
HH EU194

Design and Implementation of Vectorized Pseudorandom Number Generators

Master's Thesis Defense and Presentation

Markus Pawellek

May 25, 2020

Outline

Introduction

Introduction

What do we need random numbers for?

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- vectorize existing PRNGs

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- vectorize existing PRNGs
- create a software library and design a good API

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- vectorize existing PRNGs
- create a software library and design a good API
- apply library to physical problems

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- vectorize existing PRNGs
- create a software library and design a good API
- apply library to physical problems
- compare performance to other implementations

Pseudorandom Number Generators

True Randomness

What is a random sequence?

True Randomness

What is a random sequence?

- existing formal concepts not applicable to computer systems

True Randomness

What is a random sequence?

- existing formal concepts not applicable to computer systems
- nondeterministic, noncomputable, unpredictable

True Randomness

What is a random sequence?

- existing formal concepts not applicable to computer systems
- nondeterministic, noncomputable, unpredictable
- generated by hardware components based on chaotic processes

True Randomness

What is a random sequence?

- existing formal concepts not applicable to computer systems
- nondeterministic, noncomputable, unpredictable
- generated by hardware components based on chaotic processes

Disadvantages:

True Randomness

What is a random sequence?

- existing formal concepts not applicable to computer systems
- nondeterministic, noncomputable, unpredictable
- generated by hardware components based on chaotic processes

Disadvantages:

- Unreproducibility

True Randomness

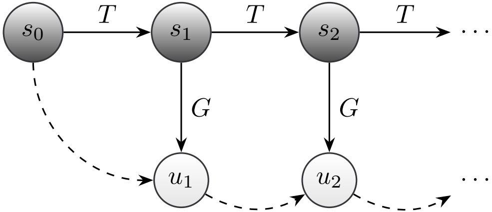
What is a random sequence?

- existing formal concepts not applicable to computer systems
- nondeterministic, noncomputable, unpredictable
- generated by hardware components based on chaotic processes

Disadvantages:

- Unreproducibility
- Speed Limitations

Pseudorandom Number Generator Definition



S . . . Set of States

T . . . Transition Function

U . . . Set of Possible Outputs

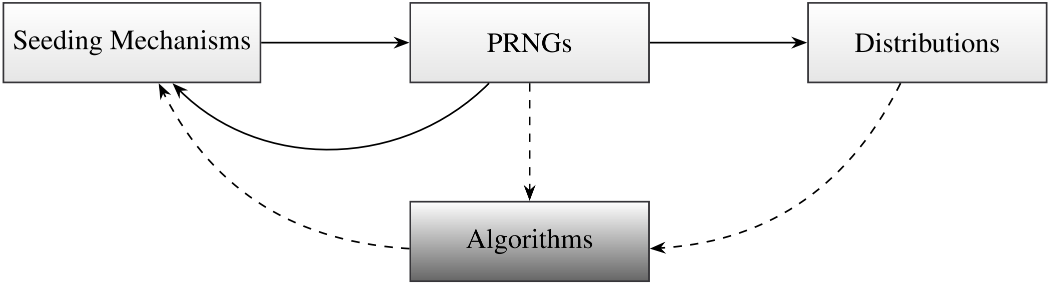
G . . . Generator Function

$$\mathcal{G} := (S, T, U, G), \qquad T\colon S \to S, \qquad G\colon S \to U$$

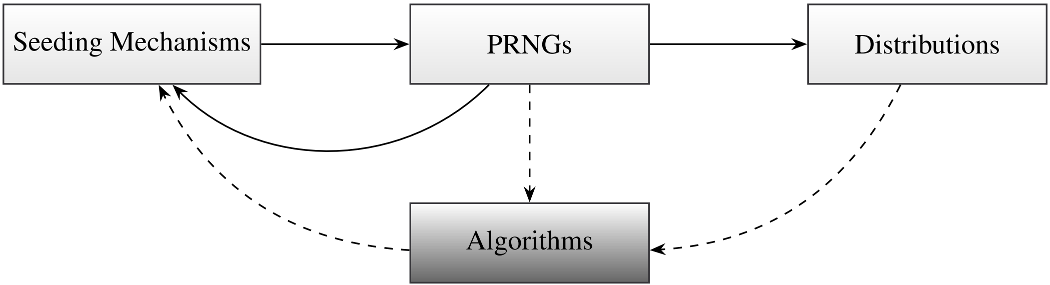
$$s_0 \in S, \qquad s_{n+1} := T(s_n), \qquad u_n := G(s_n)$$

Design of the Library

Design Components

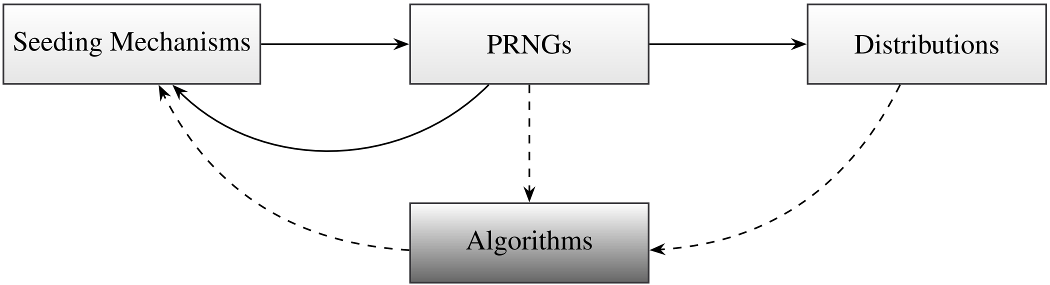


Design Components



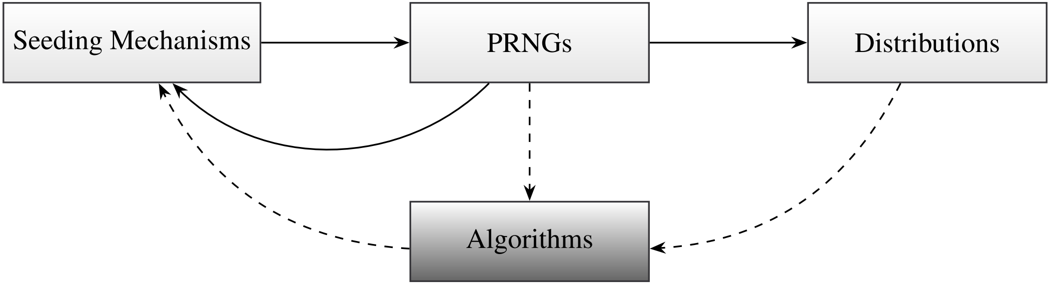
- pXart: header-only library written in C++

Design Components



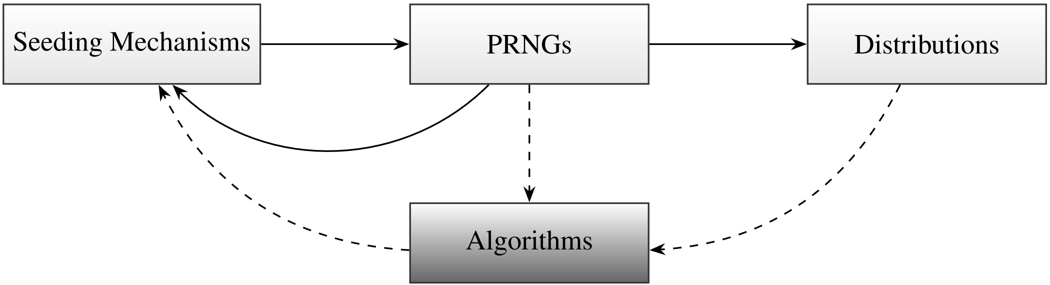
- pXart: header-only library written in C++
- support for CMake and build2

