Simulating spin models on GPU
Lecture 3: Random number generators

Martin Weigel

Applied Mathematics Research Centre, Coventry University, Coventry, United Kingdom and
Institut für Physik, Johannes Gutenberg-Universität Mainz, Germany

IMPRS School 2012: GPU Computing,
Wroclaw, Poland, October 31, 2012

Emmy Noether-Programm
Deutsche Forschungsgemeinschaft DFG

Junior Research Group
"Complex Systems with Frustration"

JG|U

Coventry University

Outline

RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

Approaches:

- true randomness from, e.g., fluctuations in a resistor: too slow

RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

Approaches:

- true randomness from, e.g., fluctuations in a resistor: too slow
- pseudorandom number generator: deterministic sequence of (typically integer) numbers with the following properties

RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of “randomness”.

Approaches:

- true randomness from, e.g., fluctuations in a resistor: too slow
- pseudorandom number generator: deterministic sequence of (typically integer) numbers with the following properties
  - based on a state vector
  - with a finite period
  - reproducible if using the same seed
  - typically produce uniform distribution on $[0, NMAX]$ or $[0, 1]$
  - further distributions (such as Gaussian) generated from transformations

RNG: definition

Stochastic simulations such as Monte Carlo and molecular dynamics (with a thermostat) require a reliable stream of "randomness".

Approaches:

- true randomness from, e.g., fluctuations in a resistor: too slow
- pseudorandom number generator: deterministic sequence of (typically integer) numbers with the following properties
  - based on a state vector
  - with a finite period
  - reproducible if using the same seed
  - typically produce uniform distribution on $[0, NMAX]$ or $[0, 1]$
  - further distributions (such as Gaussian) generated from transformations
- generally two types of pseudo RNGs considered
  - for general purposes, including simulations
  - or for cryptographic purposes, requiring sufficient randomness to prevent efficient stochastic inference

The story of R250

John von Neumann

"*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*" (1951)

The story of R250

John von Neumann

"*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*" (1951)

For any pseudo RNG (or RNG, for short) there must exist an algorithm/test that distinguishes the generated sequence from a truly random sequence.

The story of R250

John von Neumann

"*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*" (1951)

For any pseudo RNG (or RNG, for short) there must exist an algorithm/test that distinguishes the generated sequence from a truly random sequence. (If nothing else, this can be the algorithm generating the sequence itself!)

The story of R250

John von Neumann

"*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*" (1951)

For any pseudo RNG (or RNG, for short) there must exist an algorithm/test that distinguishes the generated sequence from a truly random sequence. (If nothing else, this can be the algorithm generating the sequence itself!)

VOLUME 69, NUMBER 23 PHYSICAL REVIEW LETTERS 7 DECEMBER 1992

Monte Carlo Simulations: Hidden Errors from "Good" Random Number Generators

Alan M. Ferrenberg and D. P. Landau
Center for Simulational Physics, The University of Georgia, Athens, Georgia 30602

Y. Joanna Wong
IBM Corporation, Supercomputing Systems, Kingston, New York 12401
(Received 29 July 1992)

The Wolff algorithm is now accepted as the best cluster-flipping Monte Carlo algorithm for beating "critical slowing down." We show how this method can yield *incorrect* answers due to subtle correlations in "high quality" random number generators.

PACS numbers: 75.40.Mg, 05.70.Jk, 64.60.Fr

The explosive growth in the use of Monte Carlo simulations in diverse areas of physics has prompted extensive investigation of new methods and of the reliability of both old and new techniques. Monte Carlo simulations are ing model, to study the time correlations, but so far there has been no careful study of the accuracy of the thermodynamic properties which are extracted from the configurations generated by this process.

The story of R250

The New York Times

Technology

Coin-Tossing Computers Found to Show Subtle Bias

By MALCOLM W. BROWNE
Published: January 12, 1993

WHEN scientists use computers to try to predict complex trends and events, they often apply a type of calculation that requires long series of random numbers. But instructing a computer to produce acceptably random strings of digits is proving maddeningly difficult.

In deciding which team kicks off a football game, the toss of a real coin is random enough to satisfy all concerned. But the cost of even a slightly nonrandom string of electronic coin tosses can be devastating to both practical problem-solving and pure theory, and a new investigation has revealed that nonrandom computer tosses are much more common than many scientists had assumed.

Mathematical "models" designed to predict stock prices, atmospheric warming, airplane skin friction, chemical reactions, epidemics, population growth, the outcome of battles, the locations of oil deposits and hundreds of other complex matters increasingly depend on a statistical technique called Monte Carlo Simulation, which in turn depends on reliable and inexhaustible sources of random numbers.

Monte Carlo Simulation, named for Monaco's famous gambling casino, can help to represent very complex interactions in physics, chemistry, engineering, economics and environmental dynamics mathematically. Mathematicians call such a representation a "model," and if a model is accurate enough, it produces the same responses to manipulations that the real thing would do. But Monte Carlo modeling contains a dangerous flaw: if the supposedly random numbers that must be pumped into a simulation actually form some subtle, nonrandom pattern, the entire simulation (and its predictions) may be wrong.

Random number testing

A sequence u_i of pseudo-random numbers is perfect iff *all* sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary t.

Random number testing

A sequence u_i of pseudo-random numbers is perfect iff *all* sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0,1]^t$ for arbitrary t. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:

Random number testing

A sequence u_i of pseudo-random numbers is perfect iff *all* sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0,1]^t$ for arbitrary t. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)

Random number testing

A sequence u_i of pseudo-random numbers is perfect iff *all* sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0,1]^t$ for arbitrary t. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)

On the other hand, there are cryptographic tests based on the lack of predictability.

Random number testing

A sequence u_i of pseudo-random numbers is perfect iff *all* sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0,1]^t$ for arbitrary t. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)

On the other hand, there are cryptographic tests based on the lack of predictability.

No RNG can pass every conceivable test, so a *bad* RNG is one that fails simple tests, and a *good* RNG is one that only fails only very complicated tests.

Random number testing

A sequence u_i of pseudo-random numbers is perfect iff *all* sequences $(u_0, \ldots, u_{t-1})$ are uniformly distributed over $[0, 1]^t$ for arbitrary t. Clearly, this cannot be the case, already because of the finite period.

- Derived statistical tests:
  - test for uniformity
  - correlation tests
  - comparison to combinatorial identities
  - comparison to other known statistical results
  - application tests (e.g., Ising model)

On the other hand, there are cryptographic tests based on the lack of predictability.

No RNG can pass every conceivable test, so a *bad* RNG is one that fails simple tests, and a *good* RNG is one that only fails only very complicated tests.

Test batteries:

- DieHard (1995) by G. Marsaglia, now outdated
- TestU01 (2002/2009) by P. L'Ecuyer and co-workers, quasi standard

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel.

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single "RNG process"producing and handing out the numbers would be a severe bottleneck, impeding scaling.

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single "RNG process"producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single "RNG process"producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single "RNG process"producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single "RNG process"producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together
- This could be reached by
  - (a) division of the stream of a long-period generator into non-overlapping sub-streams to be produced and consumed by the different threads of the application, or

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single "RNG process"producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together
- This could be reached by
  - (a) division of the stream of a long-period generator into non-overlapping sub-streams to be produced and consumed by the different threads of the application, or
  - (b) use of very large period generators such that overlaps between the sequences of the different instances are improbable, if each instance is seeded differently, or

Requirements for parallel computing

In applications such as Monte Carlo of lattice systems, we want to update many spins in parallel. A single "RNG process"producing and handing out the numbers would be a severe bottleneck, impeding scaling.

- hence, each thread needs its own RNG (potentially millions of them)
- to minimize the pressure on the bus, on registers and shared memory, the RNG state needs to be as small as possible
- the streams of all RNG instances must be sufficiently uncorrelated to yield reliable results together
- This could be reached by
  - (a) division of the stream of a long-period generator into non-overlapping sub-streams to be produced and consumed by the different threads of the application, or
  - (b) use of very large period generators such that overlaps between the sequences of the different instances are improbable, if each instance is seeded differently, or
  - (c) setup of independent generators of the same class of RNGs using different lags, multipliers, shifts etc.

