An Introduction to CUDA

Overview of GPU Architecture and the CUDA Language

Jacques du Toit, NAG
Some History – CPUs in the 90s

The Average Computer User …

- CPUs **not** designed for floating point calcs – designed to run business applications (databases, email, spreadsheets, minesweeper, Farm Ville, etc …)
- These programs all **serial**. But we have tons of transistors! CPUs evolved all sorts of tricks to run serial code in parallel and speed up access to data
  - Pipelines, branch prediction, o-o-o execution, large caches, etc …
- About 70% to 80% of CPU devoted to cache and decoding and managing the execution of the instruction stream
- Very little devoted to floating point compute
Some History – PC Games

Games (graphics) create huge need for floating point
- Floating point units on CPUs simply can’t cope
- CPUs borrowed an idea from 60s and 70s supercomputers
- Introduced vector units (MMX, SSE, SSE2, ..., AVX)

Idea is simple: get more floating point action per operation decoded
- Spread the cost of decoding and “managing” an instruction (e.g. floating point multiply) over several data elements
- I.e. make one instruction operate on several data elements at once!
- Relies crucially on parallelism and independence
Vector Units – The SIMD paradigm

Consider simple array pairwise product:

```c
void arrayProd(int n, const float *x, const float *y, float *z) {
    for(int i=0; i<n; i++) {
        z[i] = x[i] * y[i];
    }
}
```
Vector Units – The SIMD paradigm

Consider simple array pairwise product:

```c
void arrayProd(int n, const float *x, const float *y, float *z) {
    for(int i=0; i<n; i++) {
        z[i] = x[i] * y[i];
    }
}

void arrayProd(int n, const float *x, const float *y, float *z) {
    for(int i=0; i<n/4; i+=4) {
        z[i] = x[i] * y[i];
        z[i+1] = x[i+1] * y[i+1];
        z[i+2] = x[i+2] * y[i+2];
        z[i+3] = x[i+3] * y[i+3];
    }
}
```
Vector Units – The SIMD paradigm

Consider simple array pairwise product:

```c
void arrayProd(int n, const float *x, const float *y, float *z) {
    for(int i=0; i<n; i++) {
        z[i] = x[i] * y[i];
    }
}
```

```c
void arrayProd(int n, const float *x, const float *y, float *z) {
    for(int i=0; i<n/4; i+=4) {
        z[i] = x[i] * y[i];
        z[i+1] = x[i+1] * y[i+1];
        z[i+2] = x[i+2] * y[i+2];
        z[i+3] = x[i+3] * y[i+3];
    }
}
```

```c
void arrayProd(int n, const float *x, const float *y, float *z) {
    for(int i=0; i<n/4; i+=4) {
        __m128 sse_x = _mm_load_ps(x);  // Load 4 floats from x
        __m128 sse_y = _mm_load_ps(y);  // Load 4 floats from y
        __m128 sse_z = _mm_mul_ps(sse_x, sse_y); // One clock cycle!
        _mm_store_ps(z, sse_z);        // Store 4 floats to z
        z+=4;  x+=4;  y+=4;
    }
}
```
Vector Units – What are they?

- Bunch of electronic circuits on your chip
  - Operates on several data elements at once
  - Executes a single operation (+, -, x, /) at a time
  - Applies that operation to all data elements at once
Vector Units – Birth of the GPU

▶ Graphics processing fits vector model very well
  • SSE instructions operate on 4*32bit floats at once, and initially worked quite well

▶ PC games rapidly outstripped capability of SSE units
  • Specialist hardware (GPU) became necessary
  • GPUs designed to do huge amounts of FP on lots of data, but where calculations are highly parallel
  • Make extensive use of long vector units: 32*32bits on current cards!
GPUs – Summary

- Long vector units (in fact, *only* has vector units!)
- Low clock speeds (roughly 1GHz) due to heat
- **Very** fast memory (190GB/s) – 9x faster than CPU
- No advanced instruction processing
  - No branch predictions, o-o-o execution, etc ...
  - “Small” caches
  - Latencies/stalls hidden by giving it so much work to do that it’s always busy

