
HH EU 194

Design and Implementation of Vectorized Pseudorandom Number Generators

Master's Thesis Defense and Presentation

Markus Pawellek

May 24, 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:

- implement RNGs and according algorithms

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- implement RNGs and according algorithms
- vectorize those implementations

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- implement RNGs and according algorithms
- vectorize those implementations
- create a software library with powerful API

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- implement RNGs and according algorithms
- vectorize those implementations
- create a software library with powerful API
- compare performance to others implementations

Introduction

What do we need random numbers for?

- Physical Simulations, based on Monte-Carlo Methods

Goals:

- implement RNGs and according algorithms
- vectorize those implementations
- create a software library with powerful API
- compare performance to others implementations
- apply library to physical problems

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

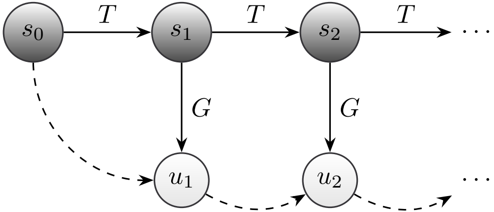
What is a random sequence?

- 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



$S \dots$ Set of States

$T \dots$ Transition Function

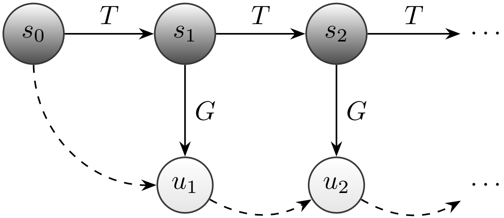
$U \dots$ Set of Possible Outputs

$G \dots$ Generator Function

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

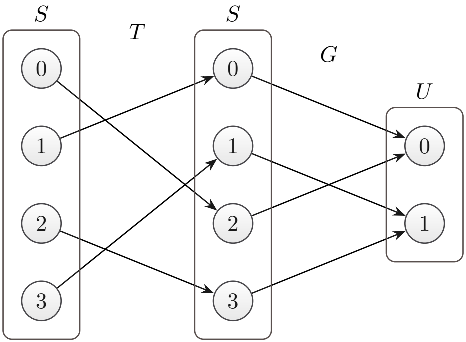
$$s_0 \in S, \qquad s_{n+1} \coloneqq T(s_n), \qquad u_n \coloneqq G(s_n)$$

Pseudorandom Number Generator Concept



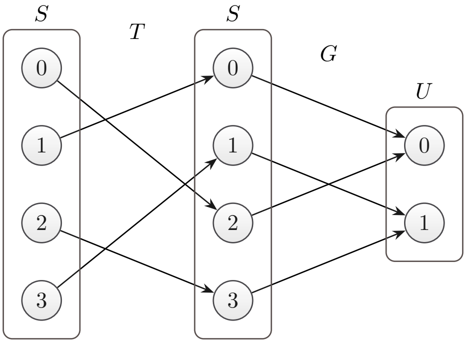
$$s_0 \sim \mathcal{U}_S, \qquad u_1 \leftarrow \mathcal{G}(), \qquad u_2 \leftarrow \mathcal{G}(), \qquad u_3 \leftarrow \mathcal{G}(), \qquad \ldots$$

Pseudorandom Number Generator Example



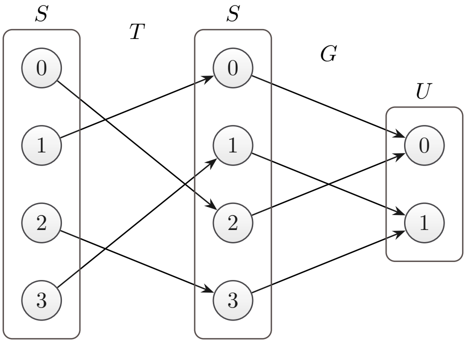
$$s_0 := 0, \qquad (s_n) = \overline{2310}, \qquad (u_n) = \overline{0110}$$