Outline

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = a x_n + c \pmod{m}.$$

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = ax_n + c \pmod{m}.$$

- for $m = 2^{32}$ or $2^{32} - 1$, the maximal period is of the order $p \approx m \approx 10^9$, much too short for large-scale simulations

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = ax_n + c \pmod{m}.$$

- for $m = 2^{32}$ or $2^{32} - 1$, the maximal period is of the order $p \approx m \approx 10^9$, much too short for large-scale simulations
- one should actually use at most $\sqrt{p}$ numbers of the sequence

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = ax_n + c \pmod m.$$

- for $m = 2^{32}$ or $2^{32} - 1$, the maximal period is of the order $p \approx m \approx 10^9$, much too short for large-scale simulations
- one should actually use at most $\sqrt{p}$ numbers of the sequence
- for $m = 2^{32}$, modulo can be implemented as overflow, but then period of lower rank bits is only 2^k

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

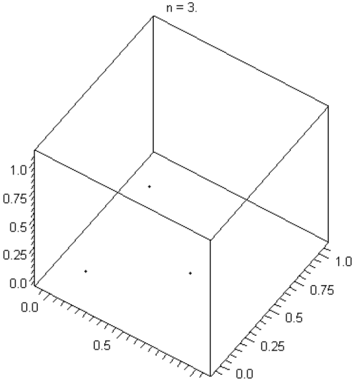
$$x_{n+1} = ax_n + c \pmod{m}.$$

- for $m = 2^{32}$ or $2^{32} - 1$, the maximal period is of the order $p \approx m \approx 10^9$, much too short for large-scale simulations
- one should actually use at most $\sqrt{p}$ numbers of the sequence
- for $m = 2^{32}$, modulo can be implemented as overflow, but then period of lower rank bits is only 2^k
- has poor statistical properties, e.g., k-tuples of (normalized) numbers lie on hyper-planes

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = a x_n + c \pmod{m}.$$



(Source: Wikipedia)

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = a x_n + c \pmod{m}.$$

- for $m = 2^{32}$ or $2^{32} - 1$, the maximal period is of the order $p \approx m \approx 10^9$, much too short for large-scale simulations
- one should actually use at most $\sqrt{p}$ numbers of the sequence
- for $m = 2^{32}$, modulo can be implemented as overflow, but then period of lower rank bits is only 2^k
- has poor statistical properties, e.g., k-tuples of (normalized) numbers lie on hyper-planes
- state is just 4 bytes per thread

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = ax_n + c \pmod{m}.$$

- for $m = 2^{32}$ or $2^{32} - 1$, the maximal period is of the order $p \approx m \approx 10^9$, much too short for large-scale simulations
- one should actually use at most $\sqrt{p}$ numbers of the sequence
- for $m = 2^{32}$, modulo can be implemented as overflow, but then period of lower rank bits is only 2^k
- has poor statistical properties, e.g., k-tuples of (normalized) numbers lie on hyper-planes
- state is just 4 bytes per thread
- can easily skip ahead via $x_{n+t} = a_t x_n + c_t$ with

$$a_t = a^t \pmod{m}, \quad c_t = \sum_{i=1}^{t} a^i c \pmod{m}.$$

Linear congruential generators

Simplest choice satisfying these requirements is linear congruential generator (LCG):

$$x_{n+1} = ax_n + c \pmod{m}.$$

- for $m = 2^{32}$ or $2^{32} - 1$, the maximal period is of the order $p \approx m \approx 10^9$, much too short for large-scale simulations
- one should actually use at most $\sqrt{p}$ numbers of the sequence
- for $m = 2^{32}$, modulo can be implemented as overflow, but then period of lower rank bits is only 2^k
- has poor statistical properties, e.g., k-tuples of (normalized) numbers lie on hyper-planes
- state is just 4 bytes per thread
- can easily skip ahead via $x_{n+t} = a_t x_n + c_t$ with

$$a_t = a^t \pmod{m}, \quad c_t = \sum_{i=1}^{t} a^i c \pmod{m}.$$

- can be improved by choosing $m = 2^{64}$ and truncation to 32 most significant bits, period $p = m \approx 10^{18}$ and 8 bytes per thread

LCGs: implementation

The implementation is indeed very simple and can be performed in-line:

LCGs: implementation

The implementation is indeed very simple and can be performed in-line:

LCG implementation

```
#define A32 1664525
#define C32 1013904223

unsigned int ran;
CONVERT(ran = A32*ran+C32);
```

LCGs: implementation

The implementation is indeed very simple and can be performed in-line:

LCG implementation

```
#define A32 1664525
#define C32 1013904223

unsigned int ran;
CONVERT(ran = A32*ran+C32);
```

The output function for converting from $[0, \mathrm{INTMAX}]$ to $[0, 1]$ could be implemented in different ways:

LCGs: implementation

The implementation is indeed very simple and can be performed in-line:

LCG implementation

```
#define A32 1664525
#define C32 1013904223

unsigned int ran;
CONVERT(ran = A32*ran+C32);
```

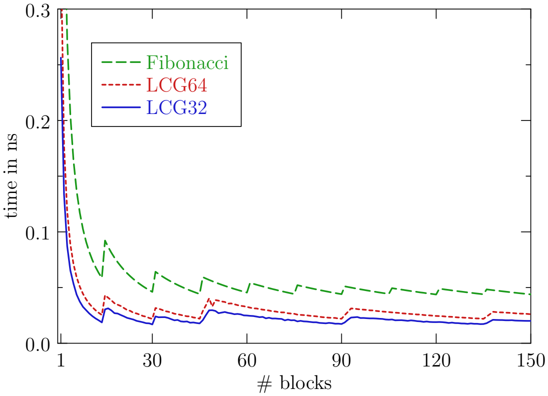
The output function for converting from $[0, \mathrm{INTMAX}]$ to $[0, 1]$ could be implemented in different ways:

LCG implementation

```
#define MULT32 2.328306437080797e-10f

#define CONVERT(x) (MULT32*((unsigned int)(x)))
//#define CONVERT(x) _curand_uniform(x)
//#define CONVERT(x) __fdividef(__uint2float_rz(x),(float)0x100000000);
```

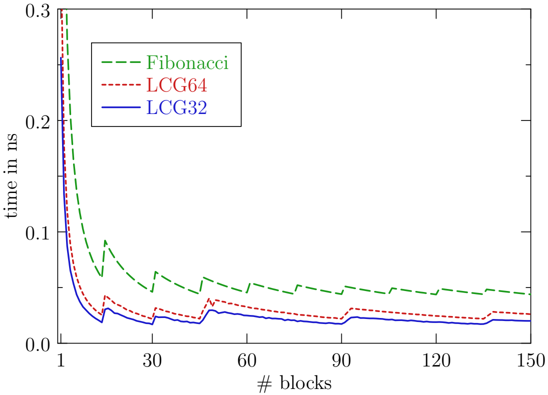
LCG: performance

How well do they perform?



LCG: performance

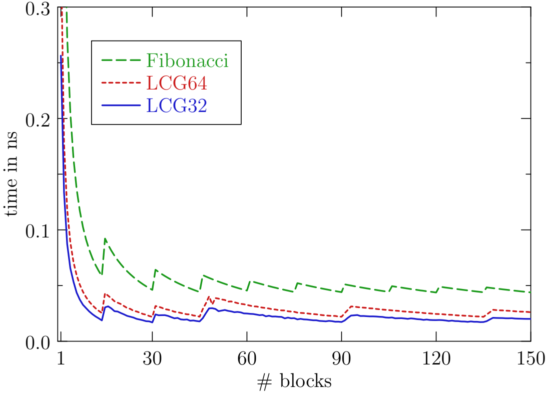
How well do they perform?



Characteristic zig-zag pattern due to commensurability (or not) of block number of with number of multiprocessors.

LCG: performance

How well do they perform?



Characteristic zig-zag pattern due to commensurability (or not) of block number of with number of multiprocessors.

Peak performance at 58×10^9 (LCG32) and 46×10^9 (LCG64) random numbers per second, respectively.

LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model.

LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison.

LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison. Test case of 1024×1024 system at $\beta = 0.4$, 10^7 sweeps.

