CS 6068 Nov 20, 2012
Lecture 11 - Parallel Monte Carlo

- History of Monte Carlo methods
- Random Numbers in CUDA
- Normal numbers: Is pi random?
- Sequential pseudo- and quasi- random number generators
- Parallel random number generators
- Generating non-uniform random numbers
- Monte Carlo case studies

Monte Carlo Methods

- Monte Carlo is another name for statistical sampling methods that are of great importance to physics and computer science
- Applications of Monte Carlo Method
  - Evaluating integrals of arbitrary functions of 6+ dimensions
  - Financial Simulations and Pricing Models, Predicting future values of stocks
  - Solving partial differential equations
  - Image processing , e.g., sharpening images
  - Modeling complex cell populations
  - Finding approximate solutions to NP-hard problems

History of Monte Carlo Method

- Credit for inventing the Monte Carlo method is shared by **Stanislaw Ulam, John von Neumann and Nicholas Metropolis.**

- Ulam, a Polish born mathematician, worked for John von Neumann on the Manhattan Project. Ulam designed the hydrogen bomb with Edward Teller in 1951. In a thought experiment he conceived of the MC method in 1946 while pondering the probabilities of winning a card game of solitaire.

- Ulam, von Neuman, and Metropolis developed algorithms that transformed statistical sampling from a mathematical curiosity to a formal methodology applicable to a wide variety of problems. Metropolis named the new methodology after the casinos of Monte Carlo. Ulam and Metropolis published a paper called "The Monte Carlo Method" in *Journal of the American Statistical Association* in 1949.

Errors in Estimation and Two Important Questions for Monte Carlo

- Errors arise from two sources
  - Statistical: using finite number of samples to estimate an infinte sum.
  - Computational: using finite state machines to produce statistically independent random numbers.
- Questions
  - how many samples are needed to ensure a level of statistical accuracy:

    m significant digits with high probability 1-1/n

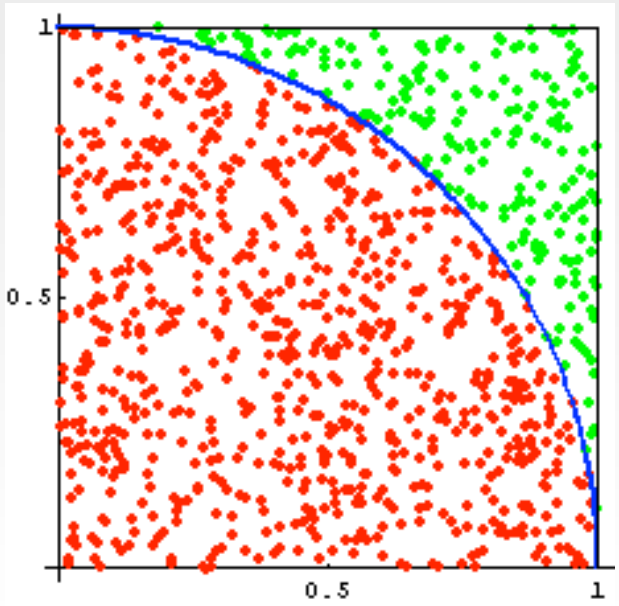
- "even the best methods rarely offers more than 2-3 digit accuracy"-- G. Fishman, *Monte Carlo Methods*, Springer

Controlling Error

- Two tenets from statistics:
  - Weak Law of Large Numbers
    - iid is shorthand for independent and identically distributed (e.g., dice rolls)
    - The value of the average of iid random variables converges to mean, so as sample size n grows the probability of large error vanishes
  - Central Limit Theorem
    - Sum of iid variables X converges to a normal distribution with mean *n*mean(X)* and variance *var(X)/n*
    - So as *n* grows error can be estimated using normal tables and theorems about tails of distribution.
- Caveats in practice
  - Convergence rates differ and can be complex
  - Results in additional error estimates or need larger sample size than normal distribution would indicate

A simple Monte Carlo simulation to approximate the value of pi could involve randomly selecting points in the unit square and determining the hit ratio, e.g. 1000 points - 787 hits

MC Estimate 0.787 * 4 = 3.148



- A more realistic example of Monte Carlo methods is in finance—for example, the price S_0 of an equity at time 0—then choose an stochastic model that appears to model previous equity paths reasonably well.
- A commonly used model is geometric Brownian motion, where the final price of the stock at time t is modeled as
- $S_t = S_0 \, exp(\mu + Ns)$, where N is a random sample from the Gaussian distribution.

