Using GPUs for Monte Carlo and Finite Difference Computations

Mike Giles

Oxford-Man Institute of Quantitative Finance
Oxford e-Research Centre

Julien Demouth, Jeremy Appleyard (NVIDIA), Endre László (Oxford)

New Thinking in Finance

Feb. 12th, 2014

GPUs

In the last 6 years, GPUs have emerged as a major new technology in high performance computing:

- approximately 20% of the Top500 supercomputer list
- two large systems in the UK – Emerald and Wilkes
- over 1000 GPUs used at JP Morgan, and over 200 at Bloomberg
- also used at a number of other banks and financial institutions
- use is driven by both energy efficiency and price/performance
- (even my MacBook has 384 graphics cores)

GPU hardware

The organisation and capabilities of a GPU are well illustrated by an NVIDIA K20X:

- 2688 cores arranged in 14 SMX functional units
- each SMX has
  - 192 cores, which can be viewed as 6 vector units of length 32
  - 64kB local shared memory / L1 cache
- shared 1.5MB L2 cache
- 288GB/s bandwidth to 6GB of GDDR5 graphics memory
- 8GB/s PCIe connection to CPU
- up to 3.9 TFlops in single precision, 1.3 TFlops in double precision

GPU software

For NVIDIA GPUs, software is written in CUDA, an extension of C/C++:

- host code on CPU; kernel code on GPU
- host launches multiple instances of a kernel known as *thread blocks* on the GPU, with each thread block running on one of the SMX units
- each thread block runs independently of the others
- within a thread block:
    - threads work in vector groups of 32 known as a *warp*
    - threads can communicate through local shared memory
- within a warp:
    - threads can exchange register data through a *shuffle* instruction
- usually have multiple active threads per core to hide memory latencies – often 10,000 active threads on a single GPU

Monte Carlo simulations

Monte Carlo simulations are naturally parallel – ideally suited to GPU execution:

- each path calculation is independent
- averaging of values computed by each thread is parallelised by using binary tree reduction:
  - sum in pairs recursively, until only one value per thread block
  - pass sums back to CPU by final summation
    <u>*or*</u> use atomic operations with data in graphics memory
- key requirement is parallel random number generation

Random number generation

Key is to use well-tested efficient libraries

- NVIDIA's CURAND library
- NAG's GPU library

Both offer a number of generators:

- `mrg32k3a`
- Mersenne Twister
- Sobol quasi-random
- . . . and others

and a number of output distributions:

- uniform
- Normal / log-Normal
- Gamma (NAG only?)
- Poisson (CURAND only?)

CURAND performance

Double precision performance – Gsamples / sec

	uniform	Normal
mrg32k3a	10.5	3.5
Sobol	14.8	6.3

Monte Carlo simulations

Two approaches:

- pre-compute and store a huge set of random numbers
  - separate random number generation from path calculation
  - needs distribution to be known beforehand
  - no registers required in path calculation, but small performance penalty in reading in random numbers
  - need to ensure that each thread gets a different random number, and each thread warp loads in a contiguous block of random numbers (non-trivial with rejection sampling)
- generate them on-the-fly as needed
  - only option when distribution is not known *a priori* (e.g. Poisson distribution with path-dependent rate)
  - minimises data transfer, but needs registers to hold state of random number generator (can limit number of threads, and hence hit performance)

Monte Carlo simulations

Additional complications?

- local volatility surface:
  - store volatility surface data in shared memory for use by all threads
- Longstaff-Schwartz regression for American options:
  - compute paths in parallel and store in 6GB of graphics memory
  - work backwards in time, use binary tree summation method to assemble regression matrices and r.h.s. to obtain approximate exercise value
  - see recent paper by Massimiliano Fatica (NVIDIA) (STAC-A2 report: `http://www.stacresearch.com/node/15807`)

Monte Carlo simulations

Final comment: see talk tomorrow at 4:00 by Hicham Lahlou (Xcelerit) on "Running Credit Value Adjustment on GPUs"

Xcelerit software uses a high-level C++ approach so that application developers can write CPU host code which gets automatically transformed into GPU code.

See `www.xcelerit.com` for more details.

Finite Difference calculations

Explicit time-marching methods are naturally parallel – again a good target for GPU acceleration

Implicit time-marching methods usually require the solution of lots of tridiagonal systems of equations – not so clear how to parallelise this.

