Monte Carlo implementations on GPUs

David B. Thomas
Imperial College
dt10@doc.ic.ac.uk

Who am I?

- Research fellow at Imperial
  - Software Engineering and FPGA background
  - Lead a small group looking at accelerated computational finance
- What do I have to do with GPUs or finance?
  - Most of my work: tools and methods for FPGA-based finance
  - Compare performance of FPGA, CPU, and now GPU
    - Initial CPU solution: day or so
    - Develop FPGA solution: couple of weeks or months
    - GPU solutions (keep paper reviewers happy): couple of days
  - Usually find FPGA and GPU about the same performance
    - GPU: 10x developer productivity; FPGA 10x more power efficient

Who am I?

- Research fellow at Imperial
  - Software Engineering and FPGA background
  - Lead a small group looking at accelerated computational finance
- What do I have to do with GPUs or finance?
  - Most of my work: tools and methods for FPGA-based finance
  - Compare performance of FPGA, CPU, and now GPU
    - Initial CPU solution: day or so
    - Develop FPGA solution: couple of weeks or months
    - GPU solutions (keep paper reviewers happy): couple of days
  - Usually find FPGA and GPU about the same performance
    - GPU: 10x developer productivity; FPGA 10x more power efficient
- NVidia guy: “Why are you still wasting time with FPGAs”?
  - I’m an academic: want to look at the hard(-ish) unsolved problems
  - GPUs are mainstream: anyone can do it (that’s why you are here)

Who are you?

- I have no idea – my guesses about you
  - Interested in, or actively working in financial modelling
  - Are a programmer in some sense (this is a hands on workshop)
  - Know something about CUDA/GPUs, but are not an expert
    - Apologies if you have *no* knowledge about CUDA or GPUs
    - Sorry if you are a hard-core expert: if you are, why aren't *you* talking?
  - Wondering whether to use GPUs, or how to use them better
- My guesses about what you might want to hear
  1. General experiences with GPU Monte-Carlo: random (ha-ha!) tips
  2. Specific things to watch out for: performance and correctness
  3. Hard-core optimisation: new uniform random number generator
- What you won't hear
  - Anything specific about pricing models or finance
  - Not enough time; everyone does something different

What is a GPU?

- Means different things to different people
  1. Something that was originally developed for use in graphics?
  2. Something made by NVidia that runs CUDA?
  3. A wide SIMD processor using threads to hide latency?
  4. A hardware accelerator that supports OpenCL?

What is a GPU?

- Means different things to different people
  1. Something that was originally developed for use in graphics?
  2. Something made by NVidia that runs CUDA?
  3. A wide SIMD processor using threads to hide latency?
  4. A hardware accelerator that supports OpenCL?
- For the purposes of this talk: option 2
  - CUDA is ahead of the competition in terms of tools
  - Everyone else here will talk CUDA/NVidia
- In a couple of years time (hopefully): option 4
  - NVidia deserve huge credit for developing and promoting CUDA
  - But... you are the end-users: seek portability, don’t get locked in
    - FPGA accelerators existed for 10 years: no portability, no market
  - Encourage NVidia/AMD/Intel to compete on hardware

GPU: Central concepts

- CPUs devote very little silicon area to actual computation
  - Most of the area is trying to make ***sequential*** code faster
  - Cache: decrease latency, increase bandwidth
  - Branch prediction/speculation: decrease the cost of branches
- GPUs devote as much area as possible to computation
  - Stick as many floating-point units on as possible
  - Get rid of the huge caches and super-scalar stuff
- Manage latency by building multi-threading in at low level
  - GPU memory latency is similar to CPU: still have to deal with it
  - Have thousands of active threads in one processor
  - If one thread stalls on memory, schedule the next one

GPU: Threading

- Threads are grouped into warps
  - Warp size is currently 32 threads
  - Threads never change their warp
    - Assigned to warps using threadIdx

```
__global__
void MyKernel(
  unsigned *pMem
){
  int wIdx=tIdx.x/32;
  int wOff=tIdx.x-32*wIdx;

  if(Condition()){
    DoOneThing();
  }else{
    DoOtherThing();
  }

  int addr=
         wIdx*32+((wOff+1)%32);
  pMem[addr]=Something();
}
```

GPU: Threading

- Threads are grouped into warps
  - Warp size is currently 32 threads
  - Threads never change their warp
    - Assigned to warps using threadIdx
- Warps are important for compute efficiency
  - One thread branches -> warp branches
  - Threads take different branches: divergence
  - Ideally: all threads in warp take same branch
    - No divergence, better performance

```
__global__
void MyKernel(
  unsigned *pMem
){
  int wIdx=tIdx.x/32;
  int wOff=tIdx.x-32*wIdx;

  if(Condition()){
    DoOneThing();
  }else{
    DoOtherThing();
  }

  int addr=
        wIdx*32+((wOff+1)%32);
  pMem[addr]=Something();
}
```

GPU: Threading

- Threads are grouped into warps
  - Warp size is currently 32 threads
  - Threads never change their warp
    - Assigned to warps using threadIdx
- Warps are important for compute efficiency
  - One thread branches -> warp branches
  - Threads take different branches: divergence
  - Ideally: all threads in warp take same branch
    - No divergence, better performance
- Warps are important for memory efficiency
  - Determine global memory coalescing[1]
  - Determine shared memory conflicts[1]

```
__global__
void MyKernel(
  unsigned *pMem
){
  int wIdx=tIdx.x/32;
  int wOff=tIdx.x-32*wIdx;

  if(Condition()){
    DoOneThing();
  }else{
    DoOtherThing();
  }

  int addr=
        wIdx*32+((wOff+1)%32);
  pMem[addr]=Something();
}
```

[1] – Yeah, half-warps, whatever

GPU: Threading

- Threads are grouped into warps
  - Warp size is currently 32 threads
  - Threads never change their warp
    - Assigned to warps using threadIdx
- Warps are important for compute efficiency
  - One thread branches -> warp branches
  - Threads take different branches: divergence
  - Ideally: all threads in warp take same branch
    - No divergence, better performance
- Warps are important for memory efficiency
  - Determine global memory coalescing[1]
  - Determine shared memory conflicts[1]
- Make sure you understand warps!
  - More important than threads
  - Read the user guide (twice)

```
__global__
void MyKernel(
  unsigned *pMem
){
  int wIdx=tIdx.x/32;
  int wOff=tIdx.x-32*wIdx;

  if(Condition()){
    DoOneThing();
  }else{
    DoOtherThing();
  }

  int addr=
         wIdx*32+((wOff+1)%32);
  pMem[addr]=Something();
}
```

[1] – Yeah, half-warps, whatever

Example: Rejection Methods

- Warp divergence hurts performance
  - Scalar code does not take into account
  - CPU algorithms are often divergent
- Rejection: optimise for average case
  - Generate cheap random candidate
    - Simple transform of uniform RNG
  - Check candidate with cheap test
  - Otherwise use a slow alternative
    - May be recursive

```
u=UnifRng();
x=Candidate(u);
if(Accept(x))
 return x;
else
 return Slow();
```

- e.g. Ziggurat method for uniform to Gaussian conversion
  - Fast: one uniform RNG, one comparison, one multiply
  - Slow: looping, exponentials, logs, more uniform RNGs
  - Designed so that fast route is taken ~98% of time
  - The Ziggurat algorithm is a work of art – superb for scalar CPUs

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads

Thread 0	Thread 1	Thread 2	Thread 3

`x=Candidate();`
`if(Accept(x))`
` return x;`
`else`
` return Slow();`
	`x=Candidate();`
	`if(Accept(x))`
	` return x;`
	`else`
	` return Slow();`
		`x=Candidate();`
		`if(Accept(x))`
		` return x;`
		`else`
		` return Slow();`
			`x=Candidate();`
			`if(Accept(x))`
			` return x;`
			`else`
			` return Slow();`

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads

Thread 0	Thread 1	Thread 2	Thread 3

`x=Candidate();`
`if(Accept(x))`
`  return x;`
`else`
`  return Slow();`
	`x=Candidate();`
	`if(Accept(x))`
	`  return x;`
	`else`
	`  return Slow();`
		`x=Candidate();`
		`if(Accept(x))`
		`  return x;`
		`else`
		`  return Slow();`
			`x=Candidate();`
			`if(Accept(x))`
			`  return x;`
			`else`
			`  return Slow();`

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads

Thread 0	Thread 1	Thread 2	Thread 3

`x=Candidate();`
`if(Accept(x))`
` return x;`
`else`
` return Slow();`
	`x=Candidate();`
	`if(Accept(x))`
	` return x;`
	`else`
	` return Slow();`
		`x=Candidate();`
		`if(Accept(x))`
		` return x;`
		`else`
		` return Slow();`
			`x=Candidate();`
			`if(Accept(x))`
			` return x;`
			`else`
			` return Slow();`

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads

Thread 0	Thread 1	Thread 2	Thread 3

`x=Candidate();`
`if(Accept(x))`
` return x;`
`else`
` return Slow();`
	`x=Candidate();`
	`if(Accept(x))`
	` return x;`
	`else`
	` return Slow();`
		`x=Candidate();`
		`if(Accept(x))`
		` return x;`
		`else`
		` return Slow();`
			`x=Candidate();`
			`if(Accept(x))`
			` return x;`
			`else`
			` return Slow();`

