RESEARCH
COUNCILS UK

HECToR

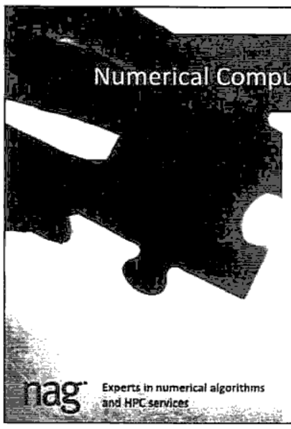
CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

NAG IV

Robert Tong

GPU accelerated numerical computing



Numerical Computing with GPUs

Robert Tong

30 September 2011

nag Experts in numerical algorithms and HPC services

Outline

- Hardware: GPU is a vector processor
- Accessing the GPU on your system
- Programming your GPU
- NAG routines for GPUs
  - Application 1: Monte Carlo simulation
  - Application 2: PDE solvers

nag 2

The need for speed

The challenge

e.g. Financial markets

- High speed data feeds
- Algorithmic trading
- Risk assessment – overnight ⟶ intraday
- Reduce power consumption and cost of computation

The solution

- GPU computing

nag 3

Top 5 supercomputers – June 2011 (www.top500.org)

1	Fujitsu	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect	Japan
2	NUDT	NUDT TH MPP, X5670 2.93Ghz 6C, **NVIDIA GPU**, FT-1000 8C	China
3	Cray Inc.	Cray XT5-HE Opteron 6-core 2.6 GHz	US
4	Dawning	Dawning TC3600 Blade, Intel X5650, **NVidia Tesla C2050 GPU**	China
5	NEC/HP	HP ProLiant SL390s G7 Xeon 6C X5670, **Nvidia GPU**, Linux/Windows	Japan

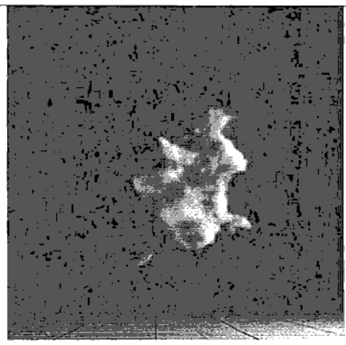
nag 4

Vector processors – an important element in HPC

- SIMD – Single Instruction Multiple Data
  - Instructions operate on arrays of data
- Basis of supercomputers through 1980s
  - *e.g.* IBM 3090
- Vector processing included in most current CPUs as SSE (Streaming SIMD Extensions) instruction set
- **GPUs are vector processors**
  - Particularly effective for updating graphics display

nag 5

Smoke particles – GPU simulation and visualisation



nag 6

Accessing your GPU

1. Need CUDA-enabled GPU

   (NVIDIA – most GeForce, Tesla, Quadro) Check with
   - Windows –> Control Panel – System – Hardware – Device Manager – Display adapters
   - Linux –> lspci | grep –i nvidia
2. When installing GPU, get appropriate driver
3. Download:
   - CUDA Toolkit & GPU Computing SDK

See: http://www.nvidia.com/cuda

CUDA_C_Getting_Started_Windows.pdf

CUDA_C_Getting_Started_Linux.pdf

nag 7

Run: deviceQuery (included in NVIDIA_CUDA_SDK)

```
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA

Device 0: "Quadro FX 5800"
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 4294246400 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host threads can use
    this device simultaneously)

Test PASSED
```

nag 8

Run: bandwidthTest (included in NVIDIA_CUDA_SDK)

```
Running on......
      device 0:Quadro FX 5800
  Quick Mode
  Host to Device Bandwidth for Pageable memory
  .
  Transfer Size (Bytes)  Bandwidth(MB/s)
   33554432              2799.1

  Quick Mode
  Device to Host Bandwidth for Pageable memory
  .
  Transfer Size (Bytes)  Bandwidth(MB/s)
   33554432              2391.6

  Quick Mode
  Device to Device Bandwidth
  .
  Transfer Size (Bytes)  Bandwidth(MB/s)
   33554432              73117.6

  &&&& Test PASSED

  Press ENTER to exit...
```

nag 9

Programming your GPU

- Access through
  - OpenCL – standard programming language for heterogeneous systems/devices (multi-core, GPU, ...)
  - CUDA C/C++ – for NVIDIA GPUs
  - PGI Fortran – for NVIDIA GPUs
  - OpenGL, DirectX – for graphics
  - GPU assembly language (PTX for NVIDIA GPUs)

nag 10

Host (CPU) – Device (GPU) Relationship

- Application program initiated on Host (CPU)
- Device 'kernels' execute on GPU in SIMT (Single Instruction Multiple Thread) manner
- Host program
  - Transfers data from Host memory to Device (GPU) memory
  - Specifies number and organisation of threads on Device
  - Calls Device 'kernel' as a C function, passing parameters
  - Copies output from Device back to Host memory

nag 11

Compute Unified Device Architecture (CUDA)