The Monte Carlo approach is easy to parallelize.
There are five major steps:

1.Assign each processing element a random sequence.
Each processing element must use a different random number sequence, which should be uncorrelated with the sequences used by all other processors.

2.Propagate the simulation parameters (for example, S_0) to all processing elements, and tell them how many simulation runs to execute.

3.Generate random number streams for use by each processing element.

4.Execute the simulation kernel on the processing elements in parallel.

5.Gather the simulation outputs from each processing element and combine them to produce the approximate results.

Pseudo-random RNGs

Main general requirements that we wish PRNGs to satisfy:

- **A long period.** Every deterministic generator must eventually loop, but the goal is to make the loop period as long as possible. There is a strong argument that if n random samples are used across all nodes in a simulation, then the period of the generator should be at least n^2.

- **Good statistical quality.** The output from the generator should be practically indistinguishable from a TRNG of the required distribution, and it should not exhibit any correlations or patterns. Poor generator quality can ruin the results of Monte Carlo applications, and it is critical that generators are able to pass the set of theoretical and empirical tests for quality that are available. Numerous statistical tests are available to verify this requirement (Knuth 1969, Marsaglia 1995, L'Ecuyer 2006).

Aside: Are the digits of pi random?

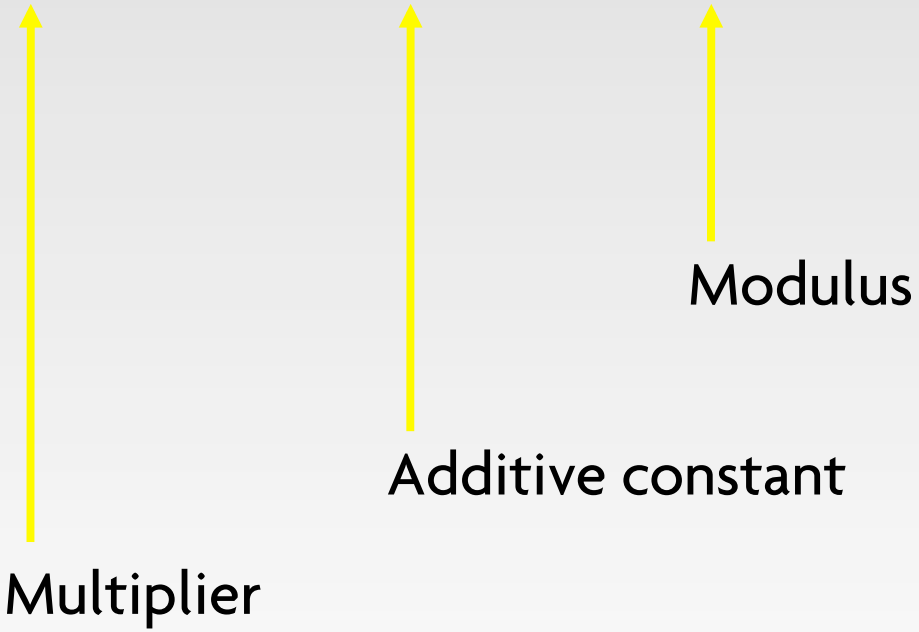
Definition: A number is *normal* (base-b) if its base-b expansion has each digit appearing with average frequency tending to 1/b.

Open Problem: Are fundamental mathematical constants such as pi, ln2, sqrt(2), and e normal?

Extensive testing show all these numbers have very strong statistically random properties.