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads

Thread 0	Thread 1	Thread 2	Thread 3

`x=Candidate();`
`if(Accept(x))`
`  return x;`
`else`
`  return Slow();`
	`x=Candidate();`
	`if(Accept(x))`
	`  return x;`
	`else`
	`  return Slow();`
		`x=Candidate();`
		`if(Accept(x))`
		`  return x;`
		`else`
		`  return Slow();`
			`x=Candidate();`
			`if(Accept(x))`
			`  return x;`
			`else`
			`  return Slow();`

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads

Thread 0	Thread 1	Thread 2	Thread 3

`x=Candidate();`
`if(Accept(x))`
` return x;`
`else`
` return Slow();`
	`x=Candidate();`
	`if(Accept(x))`
	` return x;`
	`else`
	` return Slow();`
		`x=Candidate();`
		`if(Accept(x))`
		` return x;`
		`else`
		` return Slow();`
			`x=Candidate();`
			`if(Accept(x))`
			` return x;`
			`else`
			` return Slow();`

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads

Thread 0	Thread 1	Thread 2	Thread 3

`x=Candidate();`
`if(Accept(x))`
` return x;`
`else`
` return Slow();`
	`x=Candidate();`
	`if(Accept(x))`
	` return x;`
	`else`
	` return Slow();`
		`x=Candidate();`
		`if(Accept(x))`
		` return x;`
		`else`
		` return Slow();`
			`x=Candidate();`
			`if(Accept(x))`
			` return x;`
			`else`
			` return Slow();`

Example: Rejection Methods

- Economics of rejection break down with GPU style SIMD
  - Threads execute in warps
  - Each thread can take different path through code
  - Time for warp is ***total*** time to cover paths of ***all*** threads
- Rejection relies on low probability of slow path
  - Entire thread group incurs cost of one slow thread
  - Probability of ***each*** thread taking fast path is ~98%
  - Probability of ***all*** 32 threads taking fast path is ~52%
  - Expected execution time: $t_{fast} + 0.48\ t_{slow}$
- Non-rejection algorithms are (usually) better in GPU
  - Has built-in fast log/exp/sin: use Box-Muller method
  - Rational approximations are your friend: very fast

The perils of function approximation

- Simulations need functions with no closed form
  - Standard examples: Gaussian CDF (Phi(x)) and ICDF ($Phi^{-1}(x)$)
- Obvious point[1]: read the documentation, see if it exists
  - CUDA already includes the error function as intrinsics
    - erff, erfcf : p = Phi(x) = erfc[x / -sqrt(2)] / 2
    - erfinvf, erfcinvf : x = $Phi^{-1}(p)$ = erfcinf[2 p] * -sqrt(2)
  - If you're off the critical path, intrinsics are good enough
    - Aside: you would think they would be super fast, but they aren't
- Lets assume we are doing CDF inversion
  - e.g. we are using Quasi-RNGs, or some other variance reduction
  - Inversion: take a uniform 32-bit number u, turn it into Gaussian x
  - Obvious: x = $Phi^{-1}(u * 2^{-32}))$

[1] – Yup, I didn't read the documentation, and wasted time doing my own.

CDF Inversion: simple

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);

  // [0..2^32) -> [0,1)
  float p=u*S1;

  // Phi(x) = -sqrt(2)*erfcinv(2*p)
  return S2*erfcinv(2*p);
}
```

I apologise if this is obvious. Not everyone knows about this stuff.

CDF Inversion: simple, but deceptive

- First problem: lower bound
  - NormalCdfInv(0) = - infinity

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);

  // [0..2^32) -> [0,1)
  float p=u*S1;

  // Phi(x) = -sqrt(2)*erfcinv(2*p)
  return S2*erfcinv(2*p);
}
```

I apologise if this is obvious. Not everyone knows about this stuff.

CDF Inversion: simple, but deceptive

- First problem: lower bound
  - NormalCdfInv(0) = - infinity
- Simple solution: nudge away from 0
  - Add 2^-33 during integer->float conv.

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);
  const float S3=pow(2,-33);

  // [0..2^32) -> (0,1)
  float p=u*S1 + S3;

  // Phi(x) = -sqrt(2)*erfcinv(2*p)
  return S2*erfcinv(2*p);
}
```

Sorry, this is floating-point 101, but not everyone knows about it. For instance, browse the CUDA SDK samples...

CDF Inversion: simple, but deceptive

- First problem: lower bound
  - NormalCdfInv(0) = - infinity
- Simple solution: nudge away from 0
  - Add 2^-33 during integer->float conv.
- Next problem: upper bound
  - NormalCdfInv(2^{32}-1) = infinity
  - Why?
    - $p = \quad u \quad * 2^{-32} + 2^{-33}$
    - $p = (2^{\wedge 32}-1) * 2^{-32} + 2^{-33}$
    - p = 0.99999999988358467
  - But in *single-precision* p=1
- Time to talk about single-precision

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);
  const float S3=pow(2,-33);

  // [0..2^32) -> (0,1)
  float p=u*S1 + S3;

  // Phi^-1(x) = -sqrt(2)*erfcinv(2*p)
  return S2*erfcinv(2*p);
}
```

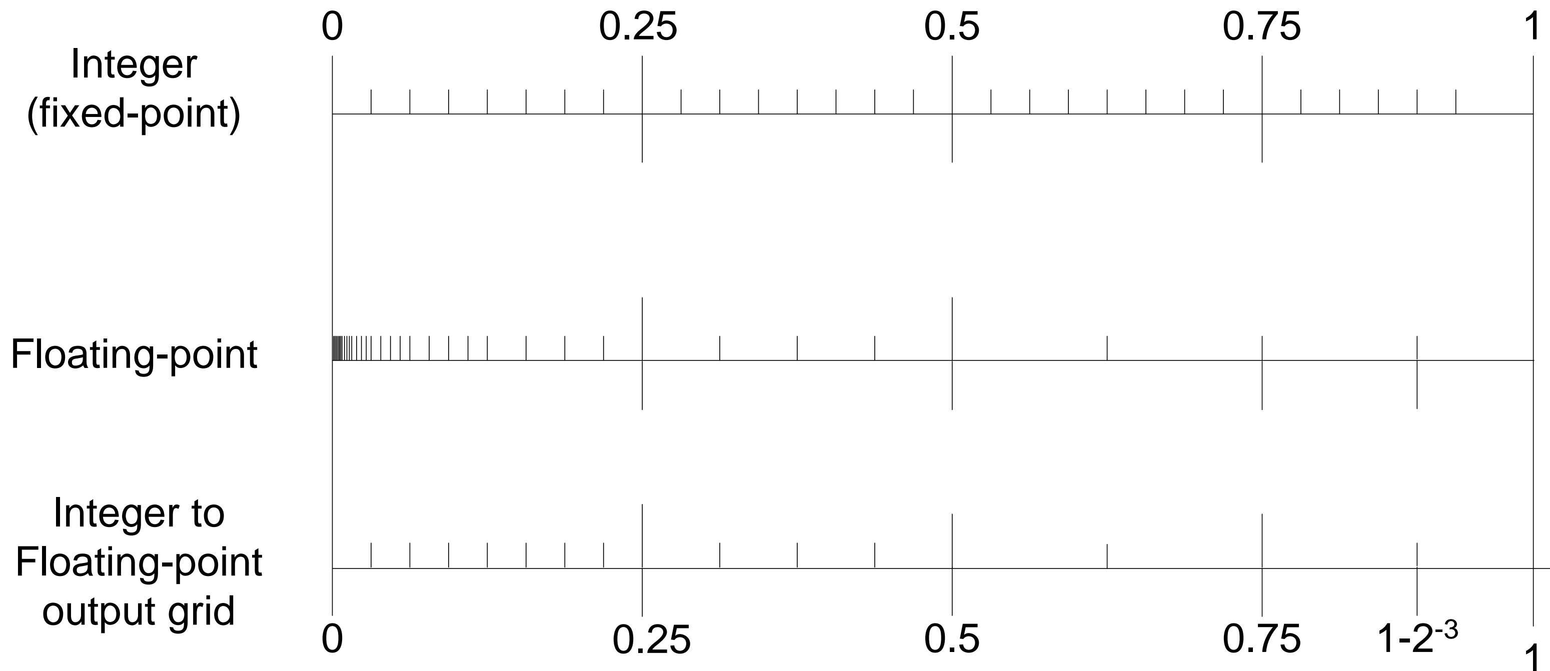
Sorry, this is floating-point 101, but not everyone knows about it. For instance, browse the CUDA SDK samples...