Pseudorandom Number Generator Example



- construction of “good” PRNG is difficult

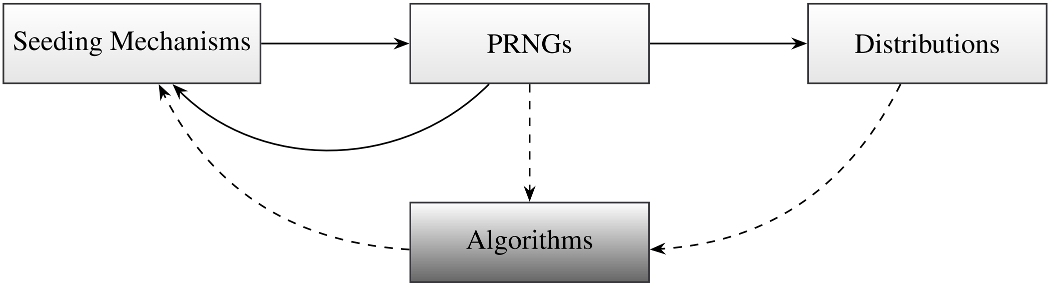
Pseudorandom Number Generator Example



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

Design of the Library

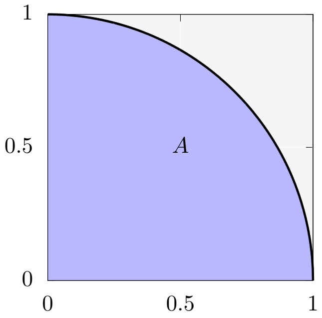
Design Components



Usage in C++

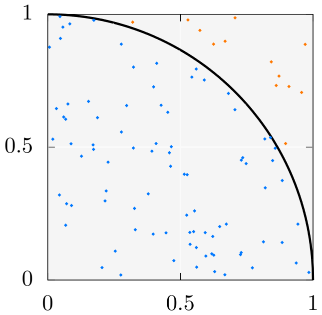
```cpp
#include <pxart/pxart.hpp>
//
std::random_device rd{};
//
pxart::mt19937 rng1{};
pxart::mt19937 rng1{rd};
pxart::mt19937 rng1{pxart::mt19937::default_seeder{rd()}};
//
pxart::xrsr128p rng2{rng1};
//
const auto x = pxart::uniform<float>(rng1);
//
const auto y = pxart::uniform(rng2, -1.0f, 1.0f);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

Computation of π



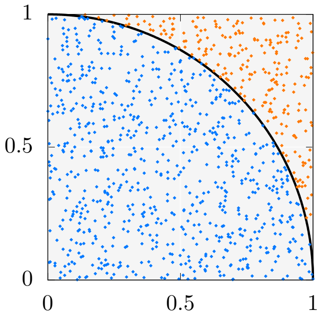
$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N}$$

Computation of π



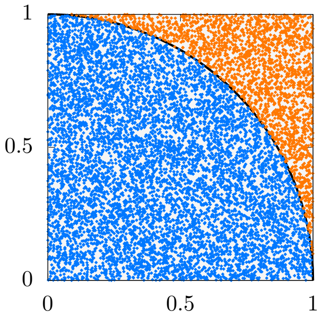
$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N} = \frac{4 \cdot 87}{100} = 3.48$$

Computation of π



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

Computation of π



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

Example Usage

```
// ...
#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;

// ...
```

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

$$+$$

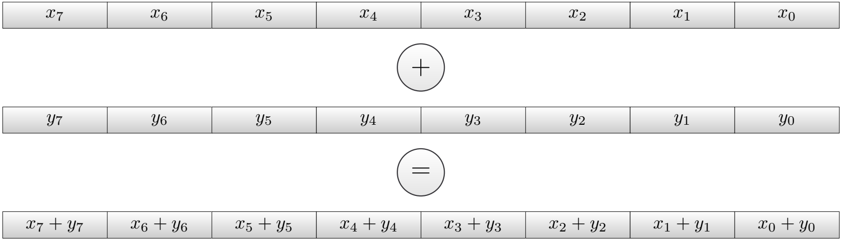
y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

$$=$$

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

- exploits data-level parallelism on a low level

SIMD Architecture



- exploits data-level parallelism on a low level
- Intel CPUs use fixed-length vector registers

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

$$=$$

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

- exploits data-level parallelism on a low level
- Intel CPUs use fixed-length vector registers
- vector operations are performed on all values at once

SIMD Features in C++

SIMD Features in C++

- SSE, AVX and AVX512
  instruction set features

SIMD Features in C++

- SSE, AVX and AVX512 instruction set features
- Assembler Instructions vs. Automatic Vectorization vs. SIMD Intrinsics

SIMD Features in C++

- SSE, AVX and AVX512 instruction set features
- Assembler Instructions vs. Automatic Vectorization vs. SIMD Intrinsics