Design Components

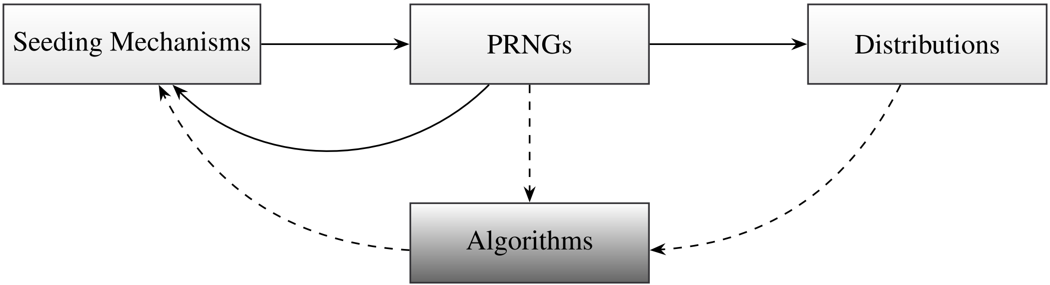


- pXart: header-only library written in C++
- support for CMake and build2
- providing online documentation

Design Components

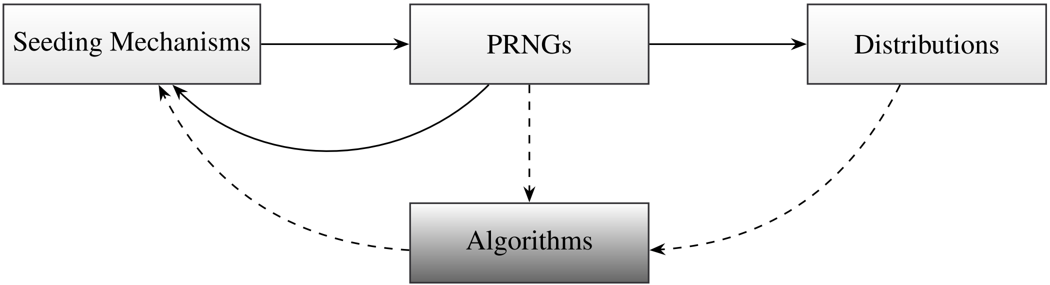


Design Components



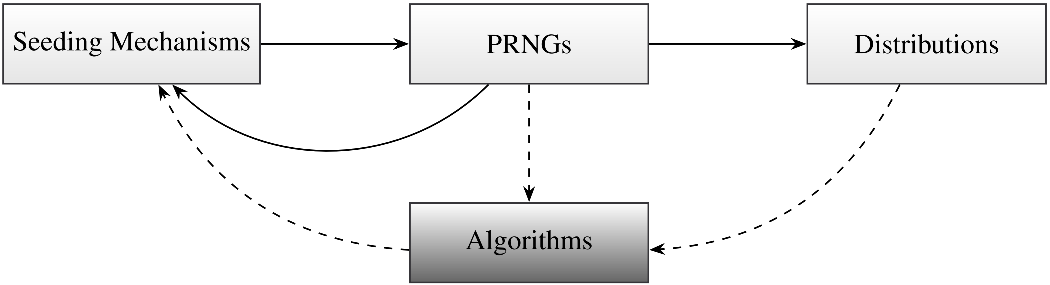
- PRNGs: MT19937, Xoroshiro128+, MSWS

Design Components



- PRNGs: MT19937, Xoroshiro128+, MSWS
- real and integer uniform distributions

Design Components



- PRNGs: MT19937, Xoroshiro128+, MSWS
- real and integer uniform distributions
- different seeding facilities

Vectorization and SIMD Architectures

SIMD Architecture

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0

$$+$$

y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

$$=$$

$x_7 + y_7$	$x_6 + y_6$	$x_5 + y_5$	$x_4 + y_4$	$x_3 + y_3$	$x_2 + y_2$	$x_1 + y_1$	$x_0 + y_0$

SIMD Architecture

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0

$$+$$

y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

$$=$$

$x_7 + y_7$	$x_6 + y_6$	$x_5 + y_5$	$x_4 + y_4$	$x_3 + y_3$	$x_2 + y_2$	$x_1 + y_1$	$x_0 + y_0$

- Single Instruction Multiple Data

SIMD Architecture

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0

$$+$$

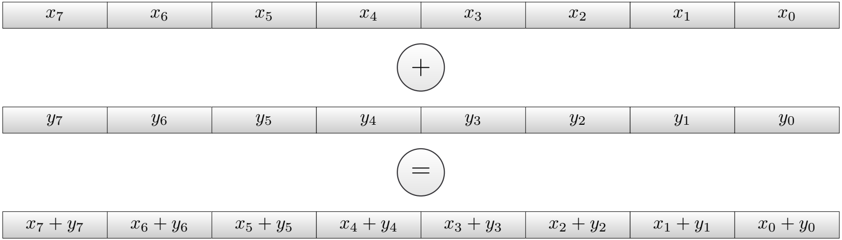
y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

$$=$$

$x_7 + y_7$	$x_6 + y_6$	$x_5 + y_5$	$x_4 + y_4$	$x_3 + y_3$	$x_2 + y_2$	$x_1 + y_1$	$x_0 + y_0$

- Single Instruction Multiple Data
- processor contains vector registers multiple elements

SIMD Architecture



- Single Instruction Multiple Data
- processor contains vector registers multiple elements
- processor operates on all values simultaneously

SIMD Implementations

Actual Hardware:

SIMD Implementations

Actual Hardware:

- SSE, AVX and AVX512 instruction sets by Intel

SIMD Implementations

Actual Hardware:

- SSE, AVX and AVX512 instruction sets by Intel
- Assembler Instructions | Automatic Vectorization | SIMD Intrinsics

SIMD Implementations

Actual Hardware:

- SSE, AVX and AVX512 instruction sets by Intel
- Assembler Instructions | Automatic Vectorization | SIMD Intrinsics

Why should we vectorize PRNGs manually?

SIMD Implementations

Actual Hardware:

- SSE, AVX and AVX512 instruction sets by Intel
- Assembler Instructions | Automatic Vectorization | SIMD Intrinsics

Why should we vectorize PRNGs manually?

- performance and speed

SIMD Implementations

Actual Hardware:

- SSE, AVX and AVX512 instruction sets by Intel
- Assembler Instructions | Automatic Vectorization | SIMD Intrinsics

Why should we vectorize PRNGs manually?

- performance and speed
- use full functionality of today's processors

SIMD Implementations

Actual Hardware:

- SSE, AVX and AVX512 instruction sets by Intel
- Assembler Instructions | Automatic Vectorization | SIMD Intrinsics

Why should we vectorize PRNGs manually?

- performance and speed
- use full functionality of today's processors
- no automatic vectorization possible

SIMD Implementations

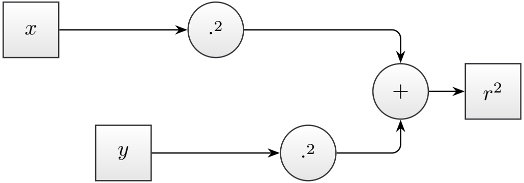
Actual Hardware:

- SSE, AVX and AVX512 instruction sets by Intel
- Assembler Instructions | Automatic Vectorization | SIMD Intrinsics

Why should we vectorize PRNGs manually?