An aside: GPUs and single-precision

- Lets be clear: single-precision is not some kind of flaw
  - It doesn’t make anything impossible
  - It doesn’t mean your answers will automatically be inaccurate
- However, it requires the programmer to think
  - Need a basic understanding of floating-point arithmetic
  - Must understand how numbers are being manipulated
- How much do you care about performance vs. effort?
  - Use double-precision: lower effort, but lower performance
  - Legitimate choice – you don’t *have* to use single precision
- Double-precision will get faster with newer hardware
  - Will it ever be as fast as single-precision? (Maybe it already is?)
  - Even so: still a performance hit from memory - twice the size

Integer to floating-point

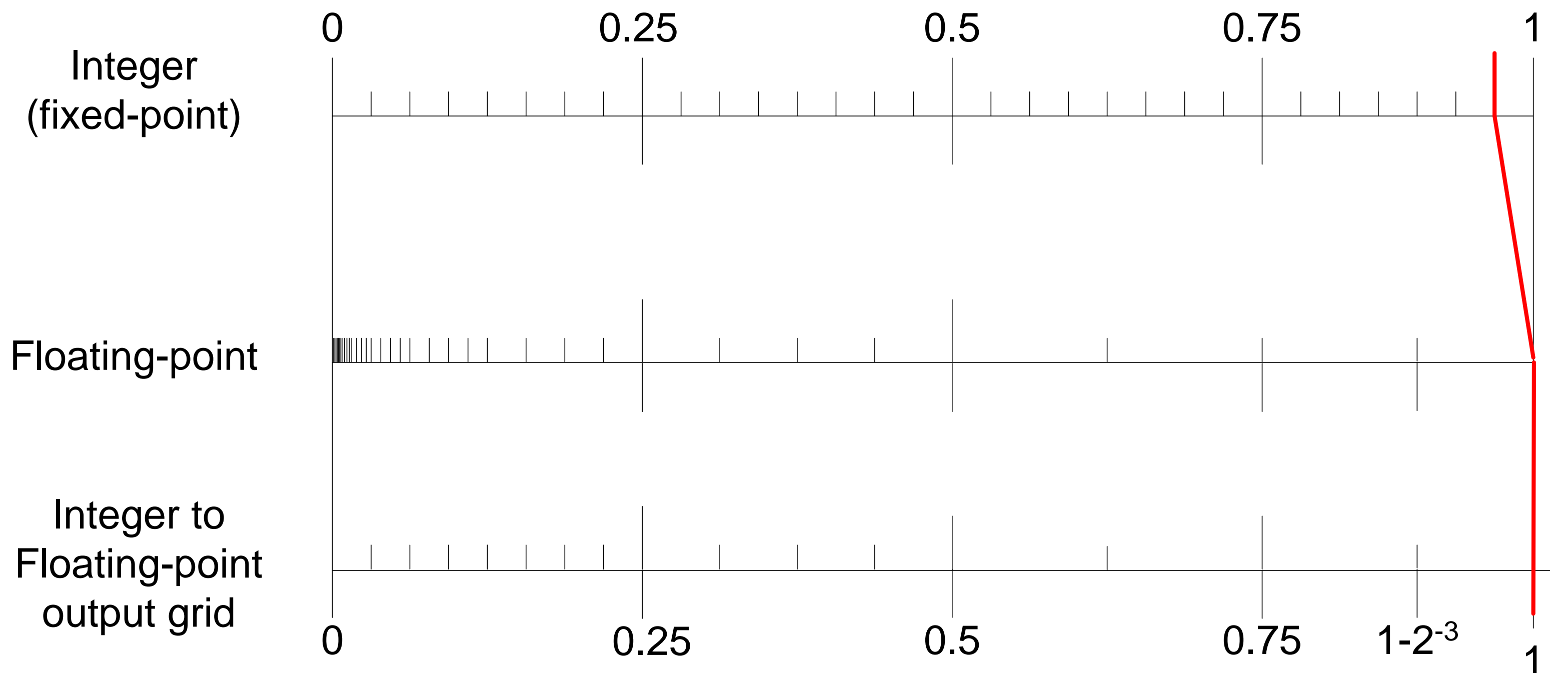
- Fixed-point (integer) and floating-point are for different jobs
  - Floating-point: accuracy relative to magnitude, over infinite[1] range
  - Fixed-point: accuracy independent of magnitude, over finite range



[1] : infinite-ish – there are obviously exponent limits

Integer to floating-point

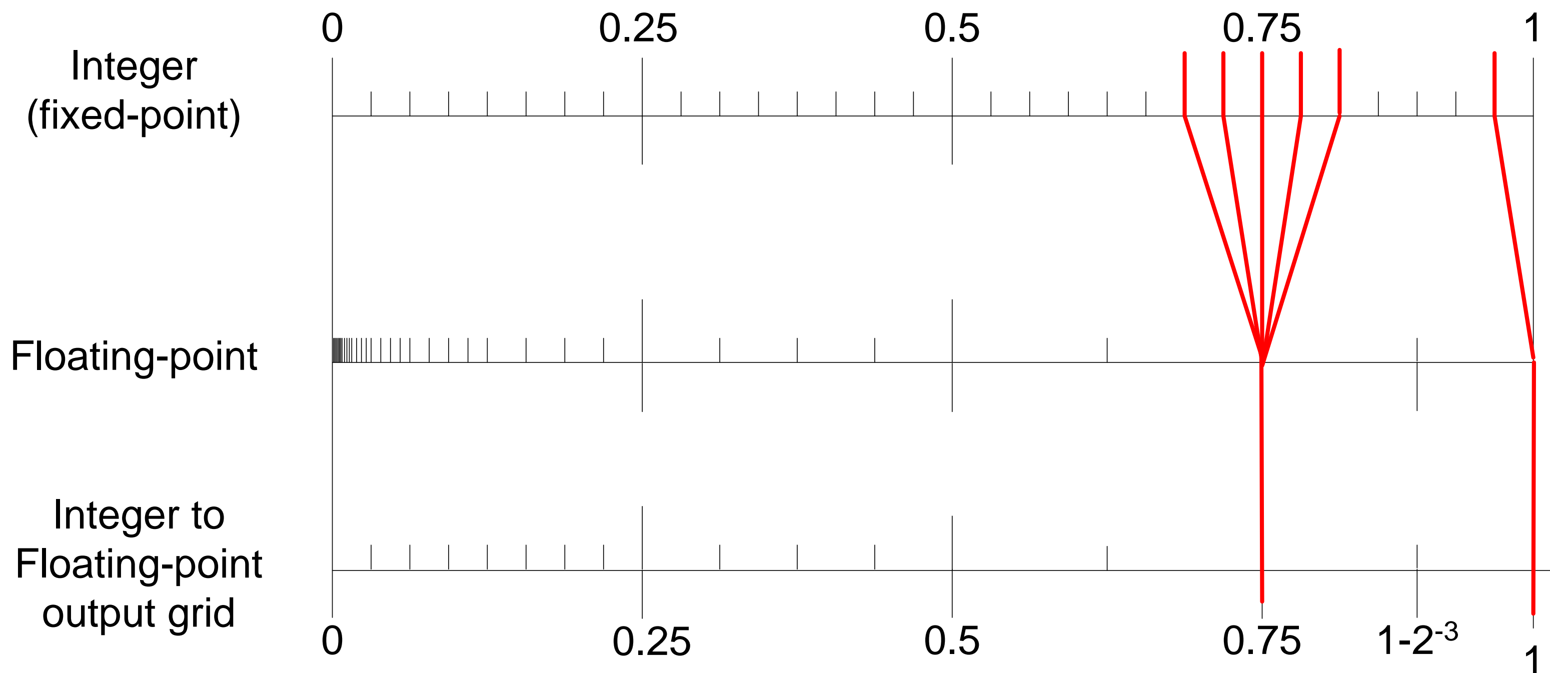
- Fixed-point (integer) and floating-point are for different jobs
  - Floating-point: accuracy relative to magnitude, over infinite[1] range
  - Fixed-point: accuracy independent of magnitude, over finite range



[1] : infinite-ish – there are obviously exponent limits

Integer to floating-point

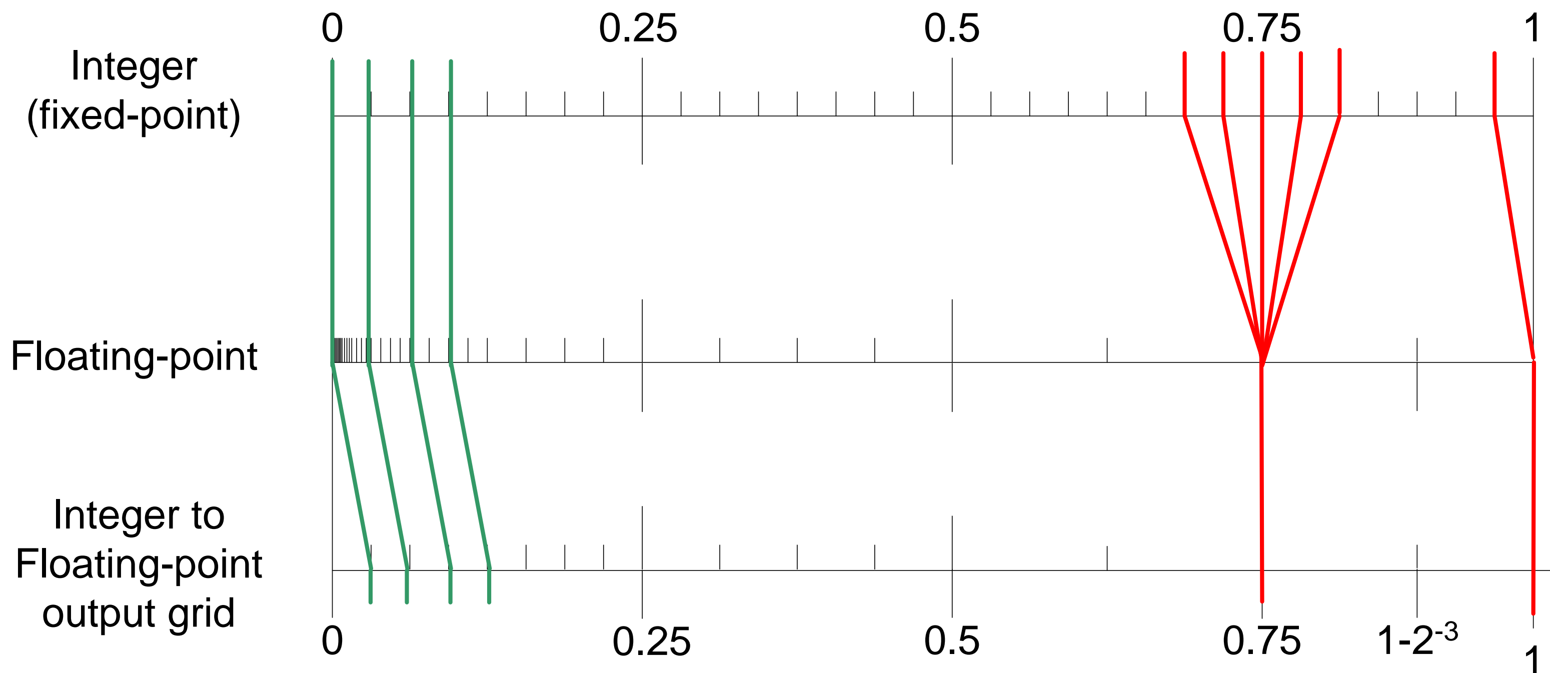
- Fixed-point (integer) and floating-point are for different jobs
  - Floating-point: accuracy relative to magnitude, over infinite[1] range
  - Fixed-point: accuracy independent of magnitude, over finite range