Linear Congruential RNGs

$$X_i = (a \times X_{i-1} + c) \bmod M$$



Sequence depends on choice of seed, X_0

Period of Linear Congruential RNG

- Maximum period is M
- For 32-bit integers maximum period is 2^{32}, or about 4 billion
- This is too small for many modern apps
- Also there are known statistical flaws

Lagged Fibonacci RNGs

$$X_i = X_{i-p} * X_{i-q}$$

- p and q are lags, $p > q$
- * is any binary arithmetic operation
- Addition modulo *M*
- Subtraction modulo *M*
- Multiplication modulo *M*
- Bitwise exclusive-or

Properties of Lagged Fibonacci RNGs

- ★ Require storage and access to *p* seed values
- ★ Careful selection of seed values, *p*, and *q* can result in very long periods and gd randomness
- ★ For example, suppose *M* has *b* bits, then maximum period for additive lagged Fibonacci RNG is $(2^p - 1)2^{b-1}$
- ★ but each thread will require its own large state, so this must be stored in global memory. This method may be useful in some GPU-based applications, because of the simplicity and small number of registers required.

Mersenne Twister

One of the most widely used methods for RNGs is the Mersenne twister (Matsumoto and Nishimura 1998), which has an enormous period of 2^19,937 and extremely good statistical quality.

However, it has a large state that must be updated serially. Thus each thread must have an individual state global memory - makes the generator too slow, except in cases where quality is needed.

Parallel Independent Sequences

- Run sequential PRNG on each thread process
- Start each with different seed(s) or other parameters
- Example: linear congruential RNGs with different additive constants
- Works well with lagged Fibonacci RNGs
- Supports goals of locality and scalability

CUDA Lib for Random Numbers

- Give a randState to each CUDA thread, from which it can sample from
- On the host, create a device pointer to hold the randStates
- Malloc number of states equal to number of threads
- Pass the device pointer to your function
- Init the random states
- Call random function - curand_uniform

with the state given to that thread

- Free the randomStates

- Random headers can be found in ‹curand.h› and ‹curand_kernel.h›

```
__global__ void monteCarlo(float *g_odata, int trials,
  curandState *states){

  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int k, incircle;
  float x, y, z;
  incircle = 0;
  curand_init(1234, i, 0, &states[i]);
  for(k = 0; k < trials; k++){
    x = curand_uniform(&states[i]);
    y = curand_uniform(&states[i]);
    z = sqrt(x*x + y*y);
    if (z <= 1) incircle++; else{} }
 __syncthreads();
g_odata[i] = incircle; }
```

```
int main() {
 float* solution = (float*)calloc(100, sizeof(float));
float *sumDev, sumHost[NUM_BLOCK*NUM_THREAD];
int trials, total; curandState *devStates;
trials = 100; total = trials*NUM_THREAD*NUM_BLOCK;
dim3 dimGrid(NUM_BLOCK,1,1); // Grid dimensions
dim3 dimBlock(NUM_THREAD,1,1); // Block dimensions
size_t size = NUM_BLOCK*NUM_THREAD*sizeof(float); //Array memory size
cudaMalloc((void **) &sumDev, size);
    // Allocate array on device
cudaMalloc((void **) &devStates, size*sizeof(curandState));
    // Do calculation on device by calling CUDA
kernel monteCarlo <<<dimGrid, dimBlock, size>>> (sumDev, trials, devStates);
    // call reduction function to sum
reduce0 <<<dimGrid, dimBlock, size>>> (sumDev); // Retrieve result from device and store
    it in host array
cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
*solution = 4*(sumHost[0]/total); printf("%.*f\n", 1000, *solution); free (solution); //
    *solution = NULL; return 0; }
```

- The total state space of the PRNG before

you start to see repeats is about 2^190

CUDA's RNG is designed so that when the same seed is used with each thread, the generated random numbers spaced 2^67 numbers away in the PRNG's sequence

- When calling curand_init with a seed, it scrambles that seed and then skips ahead 2^67 numbers
- This even spacing between threads guarantees that you can analyze the randomness of the PRNG and those results will hold no matter what seed you use

- What if you're running millions of threads and each thread needs RNs?
  - Not completely uncommon
  - You could run out of state space per thread and start seeing repeats... ((2^190) / (10^6)) / (2^67) = 1.0633824 × 10^31
- Can seed each thread with a different seed (ex. theadIdx.x), and then set the state to zero (i.e. don't advance each thread by 2^67)
  - This may introduce some bias / correlation, but not many other options
  - Don't have the same assurance of statistical properties remaining the same as seed changes
  - It's also faster (by a factor of 10x or so)