- Extension of C to enable programming of GPU devices
- Developed by NVIDIA
- Parallel threads managed as 'kernels' (sequence of operations in each thread)
- 'kernels' are scalar - 'kernel' invoked as a thread over the set of specified threads
- Threads synchronise using barriers
- Synchronisation among thread blocks achieved on completion of 'kernel'

nag 12

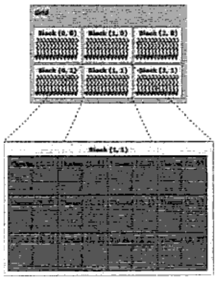
Compilers

- nvcc – compiler driver required for CUDA extensions
  - Compiles to **PTX** (Parallel Thread eXecution)
    - Object file or
    - PTX interpreted at runtime
  - Executable depends on
    - **cudart** CUDA runtime library
    - **cuda** core library
- Requires C compiler on Host system
  - MS Visual Studio C/C++
  - Linux: gcc

nag 13

Parallelism through: Grid – Blocks – Threads

Recall primary function – updating graphics display



nag 14

Organisation of threads on GPU

- Tesla GPU – divided into SM (Streaming Multiprocessor) units with typically 32 cores per SM
- SM manages threads
- Each thread is identified by `threadIdx.x`
- Threads execute as Warps of 32 threads
- Threads are grouped in blocks identified by `blockIdx.x` (user specifies number of threads per block, `blockDim.x`)
- Blocks make up a grid, size given as `gridDim.x`

nag 15

A first GPU program (1)

1. Define kernel function to be executed on GPU (Device):
   - Identify as `__global__`
2. Allocate memory on Device: `cudaMalloc`
3. Copy data from Host to Device: `cudaMemCpy`
4. Call kernel function
   - `kernelFun<<<numBlocks, threadsPerBlock>>>(...)`
5. Copy data from Device to Host (if needed): `cudaMemCpy`
6. Free memory on Device: `cudaFree`

nag 16

Sum of 2 vectors: *i* = threadIdx.x + blockDim.x * blockIdx.x

0	1	2	3	4	5	thread *i*
a_0	a_1	a_2	a_3	a_4	a_5	…
+	+	+	+	+	+	…
b_0	b_1	b_2	b_3	b_4	b_5	
c_0	c_1	c_2	c_3	c_4	c_5	

nag 17

A first GPU program (2) – kernel (executes on Device)

```
// Kernel to add 2 vectors
__global__ void vecAdd_gpu(float* d_a, float*
 d_b, float* d_c)
 {
 int i = threadIdx.x + blockDim.x*blockIdx.x;
 d_c[i] = d_a[i] + d_b[i];
 }
```

nag 18

A first GPU program (3) – calling program (Host)

```
// Host -- main program
  int main(int argc, char *argv[])
  {  ...
// Copy data from Host to Device
cudaMemcpy(d_a, h_a, sizeof(float)*N,
  cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, sizeof(float)*N,
cudaMemcpyHostToDevice);
// Add vectors on Device (GPU)
vecAdd_gpu<<<num_blocks, num_threads>>>(d_a, d_b, d_c);

// Copy result to Host if required
cudaMemcpy(h_c, d_c, sizeof(float)*N,
  cudaMemcpyDeviceToHost);
...}
```

nag 19

Memory hierarchy

Memory	Location On/off chip	Cached	Access	Scope	Lifetime
Register	On	-	R/W	1 thread	Thread
Local	Off	Yes (2.x)	R/W	1 thread	Thread
Shared	On	-	R/W	All threads in block	Block
Global	Off	Yes (2.x)	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

nag 20

Books on CUDA C for GPUs

- Rob Farber (Dec 2011)
  CUDA Application Design and Development
- David Kirk & Wen-mei Hwu (2010)
  Programming Massively Parallel Processors: A Hands On Approach
- Jason Sanders & Edward Kandrot (2010)
  CUDA by Example: An Introduction to General-Purpose GPU Programming