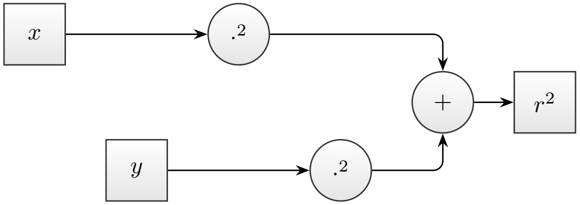
- performance and speed
- use full functionality of today's processors
- no automatic vectorization possible
- external vectorized code needs random numbers

SIMD Example

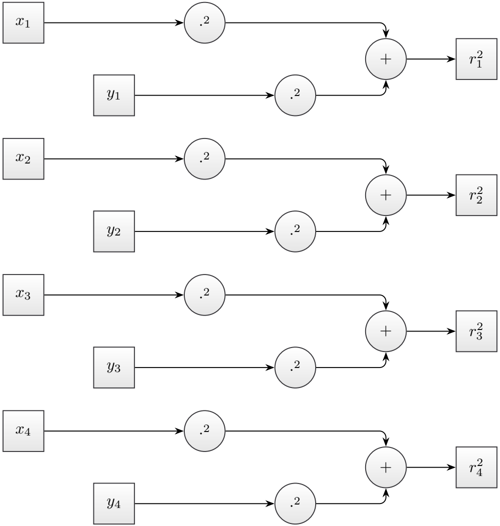


$$x, y \in \mathbb{R}, \qquad r^2 = x^2 + y^2$$

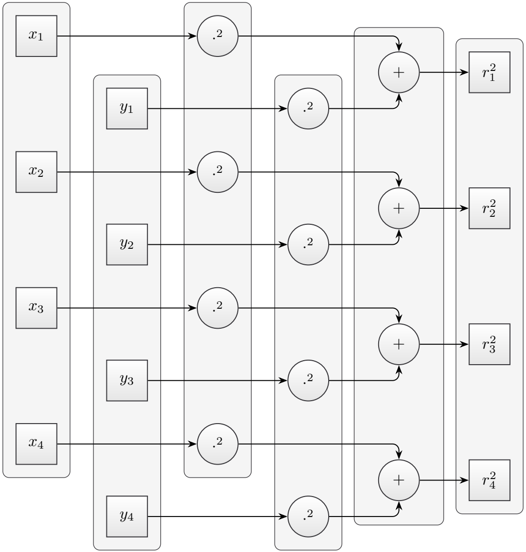
SIMD Example



SIMD Example

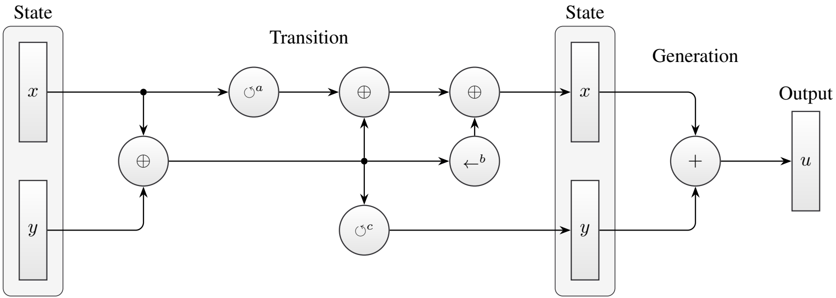


SIMD Example

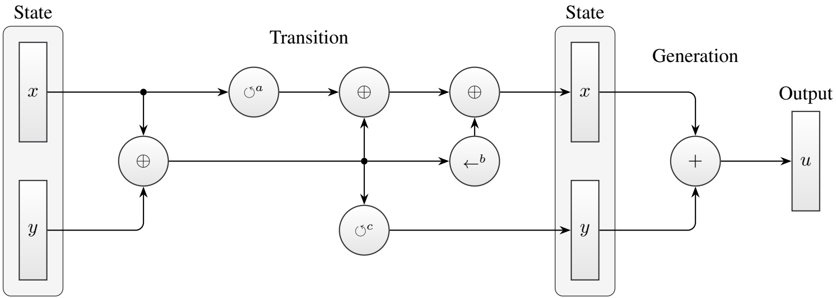


Xoroshiro128+

Xoroshiro128+ Scheme

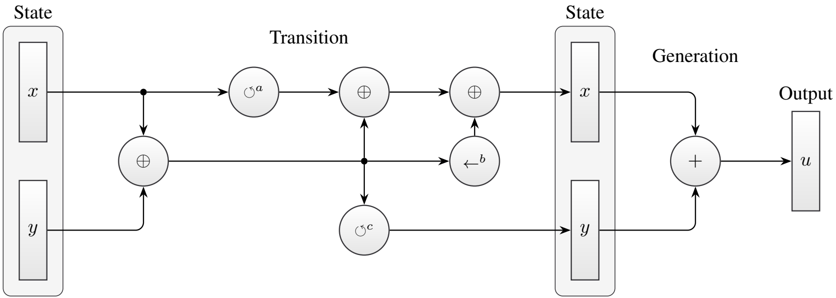


Xoroshiro128+ Scheme



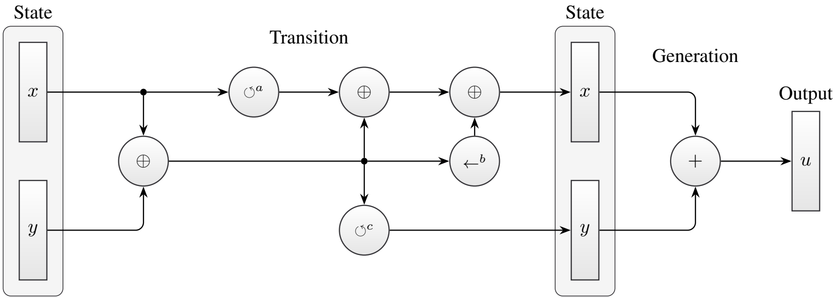
- scrambled linear PRNG

Xoroshiro128+ Scheme



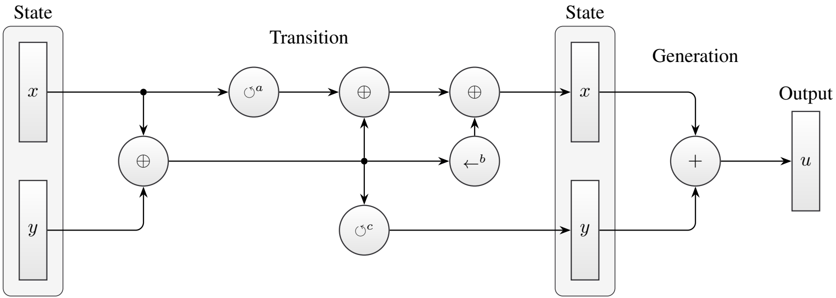
- scrambled linear PRNG
- 128-bit state, 64-bit output

Xoroshiro128+ Scheme



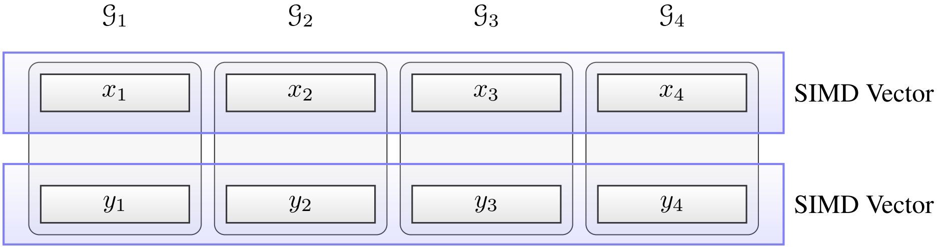
- scrambled linear PRNG
- 128-bit state, 64-bit output
- period: $2^{128} - 1$

Xoroshiro128+ Scheme

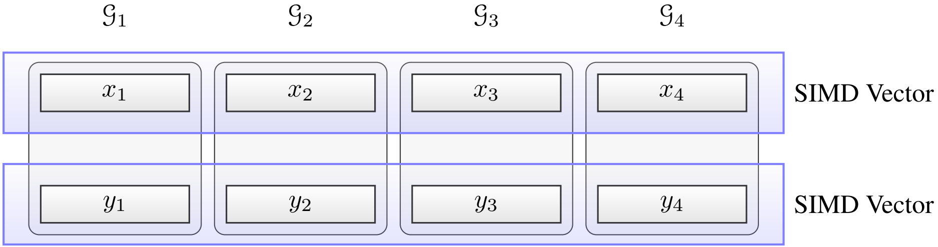


- scrambled linear PRNG
- 128-bit state, 64-bit output
- period: $2^{128} - 1$
- jump operations