Distributions other than Uniform Distribution

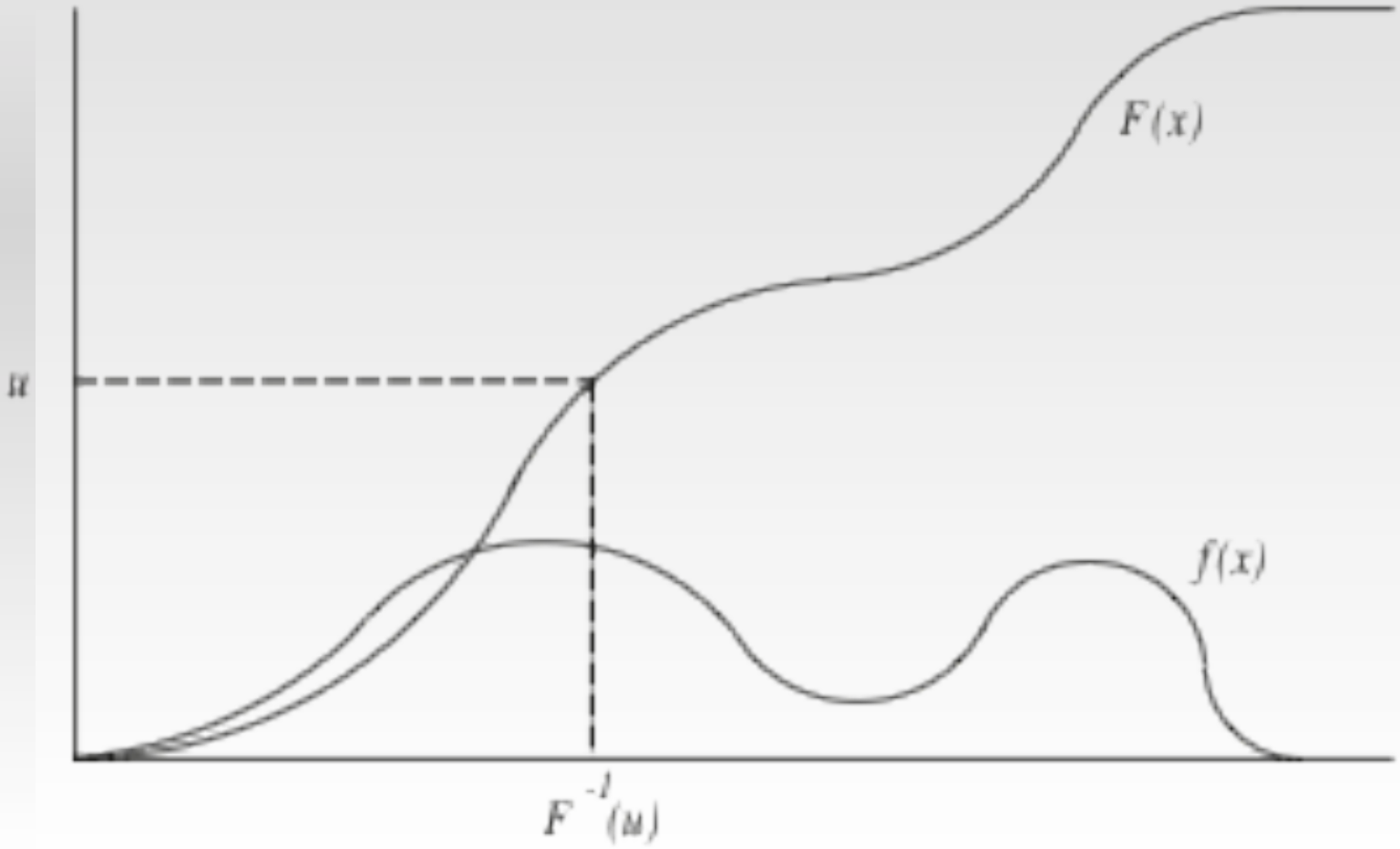
- Analytical transformations - Exponential
- Box-Muller Transformation - Normal
- Rejection method - General