Any questions?
CUDA – A PROGRAMMER’S VIEW
Hardware Model

Important to have following simple conceptual view:

- CPU
  - Core 1, Core 3
  - Core 2, Core 4
  - CPU Cache

- Main System Bus

- PCIe Bus 8GB/s

- System Memory
  - Size=16GB, Speed=20GB/s

- GPU
  - GPU Memory
    - Size=6GB, Speed=190GB/s
Suppose want to estimate the integral of a function using Monte Carlo. The integral is given by

\[ P = E[f(Z)] = \int_{-\infty}^{\infty} f(z) \frac{1}{\sqrt{2\pi}} \exp(-z^2 / 2) \, dz \]

where \( Z \) is a standard Normal random variable. The Monte Carlo estimator of this is

\[ \hat{P} = \frac{1}{N} \sum_{i=1}^{N} f(Z_i) \]

where each \( Z_i \) is an independent standard Normal random variable.
Learn by doing – Monte Carlo example

- We choose the function

\[
f(Z) = e^{-rT} \max\left[S_0 \exp\left((r - \sigma^2 / 2)T + \sigma \sqrt{T} Z\right) - K, 0\right]
\]

where \(S_0, r, T, K, \sigma\) are all constants given and fixed.

- Let’s write a small C program to do this
Monte-Carlo example – Serial

```cpp
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);
double sum = 0.0;

for(int i=0; i<N; i++) {
    double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqrt(T)*Z[i] );
    double x = (ST > K ? ST-K : 0.0);
    sum += exp(-r*T)*x;
}
double integral = sum / N;
delete Z;
```

- NAG routine `g05skc` generates array of Normal random numbers (assume generator already initialised)
- Assume constants `S0`, `r`, `T`, `K`, `sigma`, `N` already defined

▶ Any questions?
Monte-Carlo example – Parallelise?

- Loop iterations are independent – can be parallelised
- Strategy: carve up $N$ iterations between $M$ threads. Each thread computes $N/M$ iterations
- Take care collating running sums from $M$ threads at the end (synchronisation?!)
**Monte-Carlo example – First OpenMP**

```c
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);
double sum = 0.0;
#pragma omp parallel for shared(N,Z,T,sigma,r,K,S0) reduction(+:sum)
{
    for(int i=0; i<N; i++) {
        double ST = S0*exp((r-0.5*sigma*sigma)*T + sigma*sqrt(T)*Z[i]);
        double x = (ST > K ? ST-K : 0.0);
        sum += exp(-r*T)*x;
    }
}

double integral = sum / N;
delete Z;
```

- Reduction clause ensures threads update the “shared” variable `sum` in a safe manner – details are handled by compiler
Monte-Carlo Example – Timings

- Benchmark system
  - Intel Core i7 860 at 2.8GHz (quad core) running 4 threads
  - 8GB memory
  - Ubuntu 10.04 64bit with gcc v4.4.3
  - NVIDIA C2050
  - N = 100,000,000 simulations

- Serial CPU code
  - Loop time = 1157ms

- First OpenMP code
  - Loop time = 337ms
  - Speedup of 3.4x vs serial loop
GPUs and Kernels

- GPUs run *kernels*
  - Kernel is *function* written in CUDA and compiled with nvcc. When called, it is executed *M times in parallel* by *M* different CUDA threads.
  - Kernel functions can be big or small, but typically not very big. Big problems solved by launching several different kernels, each doing a different part of the computation.
  - A kernel launch takes roughly 10 microseconds (0.01ms!)