Other key observation is that when moving lots of data to/from the main graphics memory, the cost of this may exceed the cost of the floating point computations – hence, try to avoid this data transfer.

Finite Difference calculations

In 1D a simple explicit finite difference equation takes the form

$$u_j^{n+1} = a_j\, u_{j-1}^n + b_j\, u_j^n + c_j\, u_{j+1}^n$$

while an implicit finite difference equation takes the form

$$a_j\, u_{j-1}^{n+1} + b_j\, u_j^{n+1} + c_j\, u_{j+1}^{n+1} = u_j^n$$

requiring the solution of a tridiagonal set of equations.

What performance can be achieved?

Finite Difference calculations

- grid size: 256 points
- number of options: 2048
- number of timesteps: 50000 (explicit), 2500 (implicit)
- results are for a K20c, about 20% slower than a K20X

	single prec.		double prec.	
	msec	GFlops	msec	GFlops
explicit1	347	454	412	382
explicit2	89	1763	160	980
implicit1	28	1308	80	637
implicit2	33	1377	88	685
implicit3	14	1103	30	505

Testing by Jorg Lotze (Xcelerit) shows implicit3 speeds are 12-14× faster than a pair of 8-core Intel Xeon E5-2670 CPUs, using full AVX vectoristion.

Finite Difference calculations

Approach:

- each thread block does one or more options
- doing an option calculation within one thread block means no need to transfer data to/from graphics memory – can hold all data in SMX
- explicit1 holds data in shared memory – performance is limited by the speed of shared memory access
- explicit2 holds all data in registers
  - each thread handles 8 grid points, so each warp handles one option
  - exchange of data with neighbouring threads is performed using shuffle instructions
  - 3 FMA (fused multiply-add) operations per grid point per timestep – 90% of theoretical peak performance in double precision

Finite Difference calculations

Interesting challenge is how best to solve tridiagonal systems for implicit solvers.

- want to keep computation within an SMX and avoid data transfer to/from graphics memory
- prepared to do more floating point operations if necessary to avoid the data transfer
- need parallelism to achieve good performance

Finite Difference calculations

On a CPU, the tridiagonal equations

$$a_i\, u_{i-1} + b_i\, u_i + c_i\, u_{i+1} = d_i, \quad i = 0, 1, \ldots, N-1$$

would usually be solved using the Thomas algorithm – essentially just standard Gaussian elimination exploiting all of the zeros.

- inherently sequential algorithm, so would require each thread to handle separate option
- threads don't have enough registers to store the required data – would require additional data transfer (almost double) to/from graphics memory to hold / recover data from "forward sweep"
- not a good choice – want an alternative with reduced data transfer, even if it requires more floating point ops.

Finite Difference calculations

PCR (parallel cyclic reduction) is an alternative parallel algorithm.

Starting with

$$a_i\, u_{i-1} + u_i + c_i\, u_{i+1} = d_i, \qquad i = 0, 1, \ldots, N-1,$$

where $u_j = 0$ for $j < 0, j \geq N$, can subtract multiples of rows $i \pm 1$, and re-normalise, to get

$$a'_i\, u_{i-2} + u_i + c'_i\, u_{i+2} = d'_i, \qquad i = 0, 1, \ldots, N-1,$$

Repeating with rows $i \pm 2$ gives

$$a''_i\, u_{i-4} + u_i + c''_i\, u_{i+4} = d''_i, \qquad i = 0, 1, \ldots, N-1,$$

and after $\log_2 N$ repetitions end up with solution because $u_{i \pm N} = 0$.

Finite Difference calculations

Using PCR we would have:

- 1 grid point per thread
- multiple warps for each option, so data exchange via shared memory – not ideal
- $O(N \log_2 N)$ floating point operations – quite a bit more than Thomas algorithm

Finite Difference calculations

This leads us to a hybrid algorithm: implicit1.