Analytical Transformation

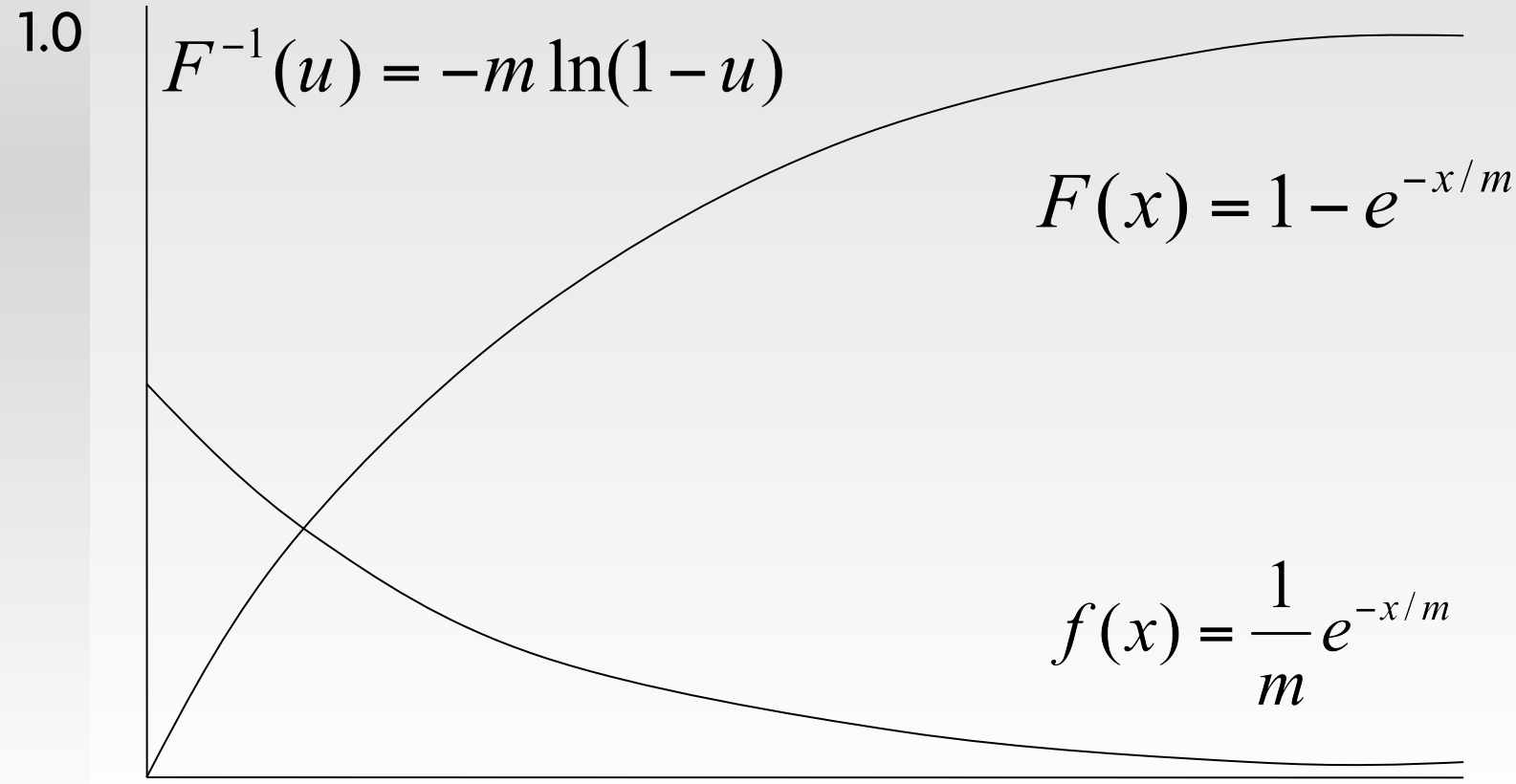
-probability density function f(x)

-cumulative distribution F(x)

In theory of probability, a quartile function of a distribution is the inverse of its cumulative distribution function.



Exponential Distribution: An exponential distribution arises naturally when modeling the time between independent events that happen at a constant average rate and are memoryless. One of the few cases where the quartile function is known analytically.