nag 21

NAG routines for GPUs

- Developing applications for GPUs should be straightforward
- Must eliminate
  - The time-consuming work of writing basic numerical components
  - Repeating standard programming tasks on the GPU
- The solution
  - Numerical components should be available as libraries for GPUs .... NAG routines

nag 22

NAG GPU library functions

- Provide a set of algorithms that
  - Make efficient use of the GPU architecture
  - Facilitate the development of applications at different levels of complexity
    - Low level – insert library functions on the device (GPU)
    - High level – hide GPU complexity by working from host (CPU)
- Enable maximum flexibility in use of hardware systems
  - Multi-core CPU + GPU

nag 23

Application 1: Monte Carlo simulation

- Random Number Generators (RNGs)
  - Pseudo-random
  - Quasi-random
    - Randomization
  - Brownian bridge constructor
- Solve Stochastic Differential Equation

$$\frac{dS_t}{S_t} = \mu dt + \sigma dW_t$$

(drift + Brownian motion)

  - Discretize and use RNGs to simulate paths

nag 24

Random Number Generators: choice of algorithm for GPU

- Must be highly parallel
  - Use ***skip ahead*** to initialise streams of numbers
- Implementation must satisfy statistical tests of randomness
- Some common generators do not guarantee randomness properties when split into parallel streams
- A suitable choice:
  - MRG32k3a (L'Ecuyer) with fast skip ahead
  - Mersenne Twister (MT19937) skip ahead, but large initial state

nag 25

NAG random number generators

Generators		Fermi GPU	Intel MKL on Xeon E5410			
		GPU pts/ms	1 Thread	2 Threads	3 Threads	4 Threads
MRG	Unif	7.7127E+06 **7.4453E+06**	88.108x **108.64x**	52.854x **74.197x**	41.900x **73.703x**	34.022x **71.321x**
	Exp	5.1368E+06 **2.0090E+06**	76.024x **47.935x**	44.643x **29.082x**	34.378x **26.143x**	28.767x **25.148x**
	Norm	4.6129E+06 **2.4418E+06**	81.436x **66.789x**	41.348x **38.034x**	31.022x **27.873x**	26.291x **23.044x**
Sobol	Unif	1.7434E+07 **1.3452E+07**	140.97x **142.68x**	103.76x **132.10x**	103.98x **123.00x**	71.724x **129.88x**
	Exp	7.3661E+06 **3.2094E+06**	60.732x **43.312x**	48.157x **35.404x**	39.014x **30.108x**	37.243x **30.304x**
	Norm	8.0020E+06 **1.0202E+06**	66.137x **21.904x**	52.291x **18.179x**	[illegible] **15.547x**	40.346x **15.314x**
Mersenne	Unif	2.9077E+06 **2.8728E+06**	27.299x **44.961x**			
	Exp	2.2352E+06 **1.2465E+06**	26.089x **23.097x**			
	Norm	2.1965E+06 **8.8145E+05**	26.430 **10.407x**			

Benchmark figures for Tesla C2050 vs. Intel Xeon E5410. Values in bold type are double precision, other values are single precision. Speed up of GPU over CPU is shown for 1-4 threads on CPU.

nag 26

LIBOR Market Model on GPU

Equally weighted portfolio of 15 swaptions each with same maturity, but different lengths and different strikes

Seed	Value (bps)	GPU time(msec)	CPU time(msec)
78, 234, 786 254, 758, 695	4938.8	2879	216360

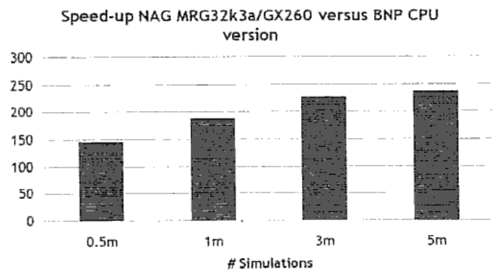
nag 27

Early Success with BNP Paribas

- Working with Fixed Income Research & Strategies Team (FIRST)
  - NAG mrg32k3a works well in BNP Paribas CUDA "Local Vol Monte-Carlo"
  - Passes rigorous statistical tests for randomness properties (TestU01, L'Ecuyer et al)
  - Performance good
  - Being able to match the GPU random numbers with the CPU version of mrg32k3a has been very valuable for establishing validity of output