Xoroshiro128+ SIMD Scheme

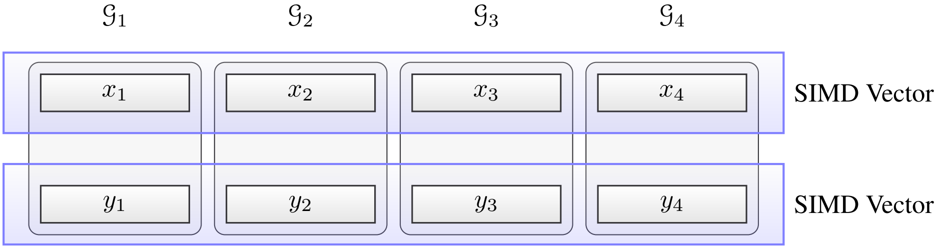


Xoroshiro128+ SIMD Scheme



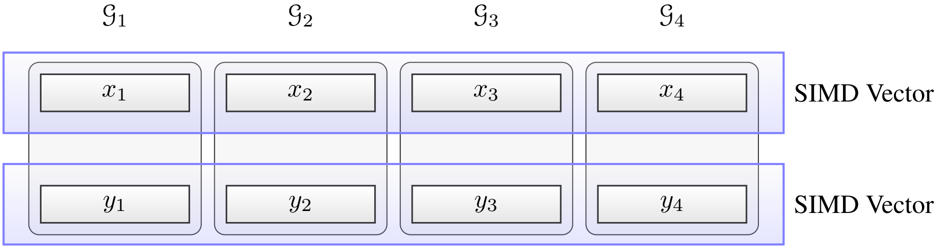
- several parallelization techniques for multiple streams

Xoroshiro128+ SIMD Scheme



- several parallelization techniques for multiple streams
- here: multiple instances of the same generator

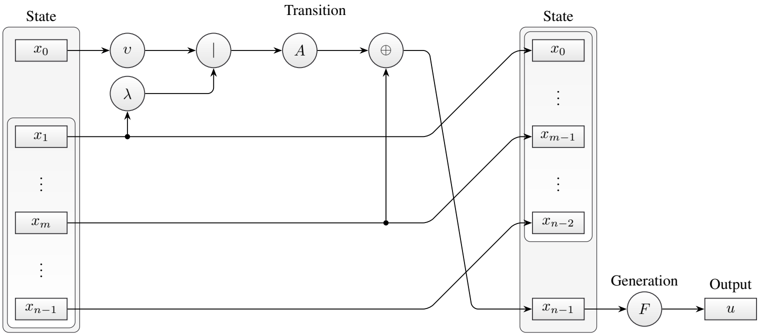
Xoroshiro128+ SIMD Scheme



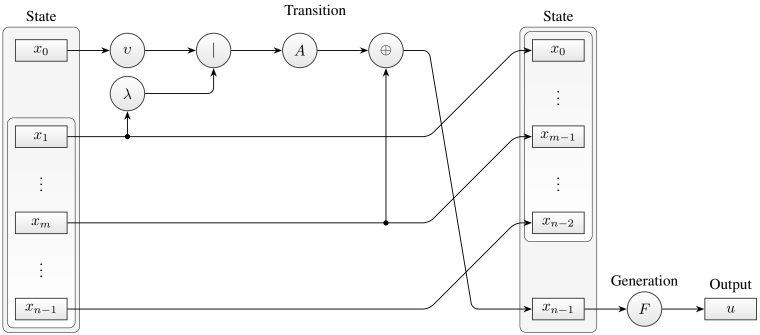
- several parallelization techniques for multiple streams
- here: multiple instances of the same generator
- seeding and parameter variations for multiple streams

Mersenne Twister MT19937

MT19937

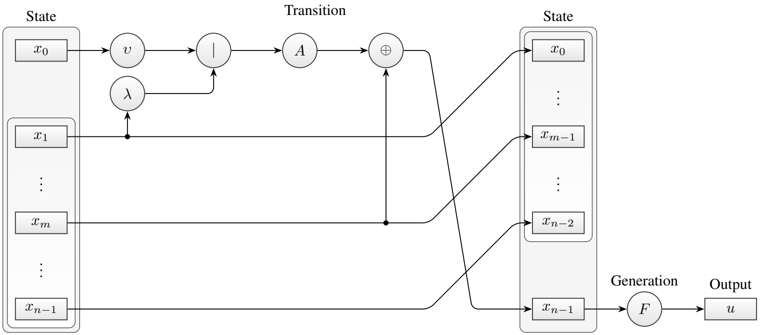

State
Transition
State
Generation
Output
x_0
x_1
x_m
x_{n-1}
υ
λ
$|$
A
$\oplus$
x_0
x_{m-1}
x_{n-2}
x_{n-1}
F
u

MT19937



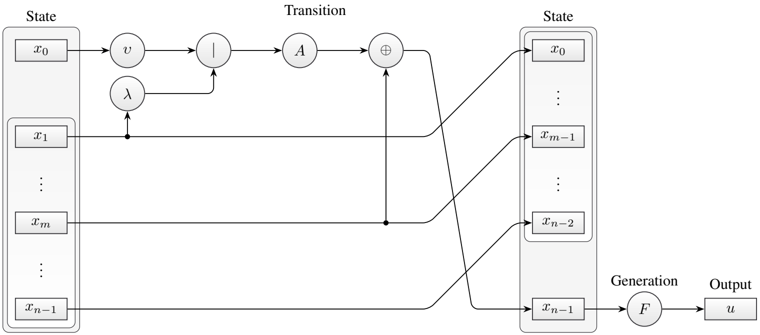
- de-facto standard

MT19937



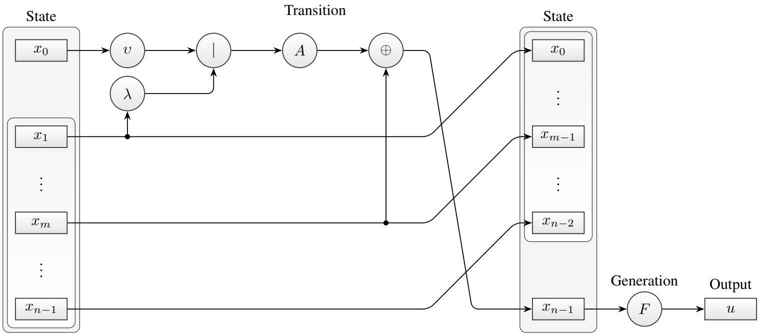
- de-facto standard
- linear PRNG

MT19937



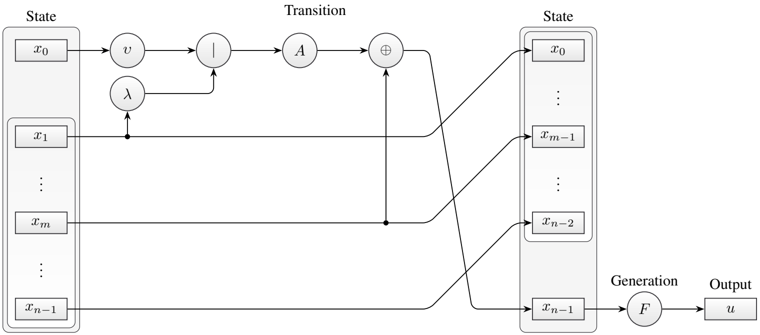
- de-facto standard
- linear PRNG
- 19937-bit state, 32-bit output