Samples of Exponential

- Produce four samples from an exponential distribution with mean 3
- Uniform sample: 0.540, 0.619, 0.452, 0.095
- Take natural log of each value and multiply by -3
- Exponential sample: 1.850, 1.440, 2.317, 7.072

Sample Example 2:

- Want to run simulation that advances in time steps of 1 second
- Probability of an event happening is from an exponential distribution with mean 5 seconds
- What is probability that event will happen in next second?
- $F(x=1/5) = 1 - \exp(-1/5)) = 0.181269247$
- Use uniform random number to test for occurrence of event (if $u < 0.181$ then 'event' else 'no event')

Normal Distributions:

Box-Muller Transformation

- Cannot invert cumulative distribution function to produce formula yielding random numbers from normal (gaussian) distribution
- Box-Muller transformation produces a pair of standard normal deviates g_1 and g_2 from a pair of uniform deviates u_1 and u_2

Box-Muller Transformation

repeat

$v_1 \leftarrow 2u_1 - 1$

$v_2 \leftarrow 2u_2 - 1$

$r \leftarrow v_1^2 + v_2^2$

until $r > 0$ and $r < 1$

$f \leftarrow \text{sqrt}(-2 \ln r / r)$

$g_1 \leftarrow f\, v_1$

$g_2 \leftarrow f\, v_2$

This is a consequence of the fact that the chi-square distribution with two degrees of freedom is an easily-generated exponential random variable. *Ref: Wikipedia*

Normal Sample Example

- Produce four samples from a normal distribution with mean 0 and standard deviation 1

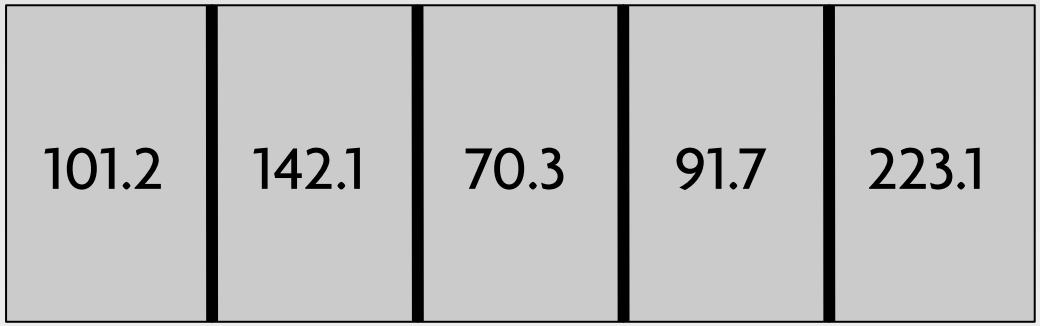
u_1	u_2	v_1	v_2	r	f	g_1	g_2
0.234	0.784	-0.532	0.568	0.605	1.290	-0.686	0.732
0.824	0.039	0.648	-0.921	1.269			
0.430	0.176	-0.140	-0.648	0.439	1.935	-0.271	-1.254

Parking Garage Simulation

- Parking garage has *S* stalls
- Car arrivals fit Poisson distribution with mean *A*: Exponentially distributed inter-arrival times
- Stay in garage fits a normal distribution with mean *M* and standard deviation *M/S*

Implementation Idea

Times Spaces Are Available



Current Time

64.2

Car Count

15

Cars Rejected

2

Monte Carlo Method in Finance

First stage: generation of a normally distributed sample sequence.

- parallel version of the Mersenne Twister
- apply Box-Müller transformation
- “MersenneTwister” sample in the CUDA SDK

Second Stage: compute an expected value and confidence width for the underlying option - evaluating payoff function for many simulation paths and computing the mean of the results.

Monte Carlo Method in Finance

Third Stage: Pricing a single option using Monte Carlo simulation is inherently a one-dimensional problem, but if we are pricing multiple options, we can think of the problem in two dimensions.

Easy to determine our grid layout: launch a grid X blocks wide by Y blocks tall, where Y is the number of options we are pricing. We also use the number of options to determine X; we want X·Y to be large enough to have plenty of thread blocks to keep the GPU busy.

If the number of options is less than 16, we use 64 blocks per option, and otherwise we use 16.