nag 28

BNP Paribas Results – local vol example



nag 29

Application 2: NAG PDE solvers

- Solve generic 3D PDE

$$\frac{\partial V}{\partial t} + \mu_1\frac{\partial V}{\partial x_1} + \mu_2\frac{\partial V}{\partial x_2} + \mu_3\frac{\partial V}{\partial x_3} + \sigma_1^2\frac{\partial^2 V}{\partial x_1^2} + \sigma_2^2\frac{\partial^2 V}{\partial x_2^2} + \sigma_3^2\frac{\partial^2 V}{\partial x_3^2}$$
$$+\beta_1\sigma_2\sigma_3\frac{\partial^2 V}{\partial x_2\partial x_3} + \beta_2\sigma_3\sigma_1\frac{\partial^2 V}{\partial x_3\partial x_1} + \beta_3\sigma_1\sigma_2\frac{\partial^2 V}{\partial x_1\partial x_2} = s\,V$$

- Method: finite difference
  - Alternating Direction Implicit (ADI)
  - Output: price and Greeks

nag 30

3D ADI for option pricing

- Suitable for
  - Multi-asset options, rainbow options, basket options, cross-currency swaps, power reverse dual currency swaps
    see Dang et al (2010)
- Some initial timings

Using device: Quadro FX 5800 (N=number of time steps)
CPU...........N = 40, 2332.266113 (ms)
GPU...........N = 40, 108.735001 (ms), speedup = 21.5x

CPU...........N = 80, 37175.652344 (ms)
GPU...........N = 80, 1412.394043 (ms), speedup = 23.3x

nag 31

Using the NAG GPU routines

- Program in CUDA C/C++
  - For greatest flexibility

OR

- Access the power of GPUs from
  - C++
  - Excel
  - Python
  - ...

(Note: much research is currently being directed at developing high level application specific languages to facilitate use of multi-core and GPU systems)

nag 32

Example Program: naggpu_rand_normalA

```
#include <nag_gpu.h>
int main(int argc, char **argv)
{ ...
double *d_buff = 0;
double mu = 0.0, sigma = 1.0;
NagGpuRandComm comm;
 // arbitrary seed and skip ahead parameters
  for (int i = 0; i < seed_length; i++) seed[i] = i;
  a1 = 14, b1 = 34, a2 = 2, b2 = 21, c = 123;
 naggpuRandInitA(NAGGPURANDGEN_MT19937, a1, b1, a2, b2,
                 c, seed, &comm, &error);
 naggpuRandNormalA(N/2, NAGGPURANDORDER_CONSISTENT, mu,
   sigma,d_buff, NULL, 0, &comm, &error);
checkNagError(&error);
```

nag 33

NAG Numerical Routines for GPUs
Random Number Generators Chapter Contents
Random Number Generators Chapter Introduction

NAG Numerical Routines for GPUs Function Document

naggpuRandNormalA

+ Contents

1 Purpose

naggpuRandNormalA generates n values X_i from a Normal distribution with mean μ and variance σ^2.

The initialization function naggpuRandInitA must be called prior to the first call to naggpuRandNormalA. Thereafter, this function may be called repeatedly to generate additional sets of random values. Once all desired values have been obtained, the function naggpuRandCleanupA must be called to free allocated system resources.

nag 34

5 Arguments

1: **n** - int *Input*

On entry: the number of random values to be generated.

Constraint: n ≥ 1.

2: **order** - NagGpuRandOrder *Input*

On entry: the ordering to be observed by the underlying GPU generator:

order = NAGGPURANDORDER_OPTIMAL

order = NAGGPURANDORDER_CONSISTENT

See NagGpuRandOrder for further details.

Constraint: order = NAGGPURANDORDER_OPTIMAL or NAGGPURANDORDER_CONSISTENT.

3: **mu** - float *Input*
3: **mu** - double *Input*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

On entry: the mean, μ, of the distribution.

4: **sigma** - float *Input*
5: **sigma** - double *Input*

nag 35

This parameter has type float or double depending on whether the single or double precision version of this function is called.

On entry: the standard deviation, σ, of the distribution.

Constraint: sigma > 0.

6: **d_buff[n]** - float * *Output*
7: **d_buff[n]** - double * *Output*

This parameter has type float or double depending on whether the single or double precision version of this function is called.

This buffer must reside in the GPU memory space.

On exit: the n pseudorandom numbers from the specified distribution. The output tuning structure comm → tuneParamsUsed will contain the parameters used to launch the kernel. If order = NAGGPURANDORDER_OPTIMAL, these parameters may determine the output ordering (see NagGpuRandTune for details).