[1] : infinite-ish – there are obviously exponent limits

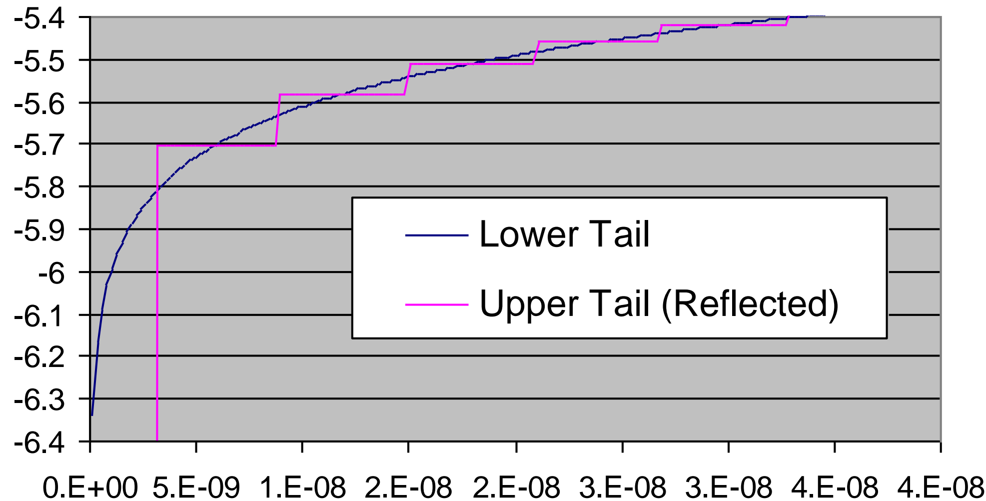
Integer to floating-point

- Fixed-point (integer) and floating-point are for different jobs
  - Floating-point: accuracy relative to magnitude, over infinite[1] range
  - Fixed-point: accuracy independent of magnitude, over finite range



[1] : infinite-ish – there are obviously exponent limits

Back to Inversion



- So the lower (negative) tail is fine, but the upper (positive) tail is not
  - Largest uniform inputs result in infinity – with probability about 2^{-24} !
- Even if we solve the infinities, upper tail is ruined
  - Positive half of distribution is discretised into only 2^{24} values
- This will mess up long-running simulations
  - Distribution is not symmetric – mean will not be zero
  - Higher moments are all slightly disturbed
  - Effects of low-discrepancy sequence reduced in upper half

CDF Inversion: a reasonable solution

- Check whether $p > 0.5$
  - Do it *before* conversion to floating-point

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);
  const float S3=pow(2,-33);

  // [0..2^32) -> (0,1)
  float s = S2;
  if(u>=0x80000000){
    u=0xFFFFFFFF - u;
    s = -S2;
  }

  float p=u*S1 + S3;

  return s*erfcinv(2*p);
}
```

CDF Inversion: a reasonable solution

- Check whether p > 0.5
  - Do it *before* conversion to floating-point
- If p>0.5 then reflect into lower tail
  - Set p = 1-p (still in integer form)
  - Record the saved sign for later

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);
  const float S3=pow(2,-33);

  // [0..2^32) -> (0,1)
  float s = S2;
  if(u>=0x80000000){
    u=0xFFFFFFFF - u;
    s = -S2;
  }

  float p=u*S1 + S3;

  return s*erfcinv(2*p);
}
```

CDF Inversion: a reasonable solution

- Check whether p > 0.5
  - Do it *before* conversion to floating-point
- If p>0.5 then reflect into lower tail
  - Set p = 1-p (still in integer form)
  - Record the saved sign for later
- Keep original nudging solution
  - Still works fine from both ends
- Restore the sign in the final step
  - We had to do a multiply here anyway

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);
  const float S3=pow(2,-33);

  // [0..2^32) -> (0,1)
  float s = S2;
  if(u>=0x80000000){
    u=0xFFFFFFFF - u;
    s = -S2;
  }

  float p=u*S1 + S3;

  return s*erfcinv(2*p);
}
```

CDF Inversion: a reasonable solution

- Performance impact is fairly small
  - Branch can be handled with predication
  - Majority of work is still in erfcinv
  - 6.6 GInv/sec vs. 6.1 GInv/sec
- About 8% perf. loss: is it worth it?
  - No infinities....
  - Output distribution is symmetric
    - Correct mean and odd moments
  - Finest resolution concentrated in tails
    - High variance regions: QRNG effective
    - Even moments more accurate
- If you want the right answer...

```
__device__
float NormalCdfInv(
  unsigned u
){
  const float S1=pow(2,-32);
  const float S2=-sqrt(2);
  const float S3=pow(2,-33);

  // [0..2^32) -> (0,1)
  float s = S2;
  if(u>=0x80000000){
    u=0xFFFFFFFF - u;
    s = -S2;
  }

  float p=u*S1 + S3;

  return s*erfcinv(2*p);
}
```

Beware code in the wild

- Code for quasi-random simulation using inversion
  - From an unnamed source of GPU example code

```
////////////////////////////////////////////////////////////////////////////////
// Moro's Inverse Cumulative Normal Distribution function approximation
////////////////////////////////////////////////////////////////////////////////
#ifndef DOUBLE_PRECISION
__device__ inline float MoroInvCNDgpu(float P){
    const float a1 = 2.50662823884f;
    const float a2 = -18.61500062529f;
    const float a3 = 41.39119773534f;
```

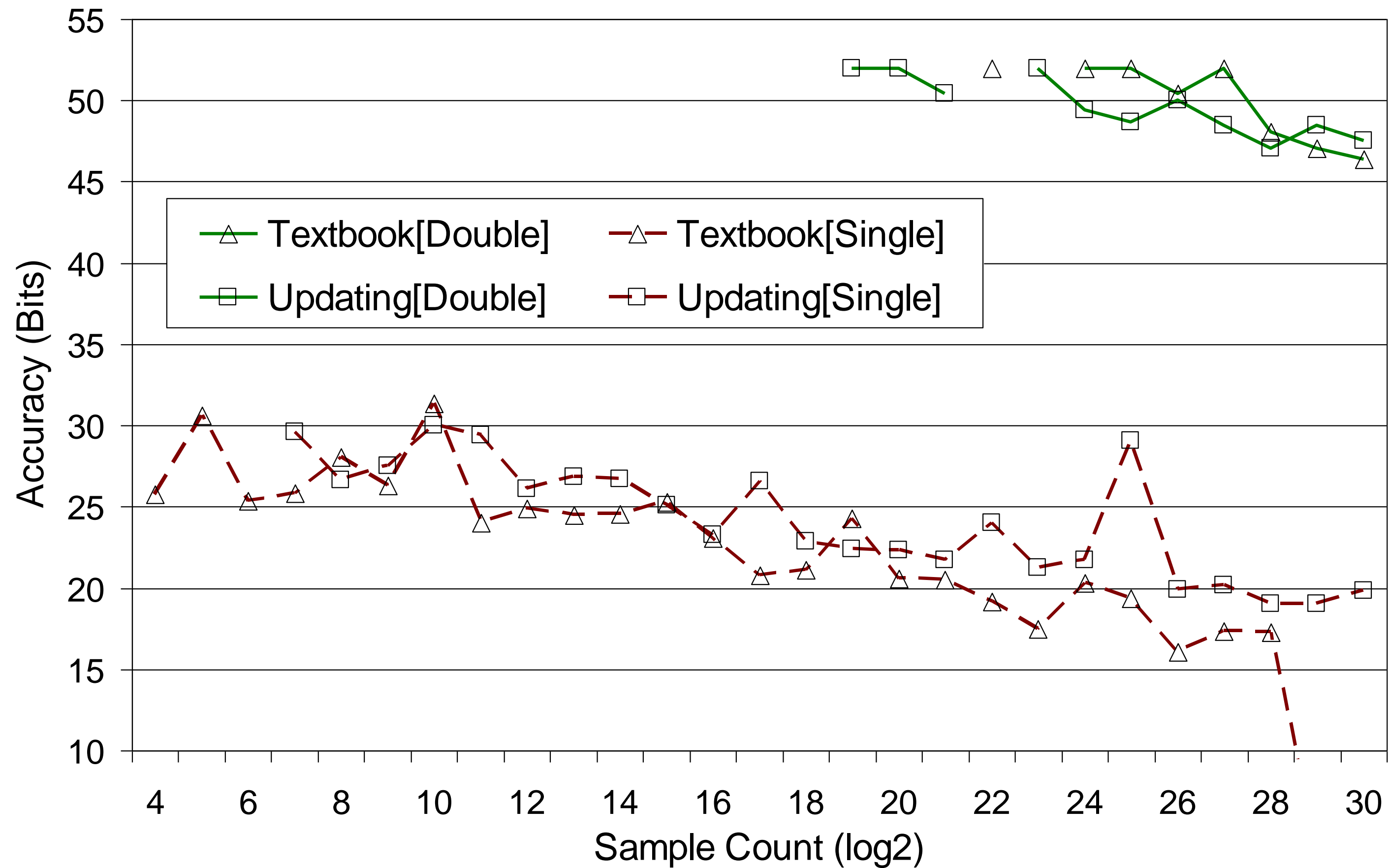
<snip>