- NB: you hardly ever port entire program to GPU!
  - Leave serial portions on CPU, GPUs not designed for these
  - Therefore, *OpenMP parallel regions are good candidates for turning into kernels*
Typical GPU Application

A GPU accelerated application usually looks like this:

```c
/* Do setup and serial processing on CPU */
mySerialFunction( ... );

// Do some GPU setup - allocate GPU memory, etc ...

// Push data to GPU
// Launch CUDA kernel1

// Launch CUDA kernel2

// Get results from GPU

/* Do more serial processing on CPU */
anotherSerialFunc( ... );

// Push data to GPU
// Launch CUDA kernel3

// Launch CUDA kernel4
... etc ...
```
Monte-Carlo example – CUDA shell

```c
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);
double sum = 0.0;

// Copy Z to GPU

// Launch kernel to compute sums of integrands.
// Can CUDA do automatic reductions ???

// Copy sum of integrands back to CPU

double integral = sum / N;
delete Z;
```

CUDA Runtime library (cudart.lib, libcuda.so)

- Set of CPU routines, called from CPU code, to “manage” the GPU.
- Copies data back and forth, allocates memory, controls synchronization, error checking, etc.
Monte-Carlo example – CUDA shell

```c
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);
double sum = 0.0;

double *d_Z;
cudaMalloc((void**)&d_Z, sizeof(double)*N);
cudaMemcpy(d_Z, Z, sizeof(double)*N, cudaMemcpyHostToDevice);

// Launch kernel to compute sums of integrands.
// Can CUDA do automatic reductions ???

// Copy sum of integrands back to CPU
cudaMemcpy(&sum, ???, sizeof(double), cudaMemcpyDeviceToHost);

double integral = sum / N;
delete Z;
cudaFree(d_Z);
```

- Use `cudaMalloc` + `cudaMemcpy` to copy Z to GPU
- Customary prefix `d_` on GPU vars, as aid to programmer
- `cudaFree` releases GPU memory so can be reused
CUDA follows SIMT approach

- Not fork-join model of OpenMP. Fixed number of threads are in flight for entire duration of kernel.
- You write code from point of view of single thread, so the program you write looks serial.
- This code is run concurrently by many different threads. Each thread executes the same program, but operates on different data.
- Each thread has its own number called the thread index which it can use to figure out which data elements to operate on.

How is the concurrent execution achieved in hardware?
CUDA Fundamentals – SIMT

CUDA and vectors

• Each CUDA thread is mapped to **ONE** element of a vector unit of length 32

• Group of 32 consecutive threads is called a **warp**, and so one warp represents a **full vector unit**

• Entire warp is processed at once by hardware. So **every line of CUDA code** you write is “**like SSE intrinsics**”: you are programming a vector unit of length 32

• There are multiple vector units, and multiple warps can be processed at the same time
So all threads in a warp **have** to do the same thing?

- Warps are a HARDWARE concept: you can ignore them when you write your code.
- Threads in the same warp **can** execute **entirely different programs** – hardware will give the correct result.
- However this can hurt performance, especially in compute-heavy sections of code.
- We’ll talk more about this later on.
CUDA Fundamentals – Hardware
CUDA Fundamentals – Kernels

- Have CUDA **threads**. At hardware level, threads grouped into **warps**, which map onto vector units.
- Threads are also grouped into **thread blocks**
  - Have up to 1024 threads per block – hardware limit
  - All blocks have same number of threads, fixed at kernel launch
- **Only threads within a thread block can communicate!**
- Blocks cannot communicate with each other, even if two happen to run on the same SM
- Blocks and threads are scheduled (run) in any order, as hardware schedulers see fit!
CUDA Fundamentals – Scalability

CUDA programming model is scalable

- Cheap GPUs = few SMs, can only run few thread blocks concurrently
- Expensive GPUs = many SMs, can run many thread blocks concurrently
- Breaking kernel into independent blocks ensures scalability – on bigger hardware, you just launch more blocks and benefit from linear speedup: 2x more SMs equals 2x faster runtime

Questions on any of this before we return to code?
Recall our OpenMP code:

```c
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);
double sum = 0.0;
#pragma omp parallel for shared(N,Z,T,sigma,r,K,S0) reduction(+:sum) {
  for(int i=0; i<N; i++) {
    double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqrt(T)*Z[i] );
    double x = (ST > K ? ST-K : 0.0);
    sum += exp(-r*T)*x;
  }
}
double integral = sum / N;
delete Z;
```