- checkerboard update uses random numbers in different way than sequential update
- linear congruential generators can skip ahead: "right" way uses non-overlapping sub-sequences
- "wrong" way uses sequences from random initial seeds, many of which must overlap

RNG quality: Ising results

Table: Internal energy e per spin and specific heat C_V for a 1024×1024 Ising model with periodic boundary conditions at $\beta = 0.4$.

method	e	Δ_{rel}	C_V	Δ_{rel}	$t_{\mathrm{up}}^{k=1}$	$t_{\mathrm{up}}^{k=100}$
exact	1.106079207	0	0.8616983594	0		
sequential update (CPU)						
LCG32	1.1060788(15)	−0.26	0.83286(45)	−63.45		
LCG64	1.1060801(17)	0.49	0.86102(60)	−1.14		
Fibonacci, $r = 512$	1.1060789(17)	−0.18	0.86132(59)	−0.64		
checkerboard update (GPU)						
LCG32	1.0944121(14)	−8259.05	0.80316(48)	−121.05	0.2221	0.0402
LCG32, random	1.1060775(18)	−0.97	0.86175(56)	0.09	0.2221	0.0402
LCG64	1.1061058(19)	13.72	0.86179(67)	0.14	0.2311	0.0471
LCG64, random	1.1060803(18)	0.62	0.86215(63)	0.71	0.2311	0.0471
MWC, same a	1.1060800(18)	0.45	0.86161(60)	−0.15	0.2293	0.0435
MWC, different a	1.1060797(18)	0.28	0.86168(62)	−0.03	0.2336	0.0438
Fibonacci, $r = 521$	1.1060890(15)	6.43	0.86099(66)	−1.09	0.2601	0.0661
Fibonacci, $r = 1279$	1.1060800(19)	0.40	0.86084(53)	−1.64	0.2904	0.0700
XORWOW (cuRAND)	1.1060654(15)	−9.13	0.86167(65)	0.04	0.7956	0.0576
XORShift/Weyl	1.1060788(18)	−0.23	0.86184(53)	0.27	0.2613	0.0721
Philox4x32_7	1.1060778(18)	−0.79	0.86109(65)	−0.93	0.2399	0.0523
Philox4x32_10	1.1060777(17)	−0.85	0.86188(61)	0.30	0.2577	0.0622

LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison. Test case of 1024×1024 system at $\beta = 0.4$, 10^7 sweeps.

- checkerboard update uses random numbers in different way than sequential update
- linear congruential generators can skip ahead: "right" way uses non-overlapping sub-sequences
- "wrong" way uses sequences from random initial seeds, many of which must overlap

TestU01 results:

- poor for LCG32
- acceptable for LCG64

RNG quality: TestU01 results

Table: The memory footprint is measured in bits per thread. For the TestU01 results, if (too many) failures in SmallCrush are found, Crush and BigCrush are not attempted; likewise with failures in Crush. The performance column shows the peak number of 32-bit uniform floating-point random numbers produced per second on a fully loaded GTX 480 device.

generator	bits/thread	failures in TestU01			Ising test	perf.
		SmallCrush	Crush	BigCrush		$\times 10^9$/s
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	$64+32$	1	29	—	passed	44
Fibonacci, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 44	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

LCG: overall benchmarks

Use these LCG generators for the previously developed simulation code for the 2D Ising model. Exact results are available for comparison. Test case of 1024×1024 system at $\beta = 0.4$, 10^7 sweeps.

- checkerboard update uses random numbers in different way than sequential update
- linear congruential generators can skip ahead: "right" way uses non-overlapping sub-sequences
- "wrong" way uses sequences from random initial seeds, many of which must overlap

TestU01 results:

- poor for LCG32
- acceptable for LCG64

General conclusion: fast, but not good enough

Outline

Multiply-with-carry

An only slightly more complicated recursion suggested by Marsaglia is defined by

$$\begin{aligned} x_{n+1} &= a x_n + c_n \pmod{m}, \\ c_{n+1} &= \lfloor (a x_n + c_n)/m \rfloor. \end{aligned}$$

Multiply-with-carry

An only slightly more complicated recursion suggested by Marsaglia is defined by

$$x_{n+1} = ax_n + c_n \pmod{m},$$
$$c_{n+1} = \lfloor (ax_n + c_n)/m \rfloor.$$

- additive c_n is the carry of the previous iteration

Multiply-with-carry

An only slightly more complicated recursion suggested by Marsaglia is defined by

$$x_{n+1} = ax_n + c_n \pmod{m},$$
$$c_{n+1} = \lfloor (ax_n + c_n)/m \rfloor.$$

- additive c_n is the carry of the previous iteration
- for $m = 2^{32}$, we can pack the whole state in one 64-bit integer variable

Multiply-with-carry

An only slightly more complicated recursion suggested by Marsaglia is defined by

$$x_{n+1} = ax_n + c_n \pmod{m},$$
$$c_{n+1} = \lfloor (ax_n + c_n)/m \rfloor.$$

- additive c_n is the carry of the previous iteration
- for $m = 2^{32}$, we can pack the whole state in one 64-bit integer variable
- maximal period is $p = am - 2$, which can be close to the $p = 2^{64}$ of the 64-bit LCG

Multiply-with-carry

An only slightly more complicated recursion suggested by Marsaglia is defined by

$$x_{n+1} = ax_n + c_n \pmod{m},$$
$$c_{n+1} = \lfloor (ax_n + c_n)/m \rfloor.$$

- additive c_n is the carry of the previous iteration
- for $m = 2^{32}$, we can pack the whole state in one 64-bit integer variable
- maximal period is $p = am - 2$, which can be close to the $p = 2^{64}$ of the 64-bit LCG
- to achieve the full period, one requires $am - 1$ as well as $(am - 2)/2$ to be prime (such that $am - 1$ is a safe prime)

Multiply-with-carry

An only slightly more complicated recursion suggested by Marsaglia is defined by

$$x_{n+1} = ax_n + c_n \pmod{m},$$
$$c_{n+1} = \lfloor (ax_n + c_n)/m \rfloor.$$

- additive c_n is the carry of the previous iteration
- for $m = 2^{32}$, we can pack the whole state in one 64-bit integer variable
- maximal period is $p = am - 2$, which can be close to the $p = 2^{64}$ of the 64-bit LCG
- to achieve the full period, one requires $am - 1$ as well as $(am - 2)/2$ to be prime (such that $am - 1$ is a safe prime)
- $\Rightarrow$ expensive to generate *many* instances, need 64 + 32 bits of state

Multiply-with-carry

An only slightly more complicated recursion suggested by Marsaglia is defined by

$$x_{n+1} = ax_n + c_n \pmod{m},$$
$$c_{n+1} = \lfloor (ax_n + c_n)/m \rfloor.$$

- additive c_n is the carry of the previous iteration
- for $m = 2^{32}$, we can pack the whole state in one 64-bit integer variable
- maximal period is $p = am - 2$, which can be close to the $p = 2^{64}$ of the 64-bit LCG
- to achieve the full period, one requires $am - 1$ as well as $(am - 2)/2$ to be prime (such that $am - 1$ is a safe prime)
- $\Rightarrow$ expensive to generate *many* instances, need $64 + 32$ bits of state

LCG implementation

```
unsigned long long int ran;
CONVERT((unsigned int)(ran = (ran&0xffffffffull)*AMWC+ran>>32));
```

RNG quality: Ising results

Table: Internal energy e per spin and specific heat C_V for a 1024×1024 Ising model with periodic boundary conditions at $\beta = 0.4$.