8: **tune** - const NagGpuRandTune * *Input*

This parameter is optional and may be set to NULL.

On entry: if specified, points to a NagGpuRandTune structure containing launch parameters for the selected GPU kernel. Upon a successful return from this function, the relevant data will be copied to the output tuning structure comm → tuneParamsUsed. Please see NagGpuRandTune for additional information about performance tuning.

10: **custream** - cudaStream_t *Input*

On entry: specifies the CUDA stream on which to launch the selected GPU kernel. If no

nag 36

streams are used, set this parameter to 0. Please see the chapter on Streams in the CUDA Programming Guide for further details.

11: comm - NagGpuRandComm * *Communication Data*

NagGpuRandComm is a structure which holds state and communication information and must not be modified in any way. Once all required points have been obtained, comm must be passed to naggpuRandCleanupA to free allocated system resources.

Upon successful return from this function, the launch configurations applied to the underlying GPU kernels may be observed through comm. This will typically only be of interest to users wanting to fine tune the performance of this function. Please see NagGpuRandTune for details on performance tuning, and consult the NagGpuRandComm documentation for how to observe the launch parameters. Note that these parameters are no longer observable after calling naggpuRandCleanupA.

12: error - NagGpuError * *Error Reporting*

This parameter contains error information and should not be modified directly. Errors are indicated through the value of error → code which should be inspected after each call to this function. If error → code = 0 then no error occurred. If error → code ≠ 0 then an error was detected and a call to naggpuErrorCopyMsg will retrieve a null terminated ANSI C string describing the error. Please see the Error Handling Chapter Introduction for further details on error handling.

6 Error Indicators and Warnings

error → code = 1

nag

On entry: the CUDA runtime error status has not been cleared, indicating a previous CUDA error. Call cudaGetLastError() in the CUDA runtime library to clear the runtime error status.

error → code = 2

During execution: a CUDA runtime error was detected.

error → code = 3

On entry: an attempt was made to launch a double precision function on a GPU device that does not support double precision.

error → code = 100

On entry: the value of comm is NULL.

error → code = 101

On entry: comm has not been initialized, or the internal state of comm is corrupted.

error → code = 110

On entry: $n \le 0$.

error → code = 111

On entry: order does not specify a valid ordering. See NagGpuRandOrder for permitted values.

error → code = 112

On entry: d_buff is NULL.

error → code = 115

nag

NAG GPU routines (1)

Functionality Index

Host-Callable Generator Functions and Error Handling

Brownian bridge, depth-ordered construction,

free generator resources ....naggpuDepthBBCleanupA
generate Brownian bridge ....naggpuDepthBBA
generate Brownian bridge increments .....naggpuDepthBBIncA
initialize bridge generator ....naggpuDepthBBInitA
initialize incremental bridge generator .....naggpuDepthBBIncInitA

Error handling,

retrieve an error message ....naggpuErrorCopyMsg

nag

NAG GPU routines (2)

Pseudorandom numbers,

free generator resources ....naggpuRandCleanupA
initialize generator ....naggpuRandInitA
variates from exponential distribution ....naggpuRandExpA
variates from Normal distribution ....naggpuRandNormalA
variates from uniform distribution ....naggpuRandUniformA

Quasi-random numbers,

free generator resources ....naggpuQuasiRandCleanupA
initialize generator ....naggpuQuasiRandInitA
variates from exponential distribution ......naggpuQuasiRandExpA
variates from Normal distribution ....naggpuQuasiRandNormalA
variates from uniform distribution ....naggpuQuasiRandUniformA

nag

NAG GPU routines (3)

Inline Device Function Generators

Pseudorandom numbers,

host cleanup for inline device function generator
......................naggpuMrg32k3aDeviceCleanupA
host setup for inline device function generator
.... naggpuMrg32k3aDeviceInitA
initialize inline device function generator
....naggpudevMrg32k3aInitA
next variate from exponential distribution
....naggpudevMrg32k3aExpA
next variate from Normal distribution
.... naggpudevMrg32k3aNormalA
next variate from uniform distribution
....naggpudevMrg32k3aUniformA

nag

NAG GPU routines (4)

Quasi-random numbers,

host cleanup for inline device function generator
..............................naggpuSobolDeviceCleanupA
host setup for inline device function generator
....naggpuSobolDeviceInitA
initialize inline device function generator
....naggpudevSobolInitA
next point from exponential distribution
....naggpudevSobolExpA
next point from Normal distribution
....naggpudevSobolNormalA
next point from uniform distribution
....naggpudevSobolUniformA

nag

NAG GPU routines (5)