CUDA parallelisation strategy:

- Spawn number of threads. Each thread computes several iterations and maintains own running sum
- CUDA can’t do automatic reductions, so running sums will be written to GPU memory, with final reduction on CPU
CUDA – First Steps

Recall our OpenMP code:

```c
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);
double sum = 0.0;
#pragma omp parallel for shared(N,Z,T,sigma,r,K,S0) reduction(+:sum) {
    for(int i=0; i<N; i++) {
        double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqrt(T)*Z[i] );
        double x = (ST > K ? ST-K : 0.0);
        sum += exp(-r*T)*x;
    }
}
double integral = sum / N;
delete Z;
```

CUDA kernel launch parameters:

- **Nblks**, **nthds** and **nloops** with **nblks*****nthds*****nloops** = **N**
- In practice, we **overlaunch**: **nblks*****nthds*****nloops** > **N**
- In kernel, make sure do **exactly the right** amount of work
CUDA – First Kernel

```c
__global__ void mcLoopCalc(double T, double r, double S0, double sigma,
                           double K, int N, int nloops, double *d_Z, double *d_sum)
{
    int thdsPerBlk = blockDim.x; // Number of threads per block
    int blkIdx = blockIdx.x; // Index of “this” thread block
    int thdIdx = threadIdx.x; // Index of “this” thread within the block
    int zIdx = blkIdx*thdsPerBlk*nloops + thdIdx;

    double sum = 0.0; double sqT = sqrt(T);
    for(int i=0; i<nloops && zIdx<N; i++) {
        double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqT*d_Z[zIdx] );
        double x = (ST>K ? ST-K : 0.0);
        sum += exp(-r*T)*x; zIdx += thdsPerBlk; // For effective cache use
    }

    int thdIdxKrnl = blkIdx*thdsPerBlk + thdIdx;
    d_sum[thdIdxKrnl] = sum;
}
```

SIMT: single program run by many threads and blocks
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);

double *d_Z, *d_sum;
cudaMalloc((void**)&d_Z, sizeof(double)*N);
// Copy random numbers to GPU
cudaMemcpy(d_Z, Z, sizeof(double)*N, cudaMemcpyHostToDevice);

int nthds = 672, nblks = 140;
cudaMalloc((void**)&d_sum, sizeof(double)*nthds*nblks);
int nloops = ceil((float)N / (nthds*nblks));

// Launch GPU kernel
mcLoopCalc<<<nblks, nthds>>>(T, r, S0, sigma, K, N, nloops, d_Z, d_sum);

// Copy partial sums back to CPU
double *sum = new double[nthds*nblks];
cudaMemcpy(sum, d_sum, sizeof(double)*nthds*nblks, cudaMemcpyDeviceToHost);

for(int i=1; i<nthds*nblks; i++)
    sum[0] += sum[i];

double integral = sum[0] / N;
delete Z; delete sum;
cudaFree(d_Z); cudaFree(d_sum);
So how to resolve our juggling conundrum?

• How many threads? How many thread blocks?

A priori – we simply don’t know

• Either thumb-suck and hope for the best ...

• ... or leave as variables which are **optimised** once the kernel is finished (**auto-tuning**)

• I just try many different combinations of threads and blocks and pick the best one: takes few minutes and gives pretty robust results

Have a slide which shows how to do this, if interested
Monte-Carlo Example – Timings