MT19937



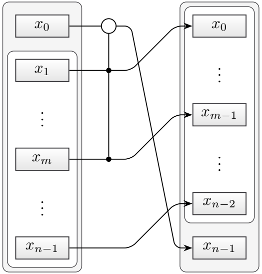
- de-facto standard
- linear PRNG
- 19937-bit state, 32-bit output
- period: $2^{19937} - 1$

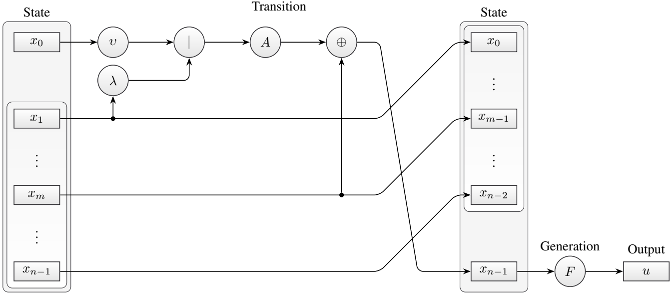
MT19937



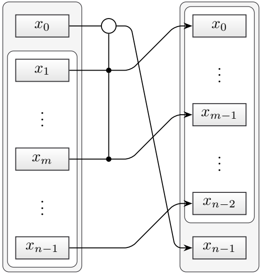
- de-facto standard
- linear PRNG
- 19937-bit state, 32-bit output
- period: $2^{19937} - 1$
- 623-dimensional equidistributed

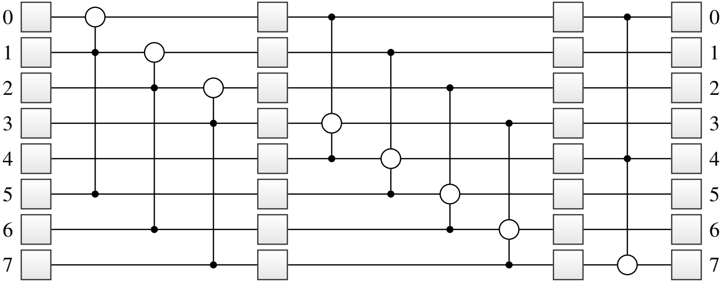
MT19937 Abbreviation



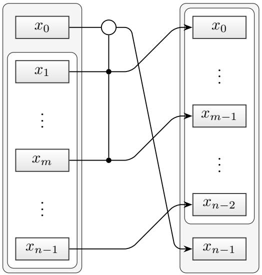


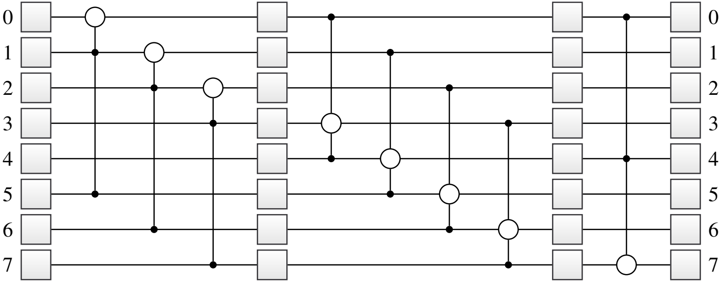
MT19937 Scalar Loop Scheme





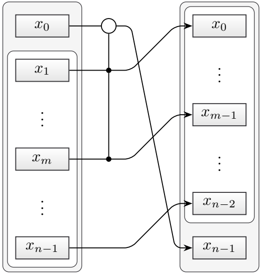
MT19937 Scalar Loop Scheme

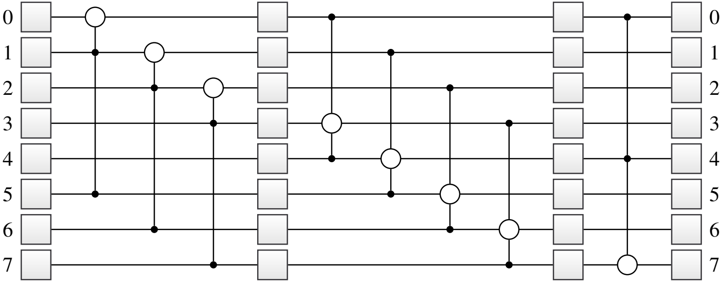




- moving all elements with one transition is inefficient

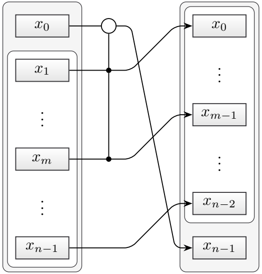
MT19937 Scalar Loop Scheme

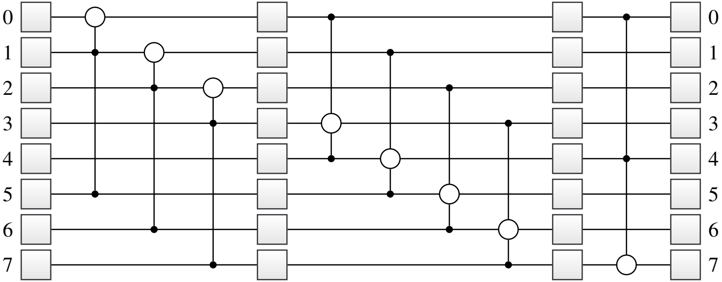




- moving all elements with one transition is inefficient
- instead do n transitions at once

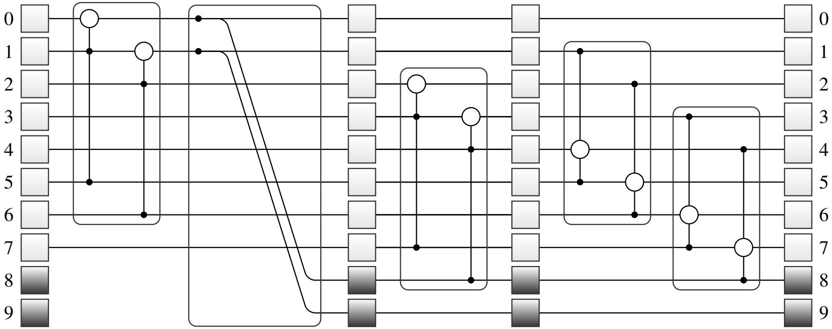
MT19937 Scalar Loop Scheme



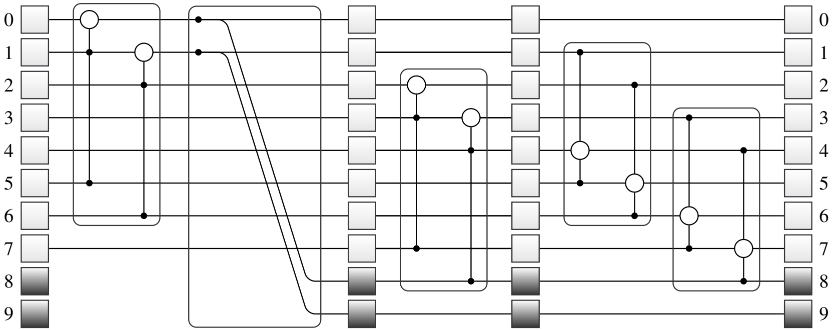


- moving all elements with one transition is inefficient
- instead do n transitions at once
- example with $n = 8$ and $m = 5$; reality with $n = 624$ and $m = 397$