```
    float y = P - 0.5f;
    if(fabsf(y) < 0.42f){
        z = y * y;
        z = y * (((a4*z+a3)*z+a2)*z+a1)/((((b4*z+b3)*z+b2)*z+b1)*z+1.0f);
    }else{
        if(y > 0)
            z = __logf(-__logf(1.0f - P));
```

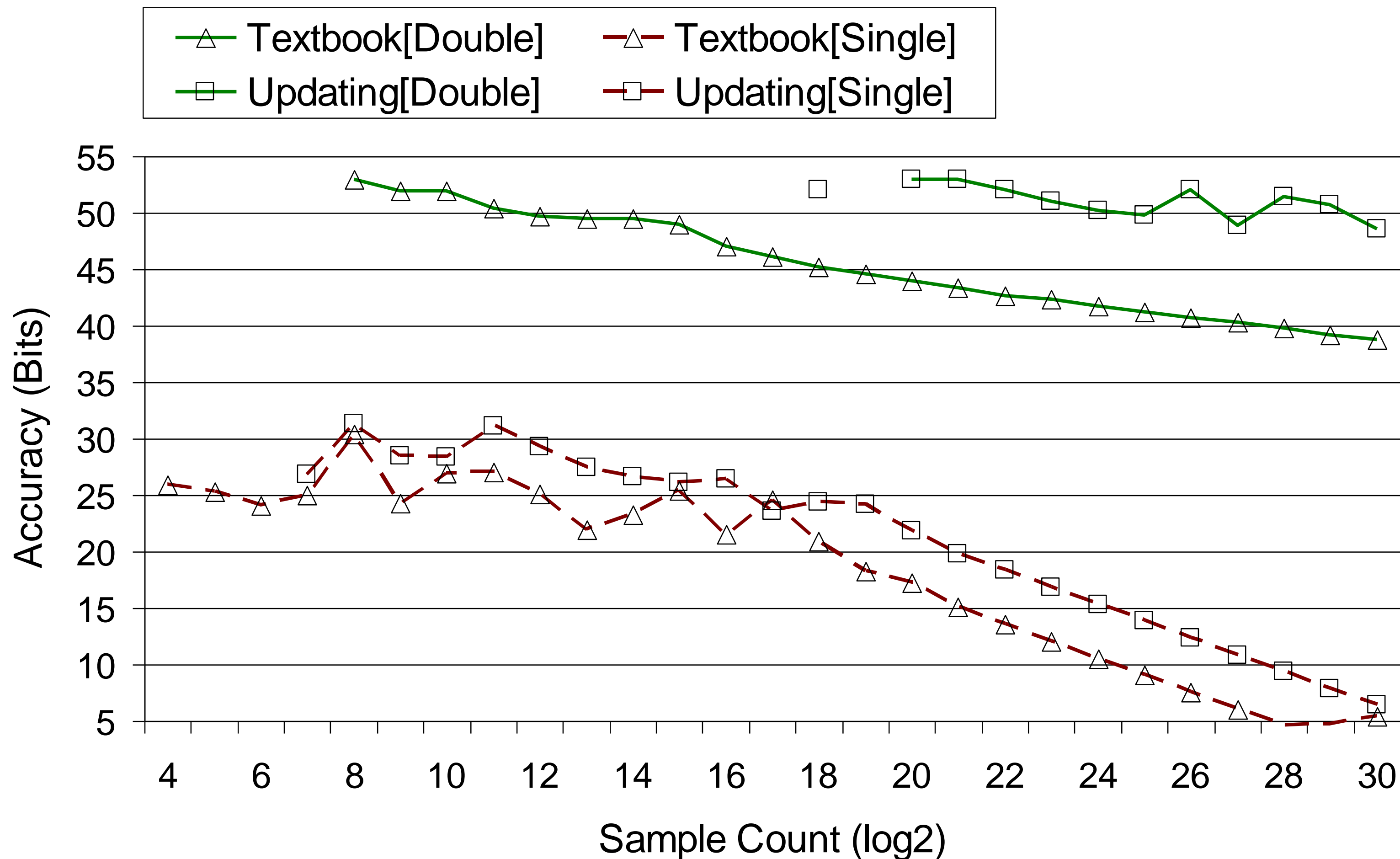
When is single-precision not enough?

- Some situations ***do*** require double-precision
  - Always possible to work around, but not worth the risk and effort
- Running sum over a stream of data
  - Use double-precision when stream is more than ~100-1000
    - Actual threshold is data-dependent: be safe rather than sorry
  - Even though data is single-precision, sum in double-precision
  - Possible exception: can use a Kahan accumulator (but test well!)
- Statistical accumulators: mean and variance
  - ***Always*** calculate means and variance in double-precision
  - Even if n is small now, someone, eventually will say “use $32n$”
- Don’t be seduced by online/updating methods
  - They can be quite useful – in double-precision
  - They don’t really help in single-precision

Single vs. Double: Mean



Single vs Double: Variance



General comments on floating-point

- None of these representation/precision issues are new
  - Occur in high-performance computing all the time
  - Lots of literature out there on safe single-precision
    - *“What Every Computer Scientist Should Know About Floating-Point Arithmetic”,* David Goldberg
- Think laterally: e.g. don’t forget the integers
  - Convert to 32-bit fixed-point (float->uniform + multiply)
  - Sum in 64-bit integer (two instructions: Cheap!)
  - Can add 2^{32} samples exactly, with no overflow
- GPUs can let you do a huge number of simulations
  - Easy to lose track of the magnitude of the result set
  - 2^{32} is not a large number of simulations; 2^{40} is not uncommon
  - Play safe: double-precision for statistical accumulators

Memory

- Two types of memory: shared and global
- Shared memory: small, but fast
  - Can almost treat as registers, with added ability to index
- Global memory: large, but slow
  - Can’t be overstated how slow (comparatively) it is
  - Minimise global memory traffic wherever possible
- Other types of memory are facades over global memory
- Constant memory: caches small part of global memory
  - Doesn’t use global memory bandwidth once it is primed
- Texture memory: caches larger part of global memory
  - Cache misses cause global memory traffic
  - Watch out!

Memory in MC: the buffer anti-pattern

- Beware spurious memory buffers
  - Strange anti-pattern that occurs
  - *I will generate **all** the uniforms*
  - *Then transform **all** the gaussians*
  - *Then construct **all** the paths*
- Not sure why it occurs
  - Mental boundaries as buffers?
  - Make testing easier?
- Usually bad for performance
  - Buffers must go in global memory
- In many apps. it can't be avoided
  - But often it can

```
void MySimulation()
{
  __global__
  unsigned uBuff[n*k],gBuff[n*k],...;

  GenUniform(n,k,uBuff);
  __syncthreads();

  UnifToGaussian(n,k,uBuff,gBuff);
  __syncthreads();

  ConstructPath(n,k,gBuff,pBuff);
  __syncthreads();

  CalcPayoff(n,k,pBuff);
  __syncthreads();
}
```

Memory in MC: reduce and re-use

```
void MySimulation()
{
  __global__
  unsigned uBuff[n*k],gBuff[n*k],...;

  GenUniform(n,k,uBuff);
  __syncthreads();

  UnifToGaussian(n,k,uBuff,gBuff);
  __syncthreads();

  ConstructPath(n,k,gBuff,pBuff);
  __syncthreads();