method	e	$\Delta_{\rm rel}$	C_V	$\Delta_{\rm rel}$	$t_{\rm up}^{k=1}$	$t_{\rm up}^{k=100}$
exact	1.106079207	0	0.8616983594	0		
sequential update (CPU)						
LCG32	1.1060788(15)	−0.26	0.83286(45)	−63.45		
LCG64	1.1060801(17)	0.49	0.86102(60)	−1.14		
Fibonacci, $r = 512$	1.1060789(17)	−0.18	0.86132(59)	−0.64		
checkerboard update (GPU)						
LCG32	1.0944121(14)	−8259.05	0.80316(48)	−121.05	0.2221	0.0402
LCG32, random	1.1060775(18)	−0.97	0.86175(56)	0.09	0.2221	0.0402
LCG64	1.1061058(19)	13.72	0.86179(67)	0.14	0.2311	0.0471
LCG64, random	1.1060803(18)	0.62	0.86215(63)	0.71	0.2311	0.0471
MWC, same a	1.1060800(18)	0.45	0.86161(60)	−0.15	0.2293	0.0435
MWC, different a	1.1060797(18)	0.28	0.86168(62)	−0.03	0.2336	0.0438
Fibonacci, $r = 521$	1.1060890(15)	6.43	0.86099(66)	−1.09	0.2601	0.0661
Fibonacci, $r = 1279$	1.1060800(19)	0.40	0.86084(53)	−1.64	0.2904	0.0700
XORWOW (cuRAND)	1.1060654(15)	−9.13	0.86167(65)	0.04	0.7956	0.0576
XORShift/Weyl	1.1060788(18)	−0.23	0.86184(53)	0.27	0.2613	0.0721
Philox4x32_7	1.1060778(18)	−0.79	0.86109(65)	−0.93	0.2399	0.0523
Philox4x32_10	1.1060777(17)	−0.85	0.86188(61)	0.30	0.2577	0.0622

RNG quality: TestU01 results

Table: The memory footprint is measured in bits per thread. For the TestU01 results, if (too many) failures in SmallCrush are found, Crush and BigCrush are not attempted; likewise with failures in Crush. The performance column shows the peak number of 32-bit uniform floating-point random numbers produced per second on a fully loaded GTX 480 device.

generator	bits/thread	failures in TestU01			Ising test	perf.
		SmallCrush	Crush	BigCrush		$\times 10^9$/s
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	$64 + 32$	1	29	—	passed	44
Fibonacci, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 44	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

Outline

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$
- state size $32 \times r$ bits (for $r > s$) $\Rightarrow$ use state sharing to reduce effective memory requirements

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$
- state size $32 \times r$ bits (for $r > s$) $\Rightarrow$ use state sharing to reduce effective memory requirements
- for $\otimes = +$ maximal period is $p = 2^r - 1$

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$
- state size $32 \times r$ bits (for $r > s$) $\Rightarrow$ use state sharing to reduce effective memory requirements
- for $\otimes = +$ maximal period is $p = 2^r - 1$
- can be implemented directly in floating point arithmetic, $u_n = u_{n-r} + u_{n-s} \pmod{1}$.

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$
- state size $32 \times r$ bits (for $r > s$) $\Rightarrow$ use state sharing to reduce effective memory requirements
- for $\otimes = +$ maximal period is $p = 2^r - 1$
- can be implemented directly in floating point arithmetic, $u_n = u_{n-r} + u_{n-s} \pmod{1}$.
- s random numbers can be generated in one vectorized call

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$
- state size $32 \times r$ bits (for $r > s$) $\Rightarrow$ use state sharing to reduce effective memory requirements
- for $\otimes = +$ maximal period is $p = 2^r - 1$
- can be implemented directly in floating point arithmetic, $u_n = u_{n-r} + u_{n-s} \pmod{1}$.
- s random numbers can be generated in one vectorized call
- choose, e.g., $r = 521$, $s = 353$ and $r = 1279$, $s = 861$, the latter with period $p \approx 10^{394}$

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$
- state size $32 \times r$ bits (for $r > s$) $\Rightarrow$ use state sharing to reduce effective memory requirements
- for $\otimes = +$ maximal period is $p = 2^r - 1$
- can be implemented directly in floating point arithmetic, $u_n = u_{n-r} + u_{n-s} \pmod{1}$.
- s random numbers can be generated in one vectorized call
- choose, e.g., $r = 521$, $s = 353$ and $r = 1279$, $s = 861$, the latter with period $p \approx 10^{394}$
- memory requirement $(r + s)/n$ words per thread

Lagged Fibonacci RNG

Longer period can only be achieved with larger state, e.g.,

$$x_n = a_s x_{n-s} \otimes a_r x_{n-r} \pmod{m},$$

- operator $\otimes$ typically denotes one of the four operations addition $+$, subtraction $-$, multiplication $*$ and bitwise XOR $\oplus$
- state size $32 \times r$ bits (for $r > s$) $\Rightarrow$ use state sharing to reduce effective memory requirements
- for $\otimes = +$ maximal period is $p = 2^r - 1$
- can be implemented directly in floating point arithmetic, $u_n = u_{n-r} + u_{n-s} \pmod{1}$.
- s random numbers can be generated in one vectorized call
- choose, e.g., $r = 521$, $s = 353$ and $r = 1279$, $s = 861$, the latter with period $p \approx 10^{394}$
- memory requirement $(r + s)/n$ words per thread
- can use skipping or random seeds

RNG quality: Ising results

Table: Internal energy e per spin and specific heat C_V for a 1024×1024 Ising model with periodic boundary conditions at $\beta = 0.4$.

method	e	$\Delta_{\rm rel}$	C_V	$\Delta_{\rm rel}$	$t_{\rm up}^{k=1}$	$t_{\rm up}^{k=100}$
exact	1.106079207	0	0.8616983594	0		
sequential update (CPU)						
LCG32	1.1060788(15)	−0.26	0.83286(45)	−63.45		
LCG64	1.1060801(17)	0.49	0.86102(60)	−1.14		
Fibonacci, $r = 512$	1.1060789(17)	−0.18	0.86132(59)	−0.64		
checkerboard update (GPU)						
LCG32	1.0944121(14)	−8259.05	0.80316(48)	−121.05	0.2221	0.0402
LCG32, random	1.1060775(18)	−0.97	0.86175(56)	0.09	0.2221	0.0402
LCG64	1.1061058(19)	13.72	0.86179(67)	0.14	0.2311	0.0471
LCG64, random	1.1060803(18)	0.62	0.86215(63)	0.71	0.2311	0.0471
MWC, same a	1.1060800(18)	0.45	0.86161(60)	−0.15	0.2293	0.0435
MWC, different a	1.1060797(18)	0.28	0.86168(62)	−0.03	0.2336	0.0438
Fibonacci, $r = 521$	1.1060890(15)	6.43	0.86099(66)	−1.09	0.2601	0.0661
Fibonacci, $r = 1279$	1.1060800(19)	0.40	0.86084(53)	−1.64	0.2904	0.0700
XORWOW (cuRAND)	1.1060654(15)	−9.13	0.86167(65)	0.04	0.7956	0.0576
XORShift/Weyl	1.1060788(18)	−0.23	0.86184(53)	0.27	0.2613	0.0721
Philox4x32_7	1.1060778(18)	−0.79	0.86109(65)	−0.93	0.2399	0.0523
Philox4x32_10	1.1060777(17)	−0.85	0.86188(61)	0.30	0.2577	0.0622

RNG quality: TestU01 results

Table: The memory footprint is measured in bits per thread. For the TestU01 results, if (too many) failures in SmallCrush are found, Crush and BigCrush are not attempted; likewise with failures in Crush. The performance column shows the peak number of 32-bit uniform floating-point random numbers produced per second on a fully loaded GTX 480 device.

generator	bits/thread	failures in TestU01			Ising test	perf.
		SmallCrush	Crush	BigCrush		$\times 10^9$/s
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	64 + 32	1	29	—	passed	44
Fibonacci, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 44	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

Outline

Mersenne twister

See:

M. Mansen, M. Weigel, and A. K. Hartmann, Eur. Phys. J. Special Topics 210, 53 (2012.)

Outline

XORShift: definition

Another generator proposed by Marsaglia based on the observation that an XOR of a word with a shifted version of itself can be performed very fast.

XORShift: definition

Another generator proposed by Marsaglia based on the observation that an XOR of a word with a shifted version of itself can be performed very fast. The suggested recursion is

$$x_n = x_{n-1}(I \oplus L^a)(I \oplus R^b)(I \oplus L^c) =: x_{n-1}M,$$

where L^a and R^b denote left shift by a bits and right shift by b positions, respectively.

XORShift: definition

Another generator proposed by Marsaglia based on the observation that an XOR of a word with a shifted version of itself can be performed very fast. The suggested recursion is

$$x_n = x_{n-1}(I \oplus L^a)(I \oplus R^b)(I \oplus L^c) =: x_{n-1}M,$$

where L^a and R^b denote left shift by a bits and right shift by b positions, respectively.