- **Serial CPU code**
  - Loop time = 1157ms
- **First OpenMP code**
  - Loop time = 337ms
  - Speedup of 3.4x vs serial loop
- **First CUDA Kernel**
  - Loop time = 32.52ms (including CPU reduction)
  - Speedup of 36.1x vs serial loop
CUDA – Execution in More Detail

```
double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqT*d_Z[zIdx] );
...```

- Each instruction is vector (or SIMD). Threads 0 to 31 are grouped in a warp, similarly threads 32 to 63, etc.
- Warp is selected and processed in **lock-step** until it **stalls**: e.g. hits load instruction and waits for data from memory.
- Memory engines operate async. While data fetched, another warp selected and processed until it stalls.
- And so on. When first warp’s data arrives, it is marked as ready and scheduler can select it when next warp stalls.
- More threads in flight=more warps=better latency hiding
- No cost to switching between warps!
What happens on conditionals?

- Recall each thread has different value of `zIdx`
- If all threads in a warp fail conditional, the body is not executed. If at least one thread passes the conditional, the entire warp is processed (remember, vector code!) but results from failing threads are discarded. Hardware handles this transparently and as efficiently as possible.

```c
if(zIdx < N) {
    double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqT*d_Z[zIdx] );
    ...
}
```
CUDA – Execution in More Detail

double x = d_Z[zIdx];
if (x > 0.0) x = exp(x);
else x = sin(x);

Suppose had code like this

- Each thread in warp gets different value of x:

  ```
  if(x > 0.0)
  ```

- Half a warp can’t do \(\exp(x)\) while other half does \(\sin(x)\)
  - Same instruction executed for all threads in a warp – vectors!
- Effectively both \(\exp(x)\) and \(\sin(x)\) are computed, and each thread selects correct value based on conditional
- Hardware handles this **warp divergence** transparently and as efficiently as possible, but will hurt performance!
CUDA – Making Threads Cooperate

- Clearly doing full reduction on host is inefficient
  - Threads in same block can communicate. We want each block to compute running sum and write to global memory
  - CPU then does final reduction over all blocks
  - Reduces by factor `nthds` (672x) amount of data copied from GPU and amount of work done by CPU

- Key is **shared memory**
  - Each SM has 48KB on-chip shared memory. Threads in a block can use this for communication
  - Each thread writes its partial sum to a shared memory array, and then all threads synchronize

- Now could do parallel reduction, but we’ll just do linear
CUDA – Second Kernel

```c
__global__ void mcLoopCalc(double T, double r, double S0, double sigma, double K,
                     int N, int nloops, double *d_Z, double *d_sum)
{
    int thdsPerBlk = blockDim.x; // Number of threads per block
    int blkIdx = blockIdx.x; // Index of “this” thread block
    int thdIdx = threadIdx.x; // Index of “this” thread within the block
    int zIdx = blkIdx*thdsPerBlk*nloops + thdIdx;

    "__shared__ double sums[thdsPerBlk];"

    double sum = 0.0; double sqT = sqrt(T);
    for(int i=0; i<nloops && zIdx<N; i++) {
        double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqT*d_Z[zIdx] );
        double x = (ST>K ? ST-K : 0.0);
        sum += exp(-r*T)*x;
        zIdx += thdsPerBlk;
    }
    sums[thdIdx] = sum;
    __syncthreads();
    if(thdIdx == 0) {
        for(int i=1; i<thdsPerBlk; i++) {
            sums[0] += sums[i];
        }
        d_sum[blkIdx] = sums[0];
    }
}
```
CUDA – Second Kernel

```c
__global__ void mcLoopCalc(double T, double r, double S0, double sigma, double K, int N, int nloops, double *d_Z, double *d_sum)
{
    int thdsPerBlk = blockDim.x; // Number of threads per block
    int blkIdx = blockIdx.x;     // Index of "this" thread block
    int thdIdx = threadIdx.x;    // Index of "this" thread within the block
    int zIndex = blkIdx*thdsPerBlk*nloops + thdIdx;

    extern __shared__ double sums[];