- follows data layout of explicit2 with each thread handling 8 grid points – means data exchanges can be performed by shuffles
- each thread uses Thomas algorithm to obtain middle values as a linear function of two (not yet known) "end" values

$$u_{J+j} = A_{J+j} + B_{J+j}\, u_J + C_{J+j}\, u_{J+7}, \quad 0 < j < 7$$

- the reduced tridiagonal system of size 2×32 for the "end" values is solved using PCR
- total number of floating point operations is approximately double what would be needed on a CPU using the Thomas algorithm

Finite Difference calculations

implicit2 is very similar to implicit1, but instead of solving

$$a_j\, u_{j-1}^{n+1} + b_j\, u_j^{n+1} + c_j\, u_{j+1}^{n+1} = u_j^n$$

it instead computes the change $\Delta u_j \equiv u_j^{n+1} - u_j^n$ by solving

$$a_j\, \Delta u_{j-1} + b_j\, \Delta u_j + c_j\, \Delta u_{j+1} = d_j^n$$

and then updates u_j. This gives better accuracy, which might be important if working in single precision.

If the matrices do not change each timestep, then some parts of the tridiagonal solution do not need to be repeated each time – implicit3 exploits this, but is otherwise the same as implicit1.

Finite Difference calculations

What about a 3D extension on a 256^3 grid?

- memory requirements imply one kernel with multiple thread blocks to handle a single option
- kernel will need to be called for each timestep, to ensure that the entire grid is updated before the next timestep starts
- based on previous experience with a 3D Jacobi iteration, initial implementation of explicit algorithm uses a separate thread for each grid point in 2D x-y plane, then marches in z-direction
- initial implementation relies on cache for data re-use – performance appears to be bandwidth-limited, achieving roughly 25% of peak bandwidth
- will experiment with the use of shared memory for better data re-use, at the expense of nastier programming

Finite Difference calculations

For implicit time-marching, the ADI discretisation requires the solution of a tridiagonal equations along each line in the x-direction, and then the same in the y- and z-directions.

Jeremy Appleyard (NVIDIA) has developed library software for this, based on the 1D hybrid PCR code – better than the Thomas method because it involves less data transfer to/from graphics memory. The clever part of his implementation is in the data transpositions required to maximise bandwidth – a bit like transposing a matrix.

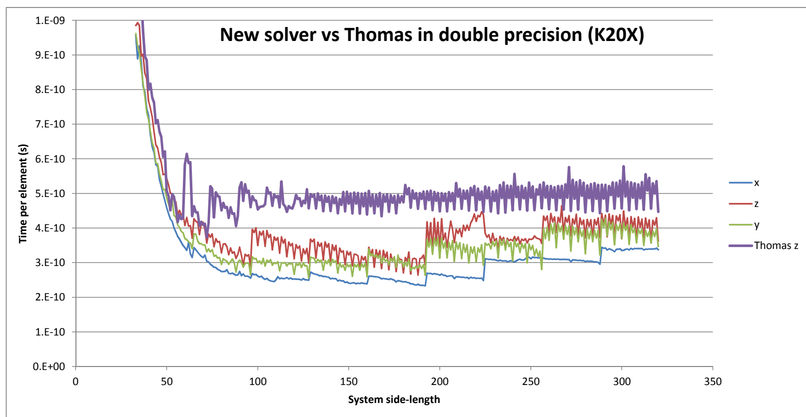
The final code will have the following structure:

- kernel similar to explicit kernel to produce matrices and r.h.s.
- 3 calls by host code to batch tridiagonal solver to perform the tridiagonal solutions in each direction

Batch Tridiagonal Solver

Hybrid PCR vs. Thomas algorithm for different tridiagonal system lengths

Note: at least 40 bytes/element $\Longrightarrow$ 1.4E-10 secs at max. bandwidth and Hybrid PCR compute time $\Longrightarrow$ 0.7E-10 secs / element



Finite Difference calculations

Other dimensions?

2D:

- if the grid is small (128^2?) one option could fit within a single SMX
  - in this case, could adapt the 1D hybrid PCR method for the 2D ADI solver
  - main complication would be transposing the data between the x-solve and y-solve so that each tridiagonal solution is within a single warp
- otherwise, will have to use the 3D approach, but with solution of multiple 2D problems to provide more parallelism

4D:

- same as 3D, provided data can fit into graphics memory (otherwise buy a K40 with 12GB graphics memory!)

Conclusions

- GPUs can deliver excellent performance for both Monte Carlo and finite difference calculation
- some parts of the implementation are straightforward, but others require a good understanding of the hardware and parallel algorithms to achieve the best performance
- the key is to use libraries as much as possible, or alternative high-level approaches like Xcelerit

For further info see

http://people.maths.ox.ac.uk/gilesm/codes/BS_1D/

and also attend Hicham Lahlou's talk tomorrow at 4:00.