MT19937 SIMD Loop Scheme

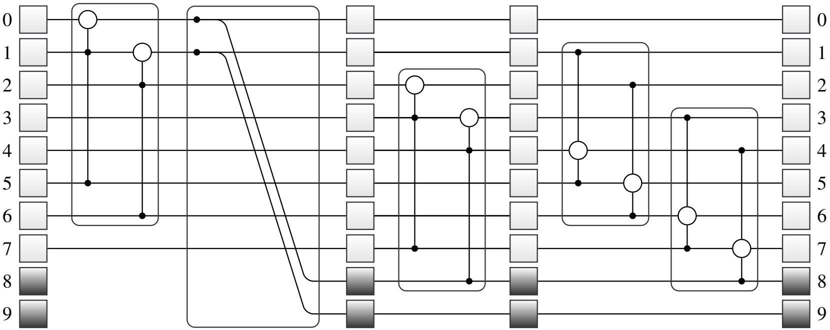


MT19937 SIMD Loop Scheme



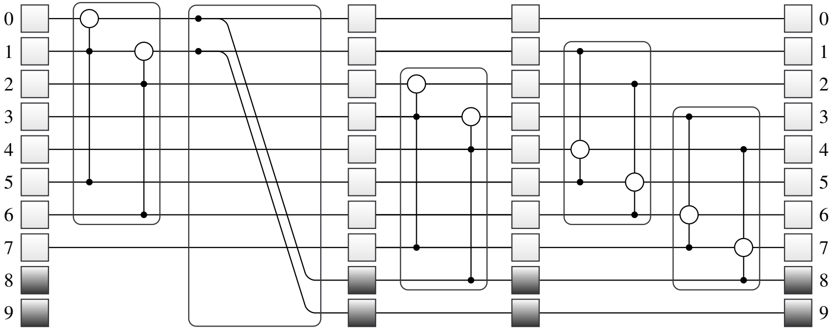
- example: two-element-vector; reality: up to eight-element-vector

MT19937 SIMD Loop Scheme



- example: two-element-vector; reality: up to eight-element-vector
- add vector-register-sized buffer at the end

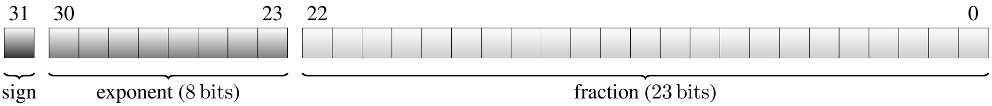
MT19937 SIMD Loop Scheme



- example: two-element-vector; reality: up to eight-element-vector
- add vector-register-sized buffer at the end
- copy generated head to the end and do the vectorized loop

Uniform Distribution Functions

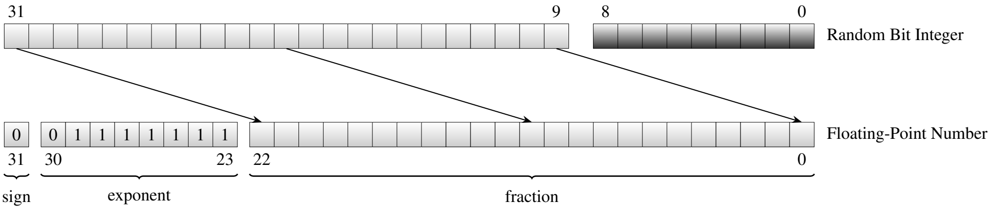
Real Uniform Distribution: Floating-Point Encoding



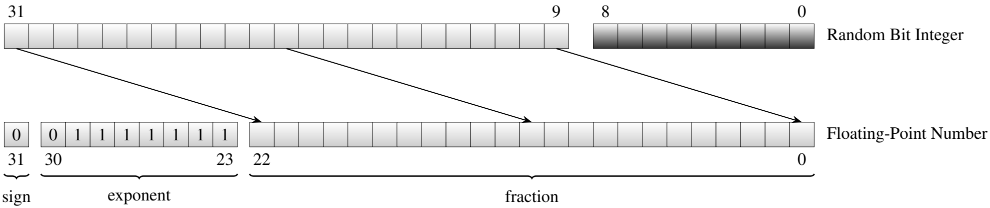
$$x = (-1)^s \cdot m \cdot 2^{e-o}$$

- IEEE 754
- we use only normalized numbers

Real Uniform Distribution

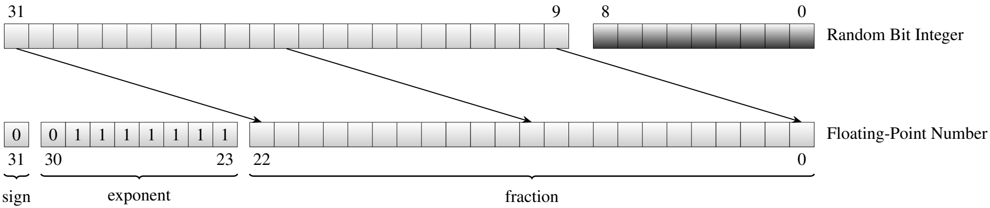


Real Uniform Distribution



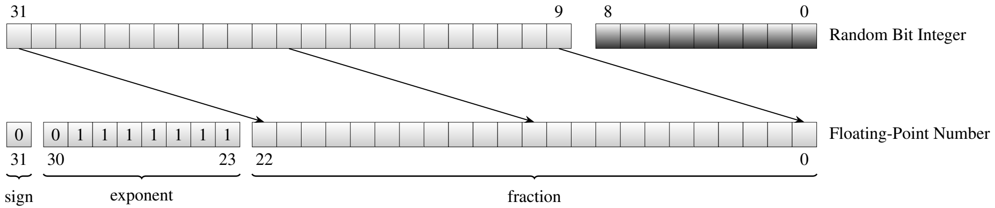
- get random integer

Real Uniform Distribution



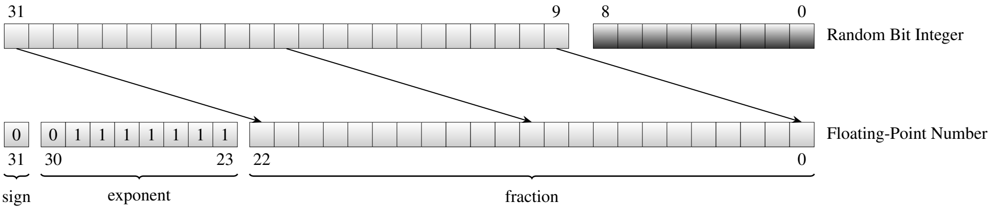
- get random integer
- shift bits with highest entropy into fraction part

Real Uniform Distribution



- get random integer
- shift bits with highest entropy into fraction part
- set sign and exponent to put floating-point value in range $[1, 2)$

Real Uniform Distribution



- get random integer
- shift bits with highest entropy into fraction part
- set sign and exponent to put floating-point value in range $[1, 2)$
- subtract one from result

Integer Uniform Distribution

Integer Uniform Distribution

- unbiased uniform integer algorithms should not be vectorized

Integer Uniform Distribution

- unbiased uniform integer algorithms should not be vectorized
- use simple multiplication-based approximation

$$x \in \mathbb{N}_0,\ x < 2^{32}, \qquad y = \left\lfloor \frac{(b-a)\cdot x}{2^{32}} \right\rfloor + a$$

Integer Uniform Distribution

- unbiased uniform integer algorithms should not be vectorized
- use simple multiplication-based approximation

$$x \in \mathbb{N}_0,\ x < 2^{32}, \qquad y = \left\lfloor \frac{(b-a)\cdot x}{2^{32}} \right\rfloor + a$$

- use 64-bit multiplication for 32-bit integers

Integer Uniform Distribution

- unbiased uniform integer algorithms should not be vectorized
- use simple multiplication-based approximation

$$x \in \mathbb{N}_0,\ x < 2^{32}, \qquad y = \left\lfloor \frac{(b-a)\cdot x}{2^{32}} \right\rfloor + a$$

- use 64-bit multiplication for 32-bit integers
- bias can be neglected for typical simulations