  CalcPayoff(n,k,pBuff);
  __syncthreads();
}
```

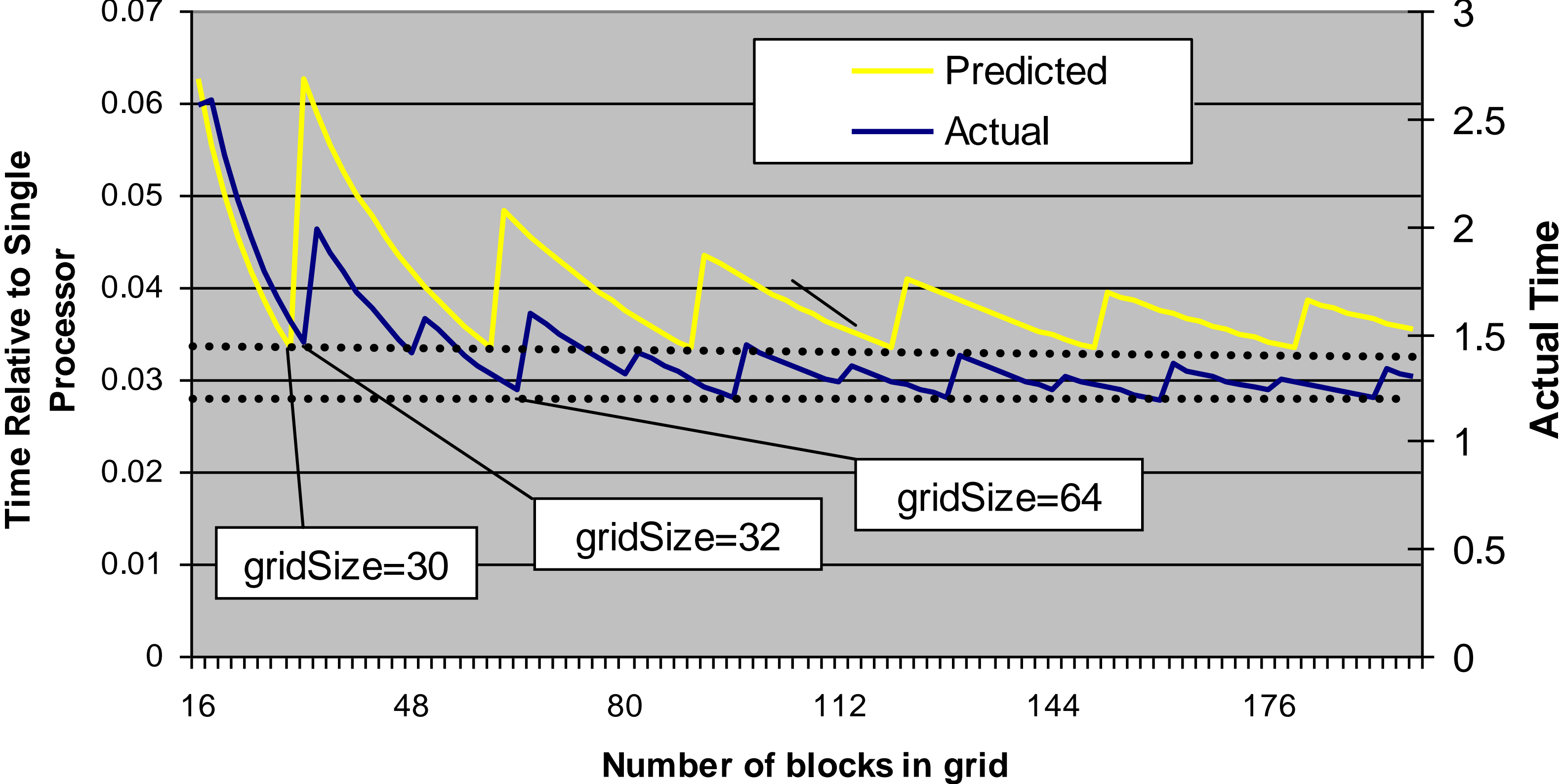
```
void MySimulation()
{
  __shared__ int buff[k];

  for(int i=0;i<n;i++){
    GenUniform(k, buff);
    __syncthreads();

    UnifToGaussian(k,buff);
    __syncthreads();

    ConstructPath(k,buff);
    __syncthreads();

    CalcPayoff(k,buff);
    __syncthreads();
  }
}
```

If possible: make a buffer big enough for just one task and operate in-place

Optimisation is highly non-linear

- Small changes produce huge performance swings...
  - ➢ Changing the number of threads per block
  - ➢ Altering the order of independent statements
  - ➢ Supposedly redundant __syncthread() calls
- General practises apply for Monte Carlo
  - Use large grid sizes: larger than you might expect
  - Allocate arrays to physical memory very carefully
  - Keep out of global memory in inner loops (and outer loops)
    - Prefer computation to global memory
  - Keep threads in a branch together
    - Prefer more complex algorithms with no branches
    - Watch out for statistical branching

The compiler+GPU is a black box



Uniform Random Number Generation

- Goal: generate stream of numbers that "looks random"
- Generated by deterministic mechanism (Pseudo-Random)
  - Must use only standard CPU instructions (unlike True-RNG)
  - Can start two RNGs from same seed and get same stream
- Long period: deterministic generators must repeat
  - Rule of thumb: if we use n samples, must have period >> n^2
  - In practise: would prefer period of at least 2^{128}
- Statistically random: high entropy, "random looking"
  - Check using test batteries: look for local correlations and biases
  - Theoretical tests: prove properties of entire sequence

Basics of RNGs

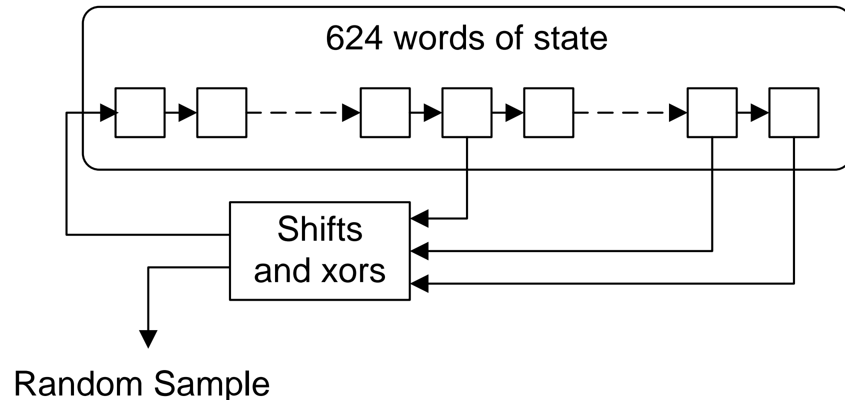
- State-space: each RNG has a finite set of states **s**
  - Given *n* bits in the state, maximum period is 2^n
  - Period of 2^{128} -> must have at least 4 words in state
- Transition function: moves generator from state to state
  - *f* : **s** -> **s**
- Output function: convert current state into output sample
  - *g* : **s** -> [0..2^{32}) or g : **s** -> [0,1)
- Choose an initial seed s_0 \in s
  - $s_{i+1}=f(s_i)$
  - $x_i = g(s_i)$
- Period: smallest p such that for all i : $x_{i+p}=x_i$

Existing RNGS

- Lots of existing software generators
  - Linear Congruential
  - Multiply Recursive
  - XorShift
  - Lagged Fibonacci
  - Mersenne Twister
- We can still use these existing generators in a GPU
  - Useful for checking results against CPU
- But! Why not derive new generators for GPU
  - GPU has interesting features: lets use them
  - CPU and GPU costs are different: old algorithms difficult to use

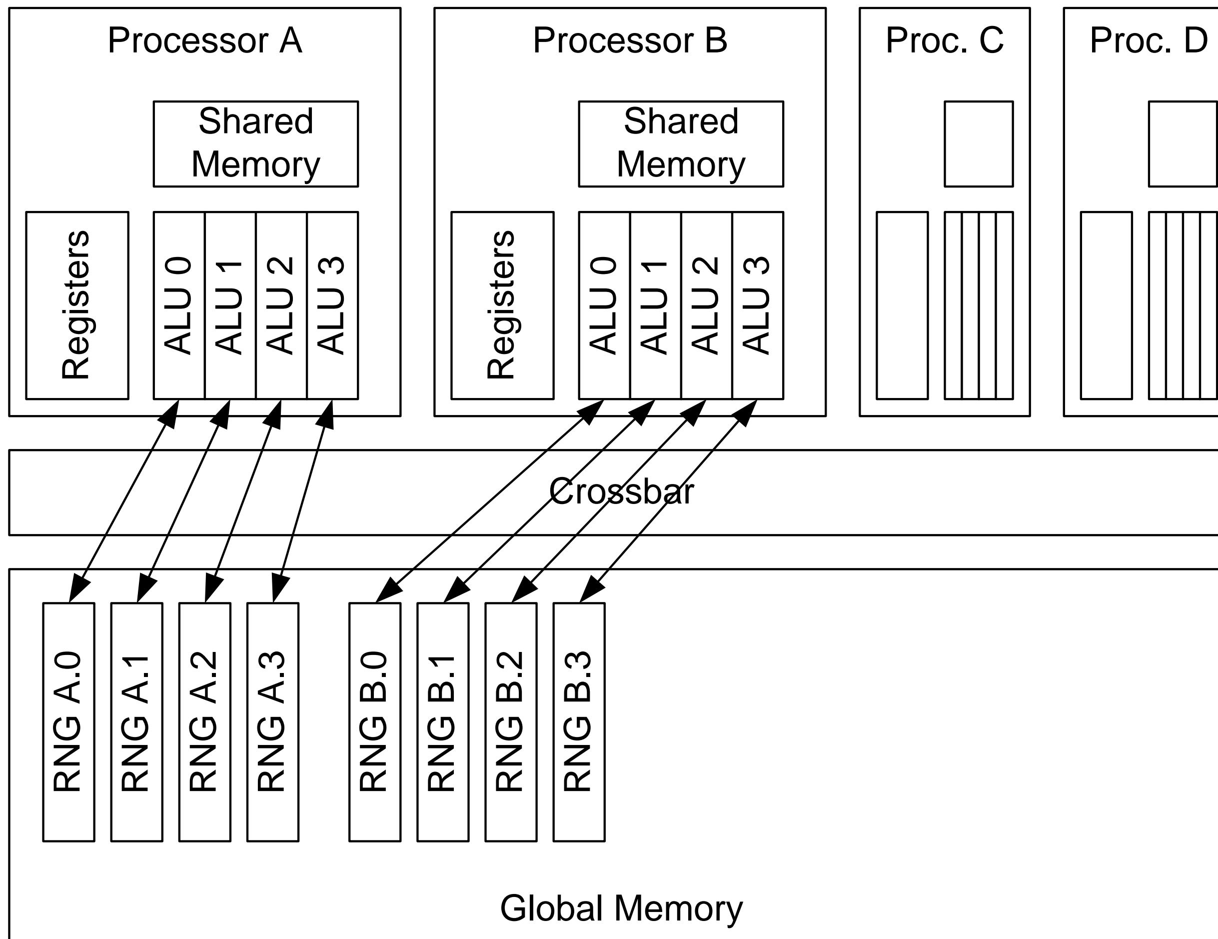
Example: Mersenne Twister