    double sum = 0.0; double sqT = sqrt(T);
    for(int i=0; i<nloops && zIndex<N; i++) {
        double ST = S0*exp( (r-0.5*sigma*sigma)*T + sigma*sqT*d_Z[zIndex] );
        double x = (ST>K ? ST-K : 0.0);
        sum += exp(-r*T)*x;
        zIndex += thdsPerBlk;
    }
    sums[thdIdx] = sum;
    __syncthreads();
    if(thdIdx == 0) {
        for(int i=1; i<thdsPerBlk; i++) {
            sums[0] += sums[i];
        }
        d_sum[blkIdx] = sums[0];
    }
}
```
double * Z = new double[N];
g05skc(N, 0.0, 1.0, genState, Z);

double *d_Z, *d_sum;
cudaMalloc((void**) &d_Z, sizeof(double)*N);

// Copy random numbers to GPU
cudaMemcpy(d_Z, Z, sizeof(double)*N, cudaMemcpyHostToDevice);

int nthds = 672, nblks = 140;
cudaMalloc((void**) &d_sum, sizeof(double)*nblks);
int nloops = ceil( (float)N / (nthds*nblks) );
size_t shmem = sizeof(double)*nthds;

// Launch GPU kernel
mcLoopCalc<<<nblks, nthds, shmem>>>(T, r, S0, sigma, K, N, nloops, d_Z, d_sum);

// Copy partial sums back to CPU
double *sum = new double[nblks];
cudaMemcpy(sum, d_sum, sizeof(double)*nblks, cudaMemcpyDeviceToHost);

for(int i=1; i<nblks; i++)
    sum[0] += sum[i];

double integral = sum[0] / N;

delete Z; delete sum; cudaFree(d_Z); cudaFree(d_sum);
Monte-Carlo Example – Timings

- Serial CPU code
  - Loop time = 1157ms

- First OpenMP code
  - Loop time = 337ms
  - Speedup of 3.4x vs serial loop

- First CUDA Kernel
  - Loop time = 32.52ms (including CPU reduction)
  - Speedup of 35.5x vs serial loop

- Second CUDA Kernel
  - Loop time = 32.12ms (including CPU reduction)
  - Speedup of 36.01x vs serial loop
Monte-Carlo Example – Timing Analysis

- **Serial CPU code**
  - Generate Normals = 5900ms (83.7% of total runtime)
  - Loop time = 1157ms (16.3% of total runtime)

- **First OpenMP code: speedup of application vs serial code = 1.13x**
  - Generating Normals = 5900ms (94.6% of total runtime)
  - Loop time = 337ms (5.4% of total runtime)

- **First CUDA Kernel: speedup of application vs serial code = 1.16x**
  - Generating Normals = 5900ms (96.9% of total runtime)
  - Copy to GPU = 155ms (2.5% of total runtime)
  - Kernel only = 32.01ms (0.5% of total runtime)
  - Copy and CPU sum = 0.57ms (0.009% of total runtime)

- **Second CUDA Kernel: speedup of application vs serial code = 1.16x**
  - Generating Normals = 5900ms (97.5% of total runtime)
  - Copy to GPU = 155ms (2.5% of total runtime)
  - Kernel only = 32.09ms (0.53% of total runtime)
  - Copy and CPU sum = 0.05ms (0.00% of total runtime)
Generating Random Numbers in Parallel

► Is it possible?
  • Yes, but it is not a simple matter
  • Need to know a lot about theory and implementation of base generator algorithms – revolves around “skip-aheads”

► Support in NAG Library?
  • NAG Library (CPU) offers skip-ahead support for several popular generators

► If I use these, what does my CPU code look like?
Monte-Carlo example – Second OpenMP

```c
double * Z = new double[N];
double sum = 0.0;
#pragma omp parallel shared(N,Z,T,sigma,r,K,S0) reduction(+:sum)
{
    int genstate[640] // RNG state private to each thread
g05kfc(..., genstate); // Initialise the generator

    int nthds = omp_get_num_threads(); // Total number of threads
    int thdIdx = omp_get_thread_num(); // Index of “this” thread
    int nloops = N/nthds; // Won’t divide perfectly
    if(thdIdx==nthds-1) // Last thread has to do more work
        nloops = N - nloops*(nthds-1);

    int skip = thdIdx * nloops; // Compute skip-ahead for thread
    g05kjc(skip, genstate); // Skip this thread’s state ahead
    g05skc(nloops, 0.0, 1.0, genState, &Z[skip]);

