical genetics
- main computation involves independent updates of the state of a large number of particles – trivially parallelisable

Sequential Monte Carlo

- tricky bit is re-sampling step which involves parallel scan operation to compute cumulative sums of normalised weights

$$C_n = \sum_{i=1}^{n-1} w_n$$

  and then use of recursive bisection and textures to find for any U_m the n such that

$$C_n \leq U_m < C_{n+1}.$$

- $55\times$ speedup with 112-core 9800GT compared to single Intel core
- currently working with colleagues (Chris Holmes, Neil Shephard) to develop generic particle filter library

Finite difference applications

- began with Jacobi iteration for simple Laplace equation on a regular gird
- then explicit and implicit (ADI) time-marching for 3D finance PDEs
- fairly 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
- tricky bits: maximising data re-use, minimising bandwidth required and working with limited shared memory

Jacobi Iteration

- a grid of size $512 \times 512 \times 512$ is broken up into blocks of size $32 \times 8 \times 512$ (plus halo)
- each grid block is worked on by a block of 256 threads, so each thread works along a grid line z-direction
- it would be very inefficient for each thread to load in both its data and its neighbours, so instead each loads its data into shared memory for access by neighbours
- problem: too little shared memory to hold whole block
- solution: hold 3 working planes at a time

Jacobi Iteration

- 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
- explicit time-marching is very similar

Implicit ADI Time-Marching

- each timestep involves 4 main stages:
  - compute r.h.s. (similar to Jacobi iteration)
  - solve tri-diagonal equations in x-direction
  - solve tri-diagonal equations in y-direction
  - solve tri-diagonal equations in z-direction
- between each phase, all data is held in graphics memory
- hence, can use a different data partitioning for each phase, unlike a standard MPI implementation

Implicit ADI Time-Marching

- first phase uses same partitioning as Jacobi iteration
- other phases use directional partitioning so each thread handles tri-diagonal solution along one line
- exploits parallelism of independent tri-diagonal solutions, rather than parallelising each one
- speedup is the same as for explicit time-marching, probably because bandwidth limited on both CPU and GPU

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 at least $10\times$ speedup and improvement in GFLOPS/$ and GFLOPS/watt
- Effectively a personal cluster in a PC under your desk
- Needs more work on tools and libraries to simplify development effort
- IT staff in banks very interested; quants will become convinced once tools/libraries are ready

Further Information

Wikipedia overviews of NVIDIA hardware:
`en.wikipedia.org/wiki/GeForce_200_Series`
`en.wikipedia.org/wiki/Nvidia_Tesla`

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

RNG library (free for academics):
`John.Holden@nag.co.uk`

LIBOR and finite difference test codes:
`www.maths.ox.ac.uk/~gilesm/hpc/`

Particle filter library (first version in a month?):
`Mike.Giles@maths.ox.ac.uk`