```
unsigned MT19937(unsigned &i, unsigned *s)
{
  t0 = s[i%N];     // can be cached in register
  t1 = s[(i+1)%N];
  t2 = s[(i+M)%N];
  tmp =someShiftsAndXors(t0,t1,t2);
  s[i%n] = tmp;
  i++;
  return moreShiftsAndXors(tmp);
}
```



- Well respected generator, widely used
  - Excellent quality: good theoretical and empirical quality
  - Very long period: 2^{19937}
  - Efficient in software
- Requires a state of 624 words organised as circular buffer
  - Two reads and one write per cycle

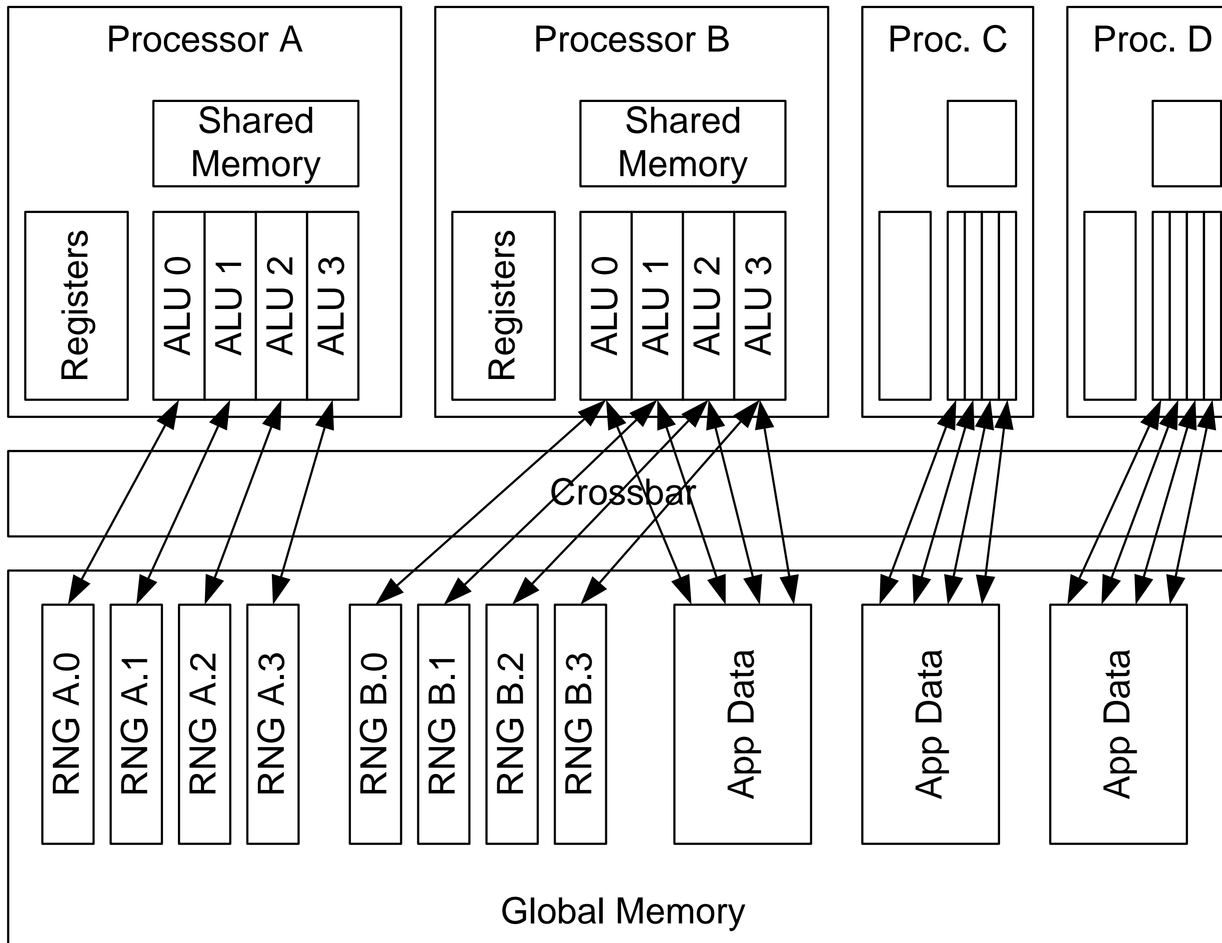
Basic approach: one RNG per thread



The memory bottleneck

- Each thread does two reads and one write per sample
  - 12 bytes of traffic to global memory per sample
  - Total bandwidth is about 18GB/s on C1060
  - Maximum generation rate: ~1.5 GSamples/s
- Might seem like a an acceptable rate
  - RNG is driving simulation: can use up memory latency cycles
  - What if simulation needs to use global memory as well?
- More sophisticated approaches are possible
  - Place RNG states in shared memory in clever ways
  - Code gets very complicated, and RNG API more complex
    - We want a function that looks like rand()
- But... why not try something new?

The memory bottleneck



Designing from scratch for a GPU

- Where can we store RNG state on a GPU
  - Global memory: large, very slow
  - Shared memory: small, fast
  - Registers: small, fast, can't be indexed
- Could store state in shared memory?
  - But would need four or more words per thread... too expensive
- Could store state in registers?
  - Around four registers per thread is ok, but only allows period 2^{128}
  - RNG generator function must be complex (and slow) for quality
- One solution: period 2^{128} generator using registers
  - e.g. Marsaglia's KISS generator: excellent quality, but ***slow***

Designing from scratch for a GPU

- Ok, what else does the GPU have that we can use?
  - Automatically synchronised fine-grain warp-level parallelism
  - Automatically synchronised warp-level access to shared memory

```
void rotateBlock(float *mem)
  float tmp=s[(tId+1)%bDim];
  __syncthreads();
  s[tId]=tmp;
  __syncthreads();
}
```

```
void rotateWarp(float *mem)
  tmp=s[32*wIdx+((wOff+1)%32)];
  s[tIdx]=tmp;
}
```

```
tId=threadIdx.x, bDim=blockDim.x
```

```
wIdx=tId/32, wOff=tId%32
```

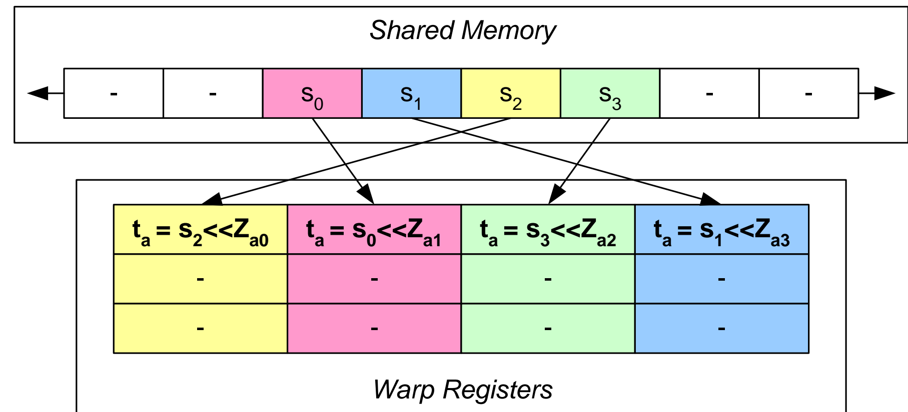
Warp Generators

- Each warp works on a shared RNG
  - All threads execute transition step in parallel
  - Each thread receives a new random number
- RNG state storage is spread across multiple threads
  - Overhead per thread is low, but can still get long periods
- Communicate via shared memory
  - Threads within warp can operate without synchronisation
  - Accesses are fast as long as we observe the rules
- Fine-grain parallelism increases quality
  - Relatively simple per-thread operation
  - Complex transformation to overall state

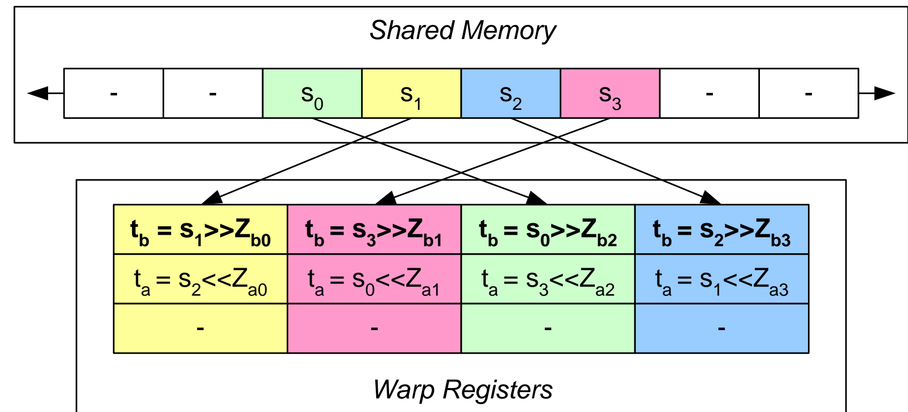
```
const unsigned K=4;   // Warp size
#define (wId threadIdx.x / K)
#define (wOff threadIdx.x % K)

const unsigned Qa[K] = {2, 0, 3, 1};
const unsigned Qb[K] = {1, 3, 0, 2};
const unsigned Za = 3;
const unsigned Zb[K] = {1, 2, 1, 3};