    for(int i=0; i<nloops; i++) {
        double ST = S0*exp((r-0.5*sigma*sigma)*T + sigma*sqrt(T)*Z[skip+i]);
        double x = (ST > K ? ST-K : 0.0);
        sum += exp(-r*T)*x;
    }
} /* Looks quite a bit like CUDA, doesn’t it? */

double integral = sum / N;
delete Z;
```
Monte-Carlo Example – Timing Analysis

- **Serial CPU code**
  - Generate Normals = 5900ms (83.7% of total runtime)
  - Loop time = 1157ms (16.3% of total runtime)

- **First OpenMP code**: speedup of *application* vs serial code = 1.13x
  - Generating Normals = 5900ms (94.6% of total runtime)
  - Loop time = 337ms (5.4% of total runtime)

- **Second OpenMP code**: speedup of *application* vs serial code = 3.4x
  - Generating Normals = 1737ms (83.7% of total runtime)
  - Loop time = 337ms (16.3% of total runtime)
If random numbers can be generated in parallel on CPU, can they be generated on a GPU?

- **YES!** But not so simple to write robust, general-purpose kernel.
- NAG has library of pre-written kernels which generate random numbers from several distributions
- And they are really easy to use!
- Philosophy is simple: Let NAG write the difficult things (RNGs, linear algebra, base PDE components, ...)
- Call our library to do these operations on GPU. Frees you to focus on parallelising your own code without having to parallelise the “library components” your code depends on
double * Z = new double[N];
double *d_Z, *d_sum;
cudaMalloc((void**)&d_Z, sizeof(double)*N);

g05skc(N, 0.0, 1.0, genState, Z);
// Copy random numbers to GPU
cudaMemcpy(d_Z, Z, sizeof(double)*N, cudaMemcpyHostToDevice);

int nthds = 672, nblks = 140;
cudaMalloc((void**)&d_sum, sizeof(double)*nblks);
int nloops = ceil((float)N / (nthds*nblks));
size_t shmem = sizeof(double)*nthds;

// Launch GPU kernel
mcLoopCalc<<<nblks, nthds, shmem>>>(T, r, S0, sigma, K, N, nloops, d_Z, d_sum);

// Copy partial sums back to CPU
double *sum = new double[nblks];
cudaMemcpy(sum, d_sum, sizeof(double)*nblks, cudaMemcpyDeviceToHost);

for(int i=1; i<nblks; i++)
    sum[0] += sum[i];

double integral = sum[0] / N;

delete Z; delete sum; cudaFree(d_Z); cudaFree(d_sum);
Monte-Carlo Example – Calling Kernel 2

```c
double * Z = new double[N];
double *d_Z, *d_sum;
cudaMalloc((void**)&d_Z, sizeof(double)*N);

// Call NAG GPU generator
naggpuRandNormal(N, 0.0, 1.0, d_Z, genState);

int nthds = 672, nblks = 140;
cudaMalloc((void**)&d_sum, sizeof(double)*nblks);
int nloops = ceil( (float)N / (nthds*nblks) );
size_t shmem = sizeof(double)*nthds;

// Launch GPU kernel
mcLoopCalc<<<nblks, nthds, shmem>>>(T, r, S0, sigma, K, N, nloops, d_Z, d_sum);

// Copy partial sums back to CPU
double *sum = new double[nblks];
cudaMemcpy(sum, d_sum, sizeof(double)*nblks, cudaMemcpyDeviceToHost);

for(int i=1; i<nblks; i++)
    sum[0] += sum[i];

double integral = sum[0] / N;