Evaluation and Results

Tests and Performance

Tests and Performance

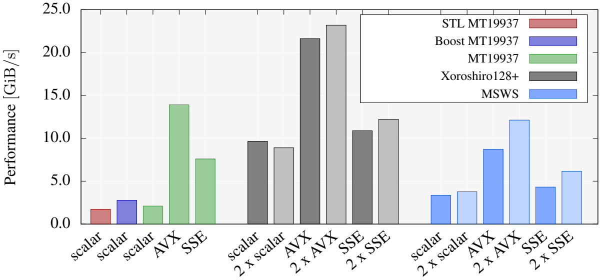
- Consistency and Correctness: Unit Tests, API Tests, Examples

Tests and Performance

- Consistency and Correctness: Unit Tests, API Tests, Examples
- Statistical Performance: TestU01, dieharder

Tests and Performance

- Consistency and Correctness: Unit Tests, API Tests, Examples
- Statistical Performance: TestU01, dieharder
- Performance: Filling a Cache, Monte Carlo π



Comparison to Intel MKL VSL and RNGAVXLIB

Table: MT19937 Monte Carlo π Benchmark for 10^8 Samples

RNGAVXLIB	Intel MKL VSL	Cached AVX	Pure AVX
$0.38\,\mathrm{s}$	$0.10\,\mathrm{s}$	$0.09\,\mathrm{s}$	$0.08\,\mathrm{s}$

Comparison to Intel MKL VSL and RNGAVXLIB

Table: MT19937 Monte Carlo π Benchmark for 10^8 Samples

RNGAVXLIB	Intel MKL VSL	Cached AVX	Pure AVX
$0.38\,\mathrm{s}$	$0.10\,\mathrm{s}$	$0.09\,\mathrm{s}$	$0.08\,\mathrm{s}$

- pXart is faster in Monte Carlo π benchmark

Comparison to Intel MKL VSL and RNGAVXLIB

Table: MT19937 Monte Carlo π Benchmark for 10^8 Samples

RNGAVXLIB	Intel MKL VSL	Cached AVX	Pure AVX
$0.38\,\mathrm{s}$	$0.10\,\mathrm{s}$	$0.09\,\mathrm{s}$	$0.08\,\mathrm{s}$

- pXart is faster in Monte Carlo π benchmark
- scalar interface of RNGAVXLIB reduces performance

Comparison to Intel MKL VSL and RNGAVXLIB

Table: MT19937 Monte Carlo π Benchmark for 10^8 Samples

RNGAVXLIB	Intel MKL VSL	Cached AVX	Pure AVX
$0.38\,\mathrm{s}$	$0.10\,\mathrm{s}$	$0.09\,\mathrm{s}$	$0.08\,\mathrm{s}$

- pXart is faster in Monte Carlo π benchmark
- scalar interface of RNGAVXLIB reduces performance
- Intel MKL VSL always fills vector of data

Comparison to Intel MKL VSL and RNGAVXLIB

Table: MT19937 Monte Carlo π Benchmark for 10^8 Samples

RNGAVXLIB	Intel MKL VSL	Cached AVX	Pure AVX
$0.38\,\mathrm{s}$	$0.10\,\mathrm{s}$	$0.09\,\mathrm{s}$	$0.08\,\mathrm{s}$

- pXart is faster in Monte Carlo π benchmark
- scalar interface of RNGAVXLIB reduces performance
- Intel MKL VSL always fills vector of data
- benchmarks are biased

Comparison to Intel MKL VSL and RNGAVXLIB

	pXart	RNGAVXLIB	Intel MKL VSL
Portable	✔	✘	✘
Good API	✔	✘	✘
Open Source	✔	✔	✘
Documentation	✔	✘	✔
Alternative Distributions	✘	✔	✔
AVX512 Support	✘	✘	✔
Header-Only	✔	✘	✘
Build System Support	✔	✘	✘

Conclusions and Future Work

Conclusions

Conclusions

- photon simulation and path tracing

Conclusions

- photon simulation and path tracing
- vectorized PRNGs speedup code even with caches

Conclusions

- photon simulation and path tracing
- vectorized PRNGs speedup code even with caches
- MT19937 or Xoroshiro128+?

Future Work

Future Work

- alternative distributions

Future Work

- alternative distributions
- seeding mechanisms for thread support

Future Work

- alternative distributions
- seeding mechanisms for thread support
- AVX512 support

Future Work

- alternative distributions
- seeding mechanisms for thread support
- AVX512 support
- latency optimizations

Future Work

- alternative distributions
- seeding mechanisms for thread support
- AVX512 support
- latency optimizations
- application to real-world problems

Thank you for Your Attention!

References

(1) Barash, L. Yu., Maria S. Guskova und Lev. N. Shchur: *Employing AVX Vectorization to Improve the Performance of Random Number Generators*. Programming and Computer Software, 43(3):145–160, 2017.

(2) Intel: *Intel Intrinsics Guide*, 2019. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`, besucht: 2019-11-21.

(3) Intel: *Intel® Math Kernel Library*, 2019. `https://software.intel.com/en-us/mkl`, besucht: 2019-11-30.

(4) Kneusel, Ronald T.: *Random Numbers and Computers*. Springer, 2018, ISBN 978-3-319-77697-2.

(5) Landau, David P. und Kurt Binder: *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press – University of Cambridge, fourth edition Auflage, 2014, ISBN 978-1-107-07402-6.

(6) L'Ecuyer, Pierre: *Uniform Random Number Generation*. Annals of Operations Research, 53:77–120, Dezember 1994.

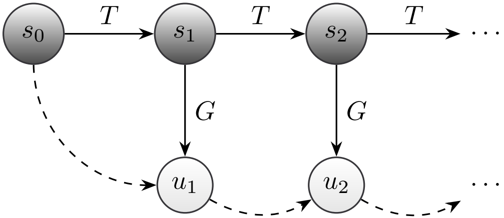
(7) Patterson, David A. und John L. Hennessy: *Computer Organization and Design*. Morgan Kaufmann – Elsevier, fifth edition Auflage, 2014, ISBN 978-0-12-407726-3.

(8) Pawellek, Markus: *Design and Implementation of Vectorized Pseudorandom Number Generators and their Application to Simulations of Photon Propagation*, 2019. `https://github.com/lyrahgames/pxart/blob/master/docs/thesis/main.pdf`, besucht: 2020-05-25.

(9) Pawellek, Markus: *pxart*, 2019. `https://github.com/lyrahgames/pxart`, besucht: 2019-12-11.

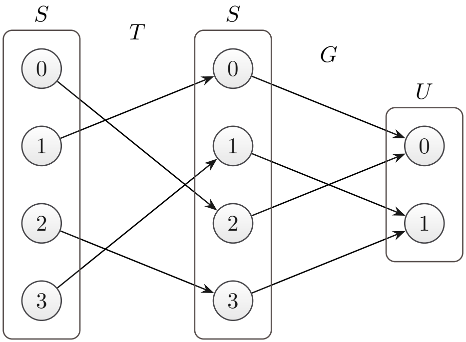
(10) Pharr, Matt, Wenzel Jakob und Greg Humphreys: *Physically Based Rendering*. Morgan Kaufmann – Elsevier, third edition Auflage, 2016, ISBN 978-0-12-800645-0.