- maximum period is $p = 2^w - 1$, where w is the number of bits in x

XORShift: definition

Another generator proposed by Marsaglia based on the observation that an XOR of a word with a shifted version of itself can be performed very fast. The suggested recursion is

$$x_n = x_{n-1}(I \oplus L^a)(I \oplus R^b)(I \oplus L^c) =: x_{n-1}M,$$

where L^a and R^b denote left shift by a bits and right shift by b positions, respectively.

- maximum period is $p = 2^w - 1$, where w is the number of bits in x
- the combination of $w = 160$ with a Weyl generator defines XORWOW included in CUDA (state is already too large)

XORShift: definition

Another generator proposed by Marsaglia based on the observation that an XOR of a word with a shifted version of itself can be performed very fast. The suggested recursion is

$$x_n = x_{n-1}(I \oplus L^a)(I \oplus R^b)(I \oplus L^c) =: x_{n-1}M,$$

where L^a and R^b denote left shift by a bits and right shift by b positions, respectively.

- maximum period is $p = 2^w - 1$, where w is the number of bits in x
- the combination of $w = 160$ with a Weyl generator defines XORWOW included in CUDA (state is already too large)
- instead, use $w = 1024$ and employ state sharing again, using the one 32-bit word for each of the 32 threads of a warp

XORShift: definition

Another generator proposed by Marsaglia based on the observation that an XOR of a word with a shifted version of itself can be performed very fast. The suggested recursion is

$$x_n = x_{n-1}(I \oplus L^a)(I \oplus R^b)(I \oplus L^c) =: x_{n-1} M,$$

where L^a and R^b denote left shift by a bits and right shift by b positions, respectively.

- maximum period is $p = 2^w - 1$, where w is the number of bits in x
- the combination of $w = 160$ with a Weyl generator defines XORWOW included in CUDA (state is already too large)
- instead, use $w = 1024$ and employ state sharing again, using the one 32-bit word for each of the 32 threads of a warp
- with appropriate parameters, period is $2^{1024} - 1$

XORShift: definition

Another generator proposed by Marsaglia based on the observation that an XOR of a word with a shifted version of itself can be performed very fast. The suggested recursion is

$$x_n = x_{n-1}(I \oplus L^a)(I \oplus R^b)(I \oplus L^c) =: x_{n-1}M,$$

where L^a and R^b denote left shift by a bits and right shift by b positions, respectively.

- maximum period is $p = 2^w - 1$, where w is the number of bits in x
- the combination of $w = 160$ with a Weyl generator defines XORWOW included in CUDA (state is already too large)
- instead, use $w = 1024$ and employ state sharing again, using the one 32-bit word for each of the 32 threads of a warp
- with appropriate parameters, period is $2^{1024} - 1$
- shifts can be implemented efficiently over word boundaries using padding of the state array

XORShift: implementation

We use $a = 329$, $b = 347$ and $c = 344$, such that `WORDSHIFT`$= \lfloor a/32 \rfloor = 10$.

XORShift: implementation

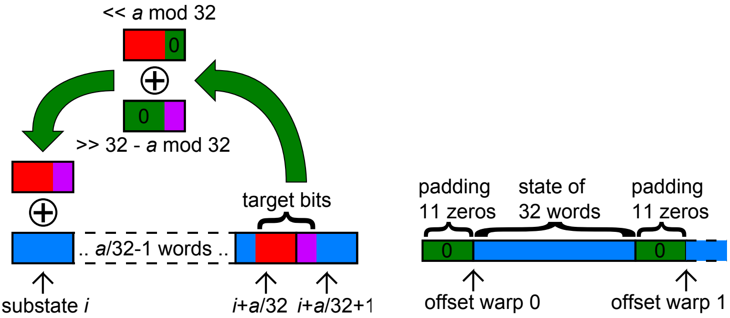
We use $a = 329$, $b = 347$ and $c = 344$, such that WORDSHIFT$= \lfloor a/32 \rfloor = 10$.

LCG implementation

```
__device__ state_t rng_update(state_t state, int tid,
                              volatile state_t* stateblock)
{
/* Indices. */
int wid = tid / WARPSIZE; // Warp index in block
int lid = tid % WARPSIZE; // Thread index in warp
int woff = wid * (WARPSIZE + WORDSHIFT + 1) + WORDSHIFT + 1;
                                          // warp offset
/* Shifted indices. */
int lp = lid + WORDSHIFT; // Left word shift
int lm = lid - WORDSHIFT; // Right word shift

/* << A. */
stateblock[woff + lid] = state; // Share states
state ^= stateblock[woff + lp] << RAND_A; // Left part
state ^= stateblock[woff + lp + 1] >> WORD - RAND_A; // Right part

/* >> B. */
stateblock[woff + lid] = state; // Share states
state ^= stateblock[woff + lm - 1] << WORD - RAND_B; // Left part
state ^= stateblock[woff + lm] >> RAND_B; // Right part

/* << C. */
stateblock[woff + lid] = state; // Share states
state ^= stateblock[woff + lp] << RAND_C; // Left part
state ^= stateblock[woff + lp + 1] >> WORD - RAND_C; // Right part

return state;
}
```

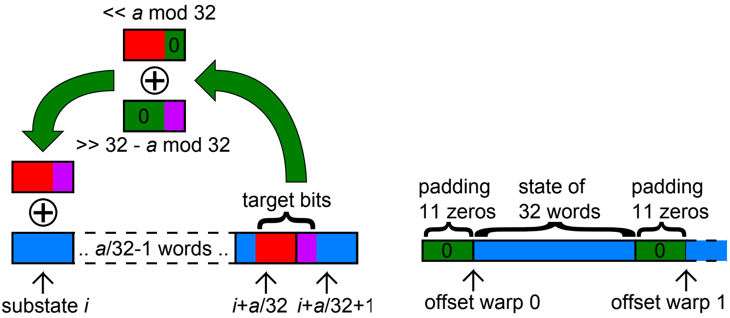
XORShift: implementation

We use $a = 329$, $b = 347$ and $c = 344$, such that WORDSHIFT$= \lfloor a/32 \rfloor = 10$.



XORShift: implementation

We use $a = 329$, $b = 347$ and $c = 344$, such that WORDSHIFT$= \lfloor a/32 \rfloor = 10$.



- due to single-thread scheduling, no thread synchronization is required
- use `volatile` keyword to ensure writes
- use skip-ahead to create sub-streams
- combine with Weyl generator, $y_n = (y_{n-1} + c) \mod 2^w$, to further improve quality

RNG quality: Ising results

Table: Internal energy e per spin and specific heat C_V for a 1024×1024 Ising model with periodic boundary conditions at $\beta = 0.4$.

method	e	Δ_{rel}	C_V	Δ_{rel}	$t_{\mathrm{up}}^{k=1}$	$t_{\mathrm{up}}^{k=100}$
exact	1.106079207	0	0.8616983594	0		
sequential update (CPU)						
LCG32	1.1060788(15)	−0.26	0.83286(45)	−63.45		
LCG64	1.1060801(17)	0.49	0.86102(60)	−1.14		
Fibonacci, $r = 512$	1.1060789(17)	−0.18	0.86132(59)	−0.64		
checkerboard update (GPU)						
LCG32	1.0944121(14)	−8259.05	0.80316(48)	−121.05	0.2221	0.0402
LCG32, random	1.1060775(18)	−0.97	0.86175(56)	0.09	0.2221	0.0402
LCG64	1.1061058(19)	13.72	0.86179(67)	0.14	0.2311	0.0471
LCG64, random	1.1060803(18)	0.62	0.86215(63)	0.71	0.2311	0.0471
MWC, same a	1.1060800(18)	0.45	0.86161(60)	−0.15	0.2293	0.0435
MWC, different a	1.1060797(18)	0.28	0.86168(62)	−0.03	0.2336	0.0438
Fibonacci, $r = 521$	1.1060890(15)	6.43	0.86099(66)	−1.09	0.2601	0.0661
Fibonacci, $r = 1279$	1.1060800(19)	0.40	0.86084(53)	−1.64	0.2904	0.0700
XORWOW (cuRAND)	1.1060654(15)	−9.13	0.86167(65)	0.04	0.7956	0.0576
XORShift/Weyl	1.1060788(18)	−0.23	0.86184(53)	0.27	0.2613	0.0721
Philox4x32_7	1.1060778(18)	−0.79	0.86109(65)	−0.93	0.2399	0.0523
Philox4x32_10	1.1060777(17)	−0.85	0.86188(61)	0.30	0.2577	0.0622

RNG quality: TestU01 results

Table: The memory footprint is measured in bits per thread. For the TestU01 results, if (too many) failures in SmallCrush are found, Crush and BigCrush are not attempted; likewise with failures in Crush. The performance column shows the peak number of 32-bit uniform floating-point random numbers produced per second on a fully loaded GTX 480 device.

generator	bits/thread	failures in TestU01			Ising test	perf.
		SmallCrush	Crush	BigCrush		$\times 10^9$/s
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	64 + 32	1	29	—	passed	44
Fibonacci, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 44	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

Outline

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad.