Host-Callable Linear Equation (LAPACK) Functions
Linear Equations

Cholesky factorization of a real symmetric positive definite matrix	**naggpuDpotrfA**
free system resources	**naggpuLinAlgCleanupA**
initialise the linear equation functions	**naggpuLinAlgInitA**
LU factorization of a real m by n matrix	**naggpuDgetrfA**

nag 43

Calling the NAG GPU routines from C++

```
// Set option data here
data[0] = new MyCallData(0.10f, 0.20f, 100.0f, 100.0f, 1.7f);
...
MultiOptionSobol<float> demo(
      false,
      CACHED_MIL,          // Type of algorithm
      1<<18,               // nTrials - must be less than 2^21
      nTimes,              // nTimeSteps - must be less than 4095
      times,               // Array of times, in increasing order
      data,
      nCalls);
...
// Compute GPU value
demo.runGPUsim();
```

nag 44

Or using CUDA C/C++

```
template<class FP>
double MultiOptionSobol<FP>::runGPUsim()      // FP can be 'double' or 'float'
{
  ...
// Compute GPU launch params
int nBlks_x = getNumBlocksPerOption();
dim3 gridDim(nBlks_x, nCalls);

// Allocate memory for the partial sum storage
double *d_accum=0;
nag_gpu_utilSafeCall(cudaMalloc((void**)&d_accum,sizeof(double)*nBlks_x*nCalls) );
// Launch simulation on GPU
demo_BS_Mil_cached<FP><<<gridDim, thdsPerBlk, sizeof(double)*thdsPerBlk+sizeof
    (float)*nTimeSteps*3>>>
                                   (nTrials, nTimeSteps, d_times, d_data,
    workPerThd, d_W, d_r, d_r_pitch, d_sigma, d_sigma_pitch, d_accum);
```

nag 45

Results: NAG Sobol generator with Brownian bridge

Using NVIDIA Device: Quadro FX 5800

DESCRIPTION:
Simple Black-Scholes path dynamics with deterministic term structures of interest and volatility. Uses NAG GPU quasi-random (Sobol) generator, and constructs sample paths using a Brownian bridge.
ALGORITHM:
Milstein (with caching), SINGLE precision
RESULTS:
CPU option price = 12.66961491 CPU runtime = 12344.74219ms
GPU option price = 12.66961543 GPU runtime = 243.1423035ms
Speedup = 50.77167511x

nag 46

Device function

__device__ qualifier declares function
- Executed on Device (GPU)
- Callable from Device only

- Supplying Device level library functions provides greater flexibility for developers of GPU applications

nag 47

Device functions

```
template <typename FP>
#include <nag_gpu.h>
  #include <nag_gpu_sobolDevFuncs.h>

__device__ void naggpudevSobolNormalA(FP *x, FP mu,
  FP sigma, const int comm1, unsigned int &comm2,
  unsigned int *comm3,
  const NagGpuSobolDeviceComm *devComm)
```

nag 48

Maximizing performance

- Auto-tuning required
- Performance affected by mapping of algorithm to GPU via threads, blocks and warps
- Implement a code generator to produce variants using the relevant parameters
- Determine optimal performance

Example: auto-tuning

```
Start of Auto-tuning:
Tuning workPerThd=10...
Found new min: workPerThd=10,      thdsPerBlk=32,     Est.Blks/SM=109.233,
        runtime=49.4908ms          OptionVal=12.6696
Tuning workPerThd=10...
...
Tuning workPerThd=119...
Tuning workPerThd=120...
Auto-tuning complete
Runtime Statistics: nRuns=1776  ave=29.4808  min=16.674  max=84.7275
```


Conclusions

- Heterogeneous systems (multi-core processors with GPUs) offer faster computing at lower cost and lower energy consumption
- These systems already exist – from budget PCs to supercomputers
- Numerical libraries are essential to fully exploit this computing power
- The development of high quality software is a collaborative effort – your input is welcome

Acknowledgments

- Mike Giles (Mathematical Institute, University of Oxford) - algorithmic input
- Funding from Technology Strategy Board through Knowledge Transfer Partnership with Smith Institute
- NVIDIA for supply of Tesla C1060, Quadro FX 5800 and Tesla C2050
- See

www.nag.co.uk/numeric/gpus/index.asp