delete Z; delete sum; cudaFree(d_Z); cudaFree(d_sum);
```
Monte-Carlo Example – Timing Analysis

- Serial CPU code
  - Generate Normals = 5900ms (83.7% of total runtime)
  - Loop time = 1157ms (16.3% of total runtime)
- First OpenMP code: speedup of application vs serial code = 1.13x
  - Generating Normals = 5900ms (94.6% of total runtime)
  - Loop time = 337ms (5.4% of total runtime)
- Second OpenMP code: speedup of application vs serial code = 3.4x
  - Generating Normals = 1737ms (83.7% of total runtime)
  - Loop time = 337ms (16.3% of total runtime)
- CUDA + NAG GPU: speedup of application vs serial code = 105.3x
  - Generating Normals = 34.92ms (52.09% of total runtime)
  - Kernel+CPU reduction = 32.12ms (47.91% of total runtime)
  - That’s 31x faster than second OpenMP code!
ACCESS TO NAG SOFTWARE
University of Manchester Site Licence

- Unlimited use for the Linux, Mac, Solaris and Windows
  - As long as for academic or research purposes
  - Installation may be on any university, staff or student machine

- Products
  - All NAG Libraries: Fortran, C, SMP, NAG Fortran Compiler
  - Also NAG Numerical Components for GPUs (CUDA)

- Full access to NAG Support support@nag.co.uk
  - Request support or licences using university e-mail xxxx@xxxx.ac.uk
  - Full licence keys via applicationsupport-eps@manchester.ac.uk

- NAG software:
  - Includes online documentation - also www.nag.co.uk
  - Supplied with extensive example programs (input data and results)
How do I get the software?

► Contact us or help yourself from
  • [http://www.nag.co.uk/downloads/index.asp](http://www.nag.co.uk/downloads/index.asp)

► Via NAG / University of Manchester list to get permanent licence keys:
  • [applicationsupport-eps@manchester.ac.uk](mailto:applicationsupport-eps@manchester.ac.uk)
  • Anything extra / unusual, [Michael.Croucher@manchester.ac.uk](mailto:Michael.Croucher@manchester.ac.uk)
  • Mike is also author of [www.walkingrandomly.com](http://www.walkingrandomly.com) where NAG features – good and bad!

► Temporary Licence keys
  • From [support@nag.co.uk](mailto:support@nag.co.uk)

► UK Academic Account Manager
  • [louise.mitchell@nag.co.uk](mailto:louise.mitchell@nag.co.uk)
NAG Numerical Routines for GPUs

- **Random number generators**
  - MRG32k3a and MT19937
  - Uniform, Exponential, Normal and Gamma distributions
  - Brownian bridge
  - MRG32k3a “device function” generators can be embedded in your kernel

- **Quasi-random number generators**
  - Sobol in up to 50,000 dimensions
  - Hickernell scrambling
  - Uniform, Exponential, Normal distributions (inv. CDF)

- **Linear Algebra**
  - Cholesky factorisation
  - LU decomposition

- **Full error checking, thread safe (for GPU clusters), highly tuned for**
  - C1060/ M1060 and C2050/M2050/M2070 cards
Getting Access to the NAG GPU Routines

► To get access, simply email
  • Louise.mitchell@nag.co.uk
  • Use your university email address xxxx@xxxx.ac.uk
  • Academic use is free subject to signing a collaborative agreement
  • Full support is provided, same as for NAG products

► GPU routines implemented for Linux and Windows
  • 32bit and 64bit
  • Full documentation and examples
NAG GPU Routines – Future Content

- Random number generators
  - Additional distributions (non-central Chi-square, NIG, VG, Stable distributions)

- Linear Algebra
  - Singular Value Decomposition
  - Solvers

- PDEs
  - ADI solver for general second order PDEs
  - Heston model in 3 and 4 dimensions
  - Building blocks for fully customisable PDE solver: grid, stencil, assembly, ...

- Optimisation

- ... and whatever else our customers need
  - Please talk to us about your requirements!
Thank You

Further CUDA reading and training

- Plenty of free NVIDIA webinars
- CUDA C Programming Guide – indispensable!
- Programming Massively Parallel Processors (Hwu, Kirk)
- NAG 2 day HECToR training course for CUDA
- Bespoke training offered by NAG

Any questions?

- jacques@nag.co.uk and info@nag.co.uk
Questions?