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad. If f_k is bijective, the period is 2^w.

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad. If f_k is bijective, the period is 2^w.

Are there better choices for f_k?

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad. If f_k is bijective, the period is 2^w.

Are there better choices for f_k? Yes, for instance cryptographic functions that are (a) bijective,

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad. If f_k is bijective, the period is 2^w.

Are there better choices for f_k? Yes, for instance cryptographic functions that are (a) bijective, (b) depend on a key k, and

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad. If f_k is bijective, the period is 2^w.

Are there better choices for f_k? Yes, for instance cryptographic functions that are (a) bijective, (b) depend on a key k, and (c) translate the plaintext n into the ciphertext x_n.

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad. If f_k is bijective, the period is 2^w.

Are there better choices for f_k? Yes, for instance cryptographic functions that are (a) bijective, (b) depend on a key k, and (c) translate the plaintext n into the ciphertext x_n. By definition, if x_n contains any structure that makes it differ from a random sequence of bits, the cipher is susceptible to an attack.

Cryptographic generators

For the Weyl generator above, we can evaluate the n element in one step,

$$y_n = (y_0 + nc) \mod 2^w.$$

This can be interpreted as applying a simple, bijective function to a counter n,

$$x_n = f_k(n).$$

Here, skip-ahead is trivial. Unfortunately, the quality of the Weyl sequence is very bad. If f_k is bijective, the period is 2^w.

Are there better choices for f_k? Yes, for instance cryptographic functions that are (a) bijective, (b) depend on a key k, and (c) translate the plaintext n into the ciphertext x_n. By definition, if x_n contains any structure that makes it differ from a random sequence of bits, the cipher is susceptible to an attack.

Well-known and proven symmetric-key cryptosystems are DES and AES.

Excursion: simplified DES

DES is a block cipher, where each block is encrypted separately. Consider a single block

L_0	R_0
6 bits	6 bits

of 12 bits.

Excursion: simplified DES

DES is a block cipher, where each block is encrypted separately. Consider a single block

L_0	R_0
6 bits	6 bits

of 12 bits. Encryption works iteratively, where in the ith round an 8-bit key K_i is used to transform $L_{i-1}R_{i-1}$ to the output L_iR_i as follows

Excursion: simplified DES

DES is a block cipher, where each block is encrypted separately. Consider a single block

L_0	R_0
6 bits	6 bits

of 12 bits. Encryption works iteratively, where in the ith round an 8-bit key K_i is used to transform $L_{i-1}R_{i-1}$ to the output L_iR_i as follows

$$L_i = R_{i-1}, \quad R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

where $\oplus$ denotes XOR or bitwise addition modulo 2.

Excursion: simplified DES

DES is a block cipher, where each block is encrypted separately. Consider a single block

L_0	R_0
6 bits	6 bits

of 12 bits. Encryption works iteratively, where in the ith round an 8-bit key K_i is used to transform $L_{i-1}R_{i-1}$ to the output L_iR_i as follows

$$L_i = R_{i-1}, \quad R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

where $\oplus$ denotes XOR or bitwise addition modulo 2.

After n rounds (known as Feistel iterations), we have L_nR_n. To decrypt, switch to R_nL_n and use the keys in reverse order,

$$[L_n][R_n \oplus f(L_n, K_n)].$$

Excursion: simplified DES

DES is a block cipher, where each block is encrypted separately. Consider a single block

L_0	R_0
6 bits	6 bits

of 12 bits. Encryption works iteratively, where in the ith round an 8-bit key K_i is used to transform $L_{i-1}R_{i-1}$ to the output L_iR_i as follows

$$L_i = R_{i-1}, \quad R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

where $\oplus$ denotes XOR or bitwise addition modulo 2.

After n rounds (known as Feistel iterations), we have L_nR_n. To decrypt, switch to R_nL_n and use the keys in reverse order,

$$[L_n][R_n \oplus f(L_n, K_n)].$$

From encryption we know $L_n = R_{n-1}$, $R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$ and hence

$$[L_n][R_n \oplus f(L_n, K_n)] = [R_{n-1}]\;[L_{n-1} \oplus f(R_{n-1}, K_n) \oplus f(L_n, K_n)] = [R_{n-1}]\;[L_{n-1}],$$

where $f(R_{n-1}, K_n) \oplus f(L_n, K_n) = 0$ since $L_n = R_{n-1}$.

Excursion: simplified DES

DES is a block cipher, where each block is encrypted separately. Consider a single block

L_0	R_0
6 bits	6 bits

of 12 bits. Encryption works iteratively, where in the ith round an 8-bit key K_i is used to transform $L_{i-1}R_{i-1}$ to the output L_iR_i as follows

$$L_i = R_{i-1}, \quad R_i = L_{i-1} \oplus f(R_{i-1}, K_i),$$

where $\oplus$ denotes XOR or bitwise addition modulo 2.

After n rounds (known as Feistel iterations), we have L_nR_n. To decrypt, switch to R_nL_n and use the keys in reverse order,

$$[L_n][R_n \oplus f(L_n, K_n)].$$

From encryption we know $L_n = R_{n-1}$, $R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$ and hence

$$[L_n][R_n \oplus f(L_n, K_n)] = [R_{n-1}]\,[L_{n-1} \oplus f(R_{n-1}, K_n) \oplus f(L_n, K_n)] = [R_{n-1}]\,[L_{n-1}],$$

where $f(R_{n-1}, K_n) \oplus f(L_n, K_n) = 0$ since $L_n = R_{n-1}$. Continuing with the key sequence K_n, K_{n-1}, ..., K_0, we arrive at R_0L_0 and hence L_0R_0.

Excursion: simplified DES (cont'd)

One advantage of this procedure is that encryption and decryption are almost identical, use the same keys and can hence use the same hardware.

Excursion: simplified DES (cont'd)

One advantage of this procedure is that encryption and decryption are almost identical, use the same keys and can hence use the same hardware.

How should we choose f?

Excursion: simplified DES (cont'd)

One advantage of this procedure is that encryption and decryption are almost identical, use the same keys and can hence use the same hardware.

How should we choose f? Obviously, it should not be "nice", e.g., linear and bijective.

Excursion: simplified DES (cont'd)

One advantage of this procedure is that encryption and decryption are almost identical, use the same keys and can hence use the same hardware.

How should we choose f? Obviously, it should not be "nice", e.g., linear and bijective. We use the following combination

- The 6-bit input R_{i-1} is sent through an expander function,

$$e(m_1 m_2 m_3 m_4 m_5 m_6) = m_1 m_2 m_4 m_3 \; m_4 m_3 m_5 m_6,$$

  yielding 8 bits.

Excursion: simplified DES (cont'd)

One advantage of this procedure is that encryption and decryption are almost identical, use the same keys and can hence use the same hardware.

How should we choose f? Obviously, it should not be "nice", e.g., linear and bijective. We use the following combination

- The 6-bit input R_{i-1} is sent through an expander function,

$$e(m_1 m_2 m_3 m_4 m_5 m_6) = m_1 m_2 m_4 m_3 \; m_4 m_3 m_5 m_6,$$

  yielding 8 bits.
- We derive the key K_i for round i from $K = k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7 k_8$ by

$$K_i = k_{(i-1) \mod 9} k_{i \mod 9} k_{(i+1) \mod 9} \cdots k_{(i+6) \mod 9}.$$

Excursion: simplified DES (cont'd)

One advantage of this procedure is that encryption and decryption are almost identical, use the same keys and can hence use the same hardware.