// RNG state, one word per thread
__shared__ unsigned s[];

// Generate new number per thread
__device__ unsigned Generate(unsigned *s)
{
  ta = s[ wId*K+Qa[wOff] ] << Za;
  tb = s[ wId*K+Qb[wOff] ] >> Zb[wOff];
  x = ta ^ tb;
  s[threadIdx.x] = x;
  return x;
}
```

- Hold state in shared memory
  - One word per thread
- Define a set of per-warp constants
  - Permutations of warp indices
  - One shared shift
  - One per-thread shift
  - *These must be chosen carefully*!
    - The ones in the code are not valid
- Four basic steps
  - Read and shift word from state
  - Read and shift different word
  - Exclusive-or them together
  - Write back new state

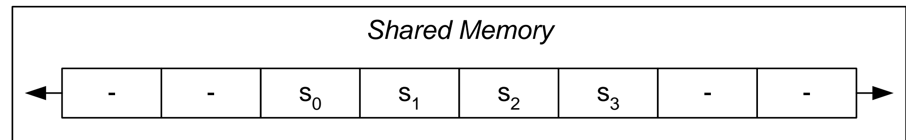
```
const unsigned K=4;    // Warp size
#define (wId threadIdx.x / K)
#define (wOff threadIdx.x % K)

const unsigned Qa[K] = {2, 0, 3, 1};
const unsigned Qb[K] = {1, 3, 0, 2};
const unsigned Za = 3;
const unsigned Zb[K] = {1, 2, 1, 3};

// RNG state, one word per thread
__shared__ unsigned s[];

// Generate new number per thread
__device__ unsigned Generate(unsigned *s)
{
  ta = s[ wId*K+Qa[wOff] ] << Za;
  tb = s[ wId*K+Qb[wOff] ] >> Zb[wOff];
  x = ta ^ tb;
  s[threadIdx.x] = x;
  return x;
}
```

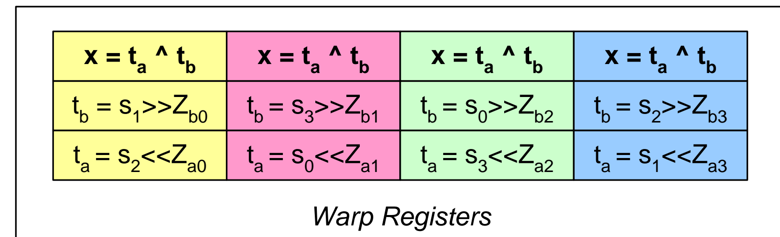


```
const unsigned K=4;    // Warp size
#define (wId threadIdx.x / K)
#define (wOff threadIdx.x % K)

const unsigned Qa[K] = {2, 0, 3, 1};
const unsigned Qb[K] = {1, 3, 0, 2};
const unsigned Za = 3;
const unsigned Zb[K] = {1, 2, 1, 3};

// RNG state, one word per thread
__shared__ unsigned s[];

// Generate new number per thread
__device__ unsigned Generate(unsigned *s)
{
  ta = s[ wId*K+Qa[wOff] ] << Za;
  tb = s[ wId*K+Qb[wOff] ] >> Zb[wOff];
  x = ta ^ tb;
  s[threadIdx.x] = x;
  return x;
}
```

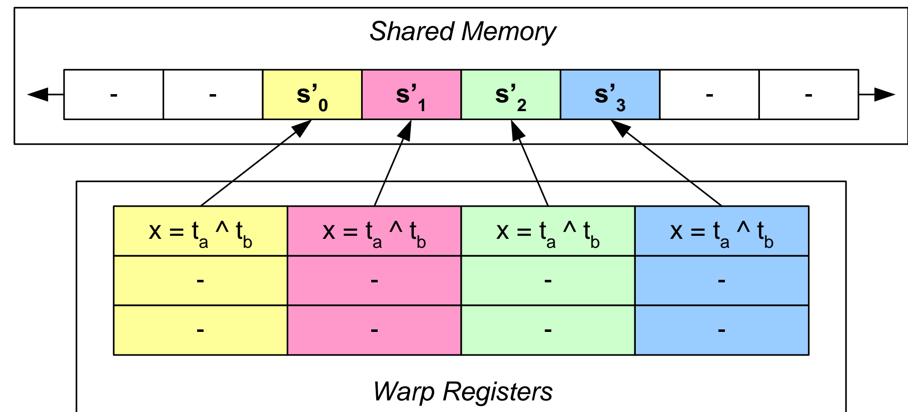


```
const unsigned K=4;    // Warp size
#define (wId threadIdx.x / K)
#define (wOff threadIdx.x % K)

const unsigned Qa[K] = {2, 0, 3, 1};
const unsigned Qb[K] = {1, 3, 0, 2};
const unsigned Za = 3;
const unsigned Zb[K] = {1, 2, 1, 3};

// RNG state, one word per thread
__shared__ unsigned s[];

// Generate new number per thread
__device__ unsigned Generate(unsigned *s)
{
  ta = s[ wId*K+Qa[wOff] ] << Za;
  tb = s[ wId*K+Qb[wOff] ] >> Zb[wOff];
  x = ta ^ tb;
  s[threadIdx.x] = x;
  return x;
}
```





```
const unsigned K=4;    // Warp size
#define (wId threadIdx.x / K)
#define (wOff threadIdx.x % K)

const unsigned Qa[K] = {2, 0, 3, 1};
const unsigned Qb[K] = {1, 3, 0, 2};
const unsigned Za = 3;
const unsigned Zb[K] = {1, 2, 1, 3};

// RNG state, one word per thread
__shared__ unsigned s[];

// Generate new number per thread
__device__ unsigned Generate(unsigned *s)
{
  ta = s[ wId*K+Qa[wOff] ] << Za;
  tb = s[ wId*K+Qb[wOff] ] >> Zb[wOff];
  x = ta ^ tb;
  s[threadIdx.x] = x;
  return x;
}
```



Features of warp RNGs

- Very efficient: ~ four instructions per random number
- Long period: warp size of 32 -> period of 2^{1024}
- Managing and seeding parallel RNGs is fast and safe
  - Random initialisation is safe as period is so large
  - Skip within stream is quite cheap: ~3000 instructions per skip
  - Use different constants for each warp: different RNG per warp
    - Can find thousands of valid RNGs easily via binary linear algebra
    - *WARNING: you cannot use arbitrary constants: it won't work*
- Statistical quality is excellent
  - Four instruction version has correlation problems
  - Very cheap (two instructions) tempering fixes them
  - Higher empirical quality than the Mersenne Twister

Comparison with other RNGs

RNG	Period	Empirical Quality - TestU01[1]			GWord/ second
		Small	Medium	Big	
Adder[2]	2^{32}	Fail	Fail	Fail	141.28
QuickAndDirty[3]	2^{32}	Fail	Fail	Fail	43.84
Warp RNG	$\mathbf{2^{1024}}$	**Pass**	**Pass**	**Pass**	**37.58**
Park-Miller[3]	2^{32}	Fail	Fail	Fail	10.67
MersenneTwister	2^{19937}	Pass	Pass[4]	Pass[4]	5.85
KISS	2^{123}	Pass	Pass	Pass	0.99

1 : TestU01 offers three levels of “crush” tests: small is quite weak, big is very stringent
2 : Adder is not really a random number generator, just a control for performance
3 : QuickAndDirty and Park-Miller are LCGs with modulus 2^{32} and (2^^{32}-1) respectively
4 : Mersenne Twister fails tests for linear complexity, but that is not a problem in most apps

http://www.doc.ic.ac.uk/~dt10/research/rngs-gpu-uniform.html

Conclusion

- GPUs are rather good for Monte-Carlo simulations
  - Random number generation (PRNG and/or QRNG) is fast
  - Embarrassingly parallel nature works well with GPU
  - Single-precision is usually good enough
- Need to pay some attention to the details
  - Watch out for scalar algorithms: warp divergence hurts
  - Inversion is trickier than it seems
  - Statistical accumulators should use double-precision
  - Keep things out of global memory (but: true of any application)
- If you have the time, think of new algorithms
  - Advantage of CUDA is ability to use existing algorithms/code
  - Potential advantage of GPUs is from new algorithms