Appendix: Pseudorandom Number Generator Concept



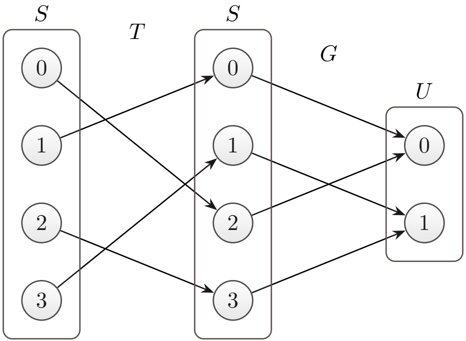
$$s_0 \sim \mathcal{U}_S, \qquad u_1 \leftarrow \mathcal{G}(), \qquad u_2 \leftarrow \mathcal{G}(), \qquad u_3 \leftarrow \mathcal{G}(), \qquad \ldots$$

Appendix: Pseudorandom Number Generator Example



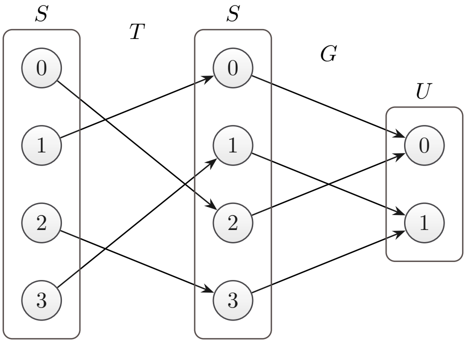
$$s_0 := 0, \qquad (s_n) = \overline{2310}, \qquad (u_n) = \overline{0110}$$

Appendix: Pseudorandom Number Generator Example



- construction of “good” PRNG is difficult

Appendix: Pseudorandom Number Generator Example

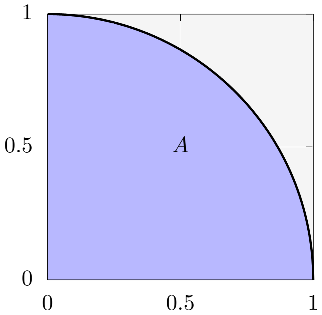


- construction of "good" PRNG is difficult
- pseudorandom number sequences will be periodic

Appendix: pXart Usage in C++

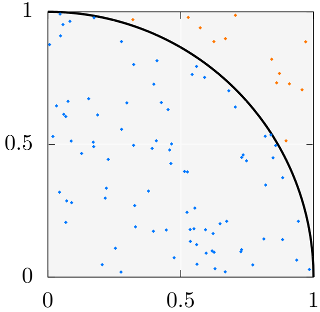
```
#include <pxart/pxart.hpp>
//
std::random_device rd{};
//
pxart::mt19937 rng1{};
pxart::mt19937 rng1{rd};
pxart::mt19937 rng1{pxart::mt19937::default_seeder{rd()}};
//
pxart::xrsr128p rng2{rng1};
//
const auto x = pxart::uniform<float>(rng1);
//
const auto y = pxart::uniform(rng2, -1.0f, 1.0f);
```

Appendix: Computation of π



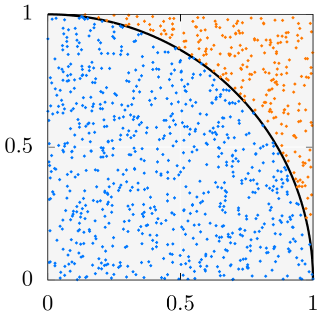
$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N}$$

Appendix: Computation of π



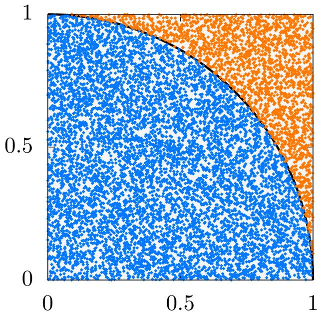
$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N} = \frac{4 \cdot 87}{100} = 3.48$$

Appendix: Computation of π



$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N} = \frac{4 \cdot 765}{1000} = 3.06$$

Appendix: Computation of π



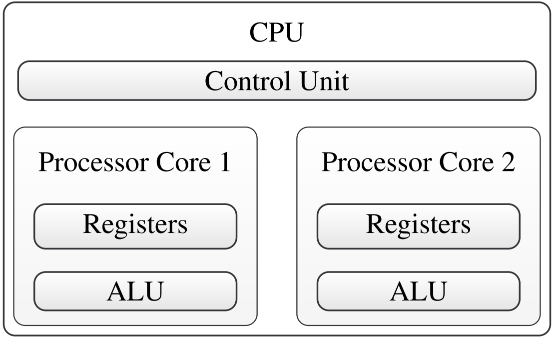
$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N} = \frac{4 \cdot 7856}{10000} = 3.1424$$

Appendix: Example Usage

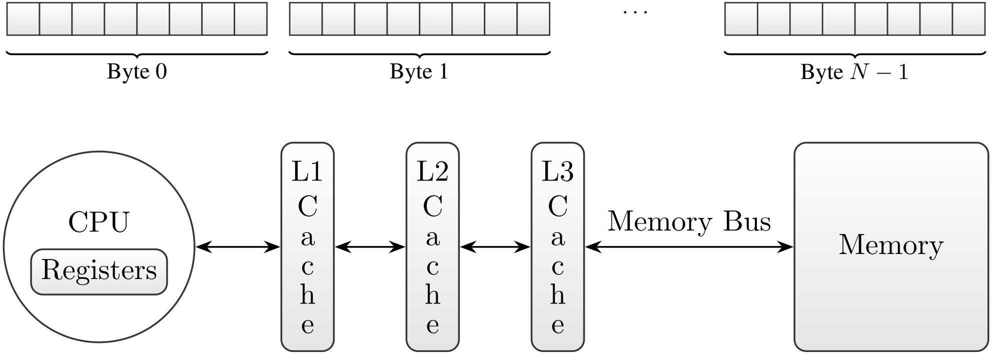
```cpp
// ...
#include <pxart/pxart.hpp>
// ...
pxart::mt19937 rng{};
const int samples = 100000000;
int pi = 0;
for (auto i = samples; i > 0; --i) {
  const auto x = pxart::uniform<float>(rng);
  const auto y = pxart::uniform<float>(rng);
  pi += (x * x + y * y <= 1);
}
pi = 4.0f * pi / samples;

// ...
```

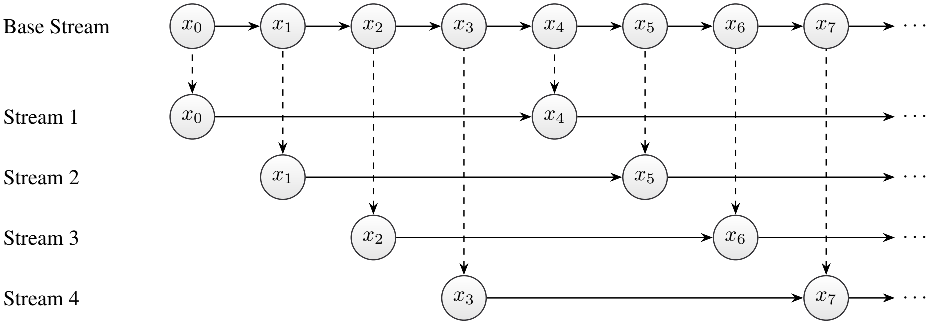
Appendix: Processor



Appendix: Memory Hierarchy

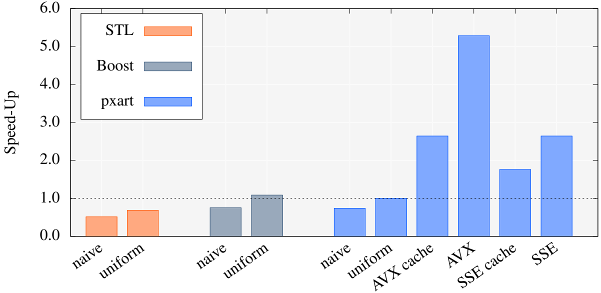


Appendix: MT19937 SIMD Leap Frogging



- vectorized generator will give same output as scalar one, only faster

Appendix: MT19937 Speed-Up Monte Carlo π



Appendix: Xoroshiro128+ Speed-Up Monte Carlo π