How should we choose f? Obviously, it should not be "nice", e.g., linear and bijective. We use the following combination

- The 6-bit input R_{i-1} is sent through an expander function,

$$e(m_1 m_2 m_3 m_4 m_5 m_6) = m_1 m_2 m_4 m_3\ m_4 m_3 m_5 m_6,$$

  yielding 8 bits.

- We derive the key K_i for round i from $K = k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7 k_8$ by

$$K_i = k_{(i-1) \mod 9} k_{i \mod 9} k_{(i+1) \mod 9} \cdots k_{(i+6) \mod 9}.$$

  The 8 bits from the expander are then XORed with K_i.

Excursion: simplified DES (cont'd)

One advantage of this procedure is that encryption and decryption are almost identical, use the same keys and can hence use the same hardware.

How should we choose f? Obviously, it should not be "nice", e.g., linear and bijective. We use the following combination

- The 6-bit input R_{i-1} is sent through an expander function,

  $$e(m_1 m_2 m_3 m_4 m_5 m_6) = m_1 m_2 m_4 m_3\; m_4 m_3 m_5 m_6,$$

  yielding 8 bits.
- We derive the key K_i for round i from $K = k_0 k_1 k_2 k_3 k_4 k_5 k_6 k_7 k_8$ by

  $$K_i = k_{(i-1) \mod 9} k_{i \mod 9} k_{(i+1) \mod 9} \cdots k_{(i+6) \mod 9}.$$

  The 8 bits from the expander are then XORed with K_i.
- In a third step, these 8 bits are passed through one of two S-boxes,

  $$S_1 = \begin{bmatrix} 101 & 010 & 001 & 110 & 011 & 100 & 111 & 000 \\ 001 & 100 & 110 & 010 & 000 & 111 & 101 & 011 \end{bmatrix}$$

  $$S_2 = \begin{bmatrix} 100 & 000 & 110 & 101 & 111 & 001 & 011 & 010 \\ 101 & 011 & 000 & 111 & 110 & 010 & 001 & 100 \end{bmatrix}$$

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column.

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Then, 1100 is sent to S_1. The second row, fifth column contains 000. The second part 1111 is sent to S_2, yielding 100.

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Then, 1100 is sent to S_1. The second row, fifth column contains 000. The second part 1111 is sent to S_2, yielding 100. Hence the total output is $f(R_{i-1}, K_i) = 000100$.

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Then, 1100 is sent to S_1. The second row, fifth column contains 000. The second part 1111 is sent to S_2, yielding 100. Hence the total output is $f(R_{i-1}, K_i) = 000100$. In total, we have

$$\begin{array}{cccccccc} L_{i-1} & R_{i-1} & \mapsto & R_{i-1} & L_{i-1} & \oplus & f(R_{i-1}, K_i) \\ 011100 & 100110 & & 100110 & 011100 & \oplus & 000100 \\ 011100 & 100110 & & 100110 & & 011000 & \end{array}$$

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Then, 1100 is sent to S_1. The second row, fifth column contains 000. The second part 1111 is sent to S_2, yielding 100. Hence the total output is $f(R_{i-1}, K_i) = 000100$. In total, we have

$$\begin{array}{ccccccc} L_{i-1} & R_{i-1} & \mapsto & R_{i-1} & L_{i-1} & \oplus & f(R_{i-1}, K_i) \\ 011100 & 100110 & & 100110 & 011100 & \oplus & 000100 \\ 011100 & 100110 & & 100110 & & 011000 & \end{array}$$

Breaking DES

A successful approach to cryptosystems of the DES type is differential cryptanalysis (which was suggested by Biham and Shamir in 1990 but was, in fact, known to the inventor of DES in 1979):

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Then, 1100 is sent to S_1. The second row, fifth column contains 000. The second part 1111 is sent to S_2, yielding 100. Hence the total output is $f(R_{i-1}, K_i) = 000100$. In total, we have

$$\begin{array}{cccccccc} L_{i-1} & R_{i-1} & \mapsto & R_{i-1} & L_{i-1} & \oplus & f(R_{i-1}, K_i) \\ 011100 & 100110 & & 100110 & 011100 & \oplus & 000100 \\ 011100 & 100110 & & 100110 & & 011000 & \end{array}$$

Breaking DES

A successful approach to cryptosystems of the DES type is differential cryptanalysis (which was suggested by Biham and Shamir in 1990 but was, in fact, known to the inventor of DES in 1979): the idea is to compare the differences in ciphertexts for suitably chosen pairs of plaintext.

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Then, 1100 is sent to S_1. The second row, fifth column contains 000. The second part 1111 is sent to S_2, yielding 100. Hence the total output is $f(R_{i-1}, K_i) = 000100$. In total, we have

$$\begin{array}{cccccccc}
L_{i-1} & R_{i-1} & \mapsto & R_{i-1} & L_{i-1} & \oplus & f(R_{i-1}, K_i) \\
011100 & 100110 & & 100110 & 011100 & \oplus & 000100 \\
011100 & 100110 & & 100110 & & 011000 &
\end{array}$$

Breaking DES

A successful approach to cryptosystems of the DES type is differential cryptanalysis (which was suggested by Biham and Shamir in 1990 but was, in fact, known to the inventor of DES in 1979): the idea is to compare the differences in ciphertexts for suitably chosen pairs of plaintext. It has been shown that for DES this is no better than a brute force attack.

Excursion: simplified DES (cont'd)

For the S-boxes, the 8 bits from step two are broken into two 4-bit parts. The first part is sent to S_1 and the second part to S_2. The first bit of each part selects the row in the S-box, the remaining three bits the column. Altogether, we have, e.g., for $R_{i-1} = 100110$ and $K_i = 01100101$

$$e(100110) \oplus K_i = 10101010 \oplus 01100101 = 1100\,1111.$$

Then, 1100 is sent to S_1. The second row, fifth column contains 000. The second part 1111 is sent to S_2, yielding 100. Hence the total output is $f(R_{i-1}, K_i) = 000100$. In total, we have

$$\begin{array}{cccccccc} L_{i-1} & R_{i-1} & \mapsto & R_{i-1} & L_{i-1} & \oplus & f(R_{i-1}, K_i) \\ 011100 & 100110 & & 100110 & 011100 & \oplus & 000100 \\ 011100 & 100110 & & 100110 & & 011000 & \end{array}$$

Breaking DES

A successful approach to cryptosystems of the DES type is differential cryptanalysis (which was suggested by Biham and Shamir in 1990 but was, in fact, known to the inventor of DES in 1979): the idea is to compare the differences in ciphertexts for suitably chosen pairs of plaintext. It has been shown that for DES this is no better than a brute force attack. A more sophisticated approach is linear cryptanalysis which attempts to find a linear approximation to the function f. This is better than brute force.

Philox et al.

DES and AES can be used as RNGs, and there are even modern CPUs that implement them in hardware.

Philox et al.

DES and AES can be used as RNGs, and there are even modern CPUs that implement them in hardware. Then, they are very fast.

Philox et al.

DES and AES can be used as RNGs, and there are even modern CPUs that implement them in hardware. Then, they are very fast.

As an alternative due to Salmon *et al.*, consider simplified iteration in the spirit of AES. The following,

$$\mathrm{mulhi}(a, b) = \lfloor (a \times b)/2^w \rfloor,$$
$$\mathrm{mullo}(a, b) = (a \times b) \mod 2^w,$$

is very fast on most architectures.

Philox et al.

DES and AES can be used as RNGs, and there are even modern CPUs that implement them in hardware. Then, they are very fast.

As an alternative due to Salmon *et al.*, consider simplified iteration in the spirit of AES. The following,

$$\begin{aligned} \mathrm{mulhi}(a, b) &= \lfloor (a \times b)/2^w \rfloor, \\ \mathrm{mullo}(a, b) &= (a \times b) \mod 2^w, \end{aligned}$$

is very fast on most architectures. Then, pick two words (L, R) out of N and define an S-box

$$\begin{aligned} L' &= \mathrm{mullo}(R, M), \\ R' &= \mathrm{mulhi}(R, M) \oplus k \oplus L, \end{aligned}$$

and perform r Feistel iterations on $N/2$ such S-boxes with constant key k (use permutations, or P-boxes in between the S-box applications).

Philox et al.

DES and AES can be used as RNGs, and there are even modern CPUs that implement them in hardware. Then, they are very fast.

As an alternative due to Salmon *et al.*, consider simplified iteration in the spirit of AES. The following,

$$\text{mulhi}(a, b) = \lfloor (a \times b)/2^w \rfloor,$$
$$\text{mullo}(a, b) = (a \times b) \mod 2^w,$$

is very fast on most architectures. Then, pick two words (L, R) out of N and define an S-box

$$L' = \text{mullo}(R, M),$$
$$R' = \text{mulhi}(R, M) \oplus k \oplus L,$$

and perform r Feistel iterations on $N/2$ such S-boxes with constant key k (use permutations, or P-boxes in between the S-box applications). This generates a bijection of the desired form.

Philox et al.

DES and AES can be used as RNGs, and there are even modern CPUs that implement them in hardware. Then, they are very fast.

As an alternative due to Salmon *et al.*, consider simplified iteration in the spirit of AES. The following,

$$\begin{aligned} \mathrm{mulhi}(a,b) &= \lfloor (a \times b)/2^w \rfloor, \\ \mathrm{mullo}(a,b) &= (a \times b) \mod 2^w, \end{aligned}$$

is very fast on most architectures. Then, pick two words (L, R) out of N and define an S-box

$$\begin{aligned} L' &= \mathrm{mullo}(R, M), \\ R' &= \mathrm{mulhi}(R, M) \oplus k \oplus L, \end{aligned}$$

and perform r Feistel iterations on $N/2$ such S-boxes with constant key k (use permutations, or P-boxes in between the S-box applications). This generates a bijection of the desired form. It defines a class of RNGs dubbed

Philox-Nxw_r

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance
- quality can be tuned with varying r

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance
- quality can be tuned with varying r
- depending on the implementation, the generator can be very fast

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance
- quality can be tuned with varying r
- depending on the implementation, the generator can be very fast
- the generator does not have a state per se as it is counter based; this significantly reduces bus pressure in parallel environments

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance
- quality can be tuned with varying r
- depending on the implementation, the generator can be very fast
- the generator does not have a state per se as it is counter based; this significantly reduces bus pressure in parallel environments
- different keys can be used to generate independent sequences of random numbers; 64-bit keys allow for 2^{64} independent sequences

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance
- quality can be tuned with varying r
- depending on the implementation, the generator can be very fast
- the generator does not have a state per se as it is counter based; this significantly reduces bus pressure in parallel environments
- different keys can be used to generate independent sequences of random numbers; 64-bit keys allow for 2^{64} independent sequences
- can use intrinsic variables such as particle number, temperature, disorder index, etc. to select sequences

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance
- quality can be tuned with varying r
- depending on the implementation, the generator can be very fast
- the generator does not have a state per se as it is counter based; this significantly reduces bus pressure in parallel environments
- different keys can be used to generate independent sequences of random numbers; 64-bit keys allow for 2^{64} independent sequences
- can use intrinsic variables such as particle number, temperature, disorder index, etc. to select sequences
- counter could be iteration number in Monte Carlo

Philox et al.: properties

- it is found empirically that for 4×32 bits, 7 Feistel iterations are sufficient to achieve Crush-resistance
- quality can be tuned with varying r
- depending on the implementation, the generator can be very fast
- the generator does not have a state per se as it is counter based; this significantly reduces bus pressure in parallel environments
- different keys can be used to generate independent sequences of random numbers; 64-bit keys allow for 2^{64} independent sequences
- can use intrinsic variables such as particle number, temperature, disorder index, etc. to select sequences
- counter could be iteration number in Monte Carlo
- this ensures identical results independent of the parallel setup

RNG quality: Ising results

Table: Internal energy e per spin and specific heat C_V for a 1024×1024 Ising model with periodic boundary conditions at $\beta = 0.4$.

method	e	Δ_{rel}	C_V	Δ_{rel}	$t_{\mathrm{up}}^{k=1}$	$t_{\mathrm{up}}^{k=100}$
exact	1.106079207	0	0.8616983594	0		
sequential update (CPU)						
LCG32	1.1060788(15)	−0.26	0.83286(45)	−63.45		
LCG64	1.1060801(17)	0.49	0.86102(60)	−1.14		
Fibonacci, $r = 512$	1.1060789(17)	−0.18	0.86132(59)	−0.64		
checkerboard update (GPU)						
LCG32	1.0944121(14)	−8259.05	0.80316(48)	−121.05	0.2221	0.0402
LCG32, random	1.1060775(18)	−0.97	0.86175(56)	0.09	0.2221	0.0402
LCG64	1.1061058(19)	13.72	0.86179(67)	0.14	0.2311	0.0471
LCG64, random	1.1060803(18)	0.62	0.86215(63)	0.71	0.2311	0.0471
MWC, same a	1.1060800(18)	0.45	0.86161(60)	−0.15	0.2293	0.0435
MWC, different a	1.1060797(18)	0.28	0.86168(62)	−0.03	0.2336	0.0438
Fibonacci, $r = 521$	1.1060890(15)	6.43	0.86099(66)	−1.09	0.2601	0.0661
Fibonacci, $r = 1279$	1.1060800(19)	0.40	0.86084(53)	−1.64	0.2904	0.0700
XORWOW (cuRAND)	1.1060654(15)	−9.13	0.86167(65)	0.04	0.7956	0.0576
XORShift/Weyl	1.1060788(18)	−0.23	0.86184(53)	0.27	0.2613	0.0721
Philox4x32_7	1.1060778(18)	−0.79	0.86109(65)	−0.93	0.2399	0.0523
Philox4x32_10	1.1060777(17)	−0.85	0.86188(61)	0.30	0.2577	0.0622

RNG quality: TestU01 results

Table: The memory footprint is measured in bits per thread. For the TestU01 results, if (too many) failures in SmallCrush are found, Crush and BigCrush are not attempted; likewise with failures in Crush. The performance column shows the peak number of 32-bit uniform floating-point random numbers produced per second on a fully loaded GTX 480 device.

generator	bits/thread	failures in TestU01			Ising test	perf.
		SmallCrush	Crush	BigCrush		$\times 10^9$/s
LCG32	32	12	—	—	failed	58
LCG32, random	32	3	14	—	passed	58
LCG64	64	None	6	—	failed	46
LCG64, random	64	None	2	8	passed	46
MWC	$64 + 32$	1	29	—	passed	44
Fibonacci, $r = 521$	≥ 80	None	2	—	failed	23
Fibonacci, $r = 1279$	≥ 80	None	(1)	2	passed	23
XORWOW (cuRAND)	192	None	None	1/3	failed	19
MTGP (cuRAND)	≥ 44	None	2	2	—	18
XORShift/Weyl	32	None	None	None	passed	18
Philox4x32_7	(128)	None	None	None	passed	41
Philox4x32_10	(128)	None	None	None	passed	30

Summary and outlook

This lecture

This lecture has given a survey of random number generators in a massively parallel environment. On GPUs, we need a massive number of independent RNGs with small state. Two strategies have been explored: individual generators with small states which, however, suffer from small periods and state-sharing among several instances. An independent alternative are counter-based generators.

Summary and outlook

This lecture

This lecture has given a survey of random number generators in a massively parallel environment. On GPUs, we need a massive number of independent RNGs with small state. Two strategies have been explored: individual generators with small states which, however, suffer from small periods and state-sharing among several instances. An independent alternative are counter-based generators.

Next lecture

In lecture 4, we will have a look at some more advanced simulation of spin models, including cluster-update and multicanonical simulations.

Summary and outlook

This lecture

This lecture has given a survey of random number generators in a massively parallel environment. On GPUs, we need a massive number of independent RNGs with small state. Two strategies have been explored: individual generators with small states which, however, suffer from small periods and state-sharing among several instances. An independent alternative are counter-based generators.

Next lecture

In lecture 4, we will have a look at some more advanced simulation of spin models, including cluster-update and multicanonical simulations.

Reading

- M. Manssen, M. Weigel, and A. K. Hartmann, Eur. Phys. J. Special Topics **210**, 53 (2012) [arXiv:1204.6193].