```
// 128-bit registers
__m128 a;
__m128d b;
__m128i c;

c = _mm_add_ps(a, b);

// 256-bit registers
__m256 a;
__m256i b;
__m256d c;

c = _mm256_add_ps(a, b);
```

SIMD Architecture

Why should we vectorize PRNGs manually?

SIMD Architecture

Why should we vectorize PRNGs manually?

- exploit full functionality of today's processors

SIMD Architecture

Why should we vectorize PRNGs manually?

- exploit full functionality of today's processors
- no automatic vectorization possible

SIMD Architecture

Why should we vectorize PRNGs manually?

- exploit full functionality of today's processors
- no automatic vectorization possible
- other vectorized code needs vectorized random numbers

SIMD Architecture

Why should we vectorize PRNGs manually?

- exploit full functionality of today's processors
- no automatic vectorization possible
- other vectorized code needs vectorized random numbers
- faster generation of numbers

SIMD Architecture

Why should we vectorize PRNGs manually?

- exploit full functionality of today's processors
- no automatic vectorization possible
- other vectorized code needs vectorized random numbers
- faster generation of numbers
- PRNGs are low-level, SIMD is low-level

SIMD Architecture

What are conditions for good vectorization?

SIMD Architecture

What are conditions for good vectorization?

- nearly no data dependencies

SIMD Architecture

What are conditions for good vectorization?

- nearly no data dependencies
- same processing pipeline

SIMD Architecture

What are conditions for good vectorization?

- nearly no data dependencies
- same processing pipeline
- branchless execution

SIMD Architecture

What are conditions for good vectorization?

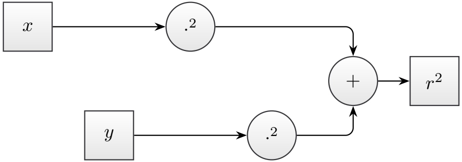
- nearly no data dependencies
- same processing pipeline
- branchless execution
- CPU-bound algorithms

SIMD Example

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

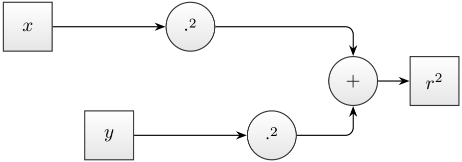
SIMD Example

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



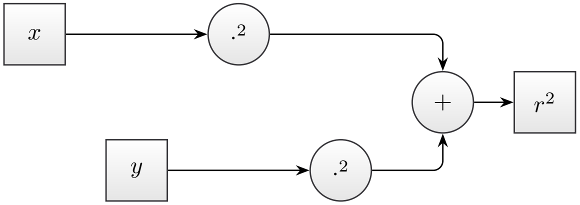
SIMD Example

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

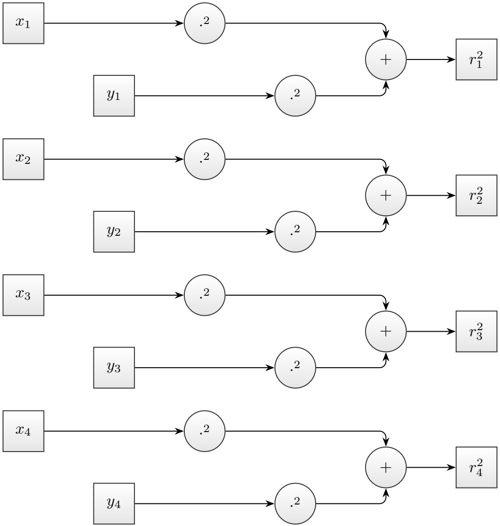



```
double x = pxart::uniform<double>(rng);
double y = pxart::uniform<double>(rng);

double x2 = x * x;
double y2 = y * y;
double r2 = x2 + y2;
```

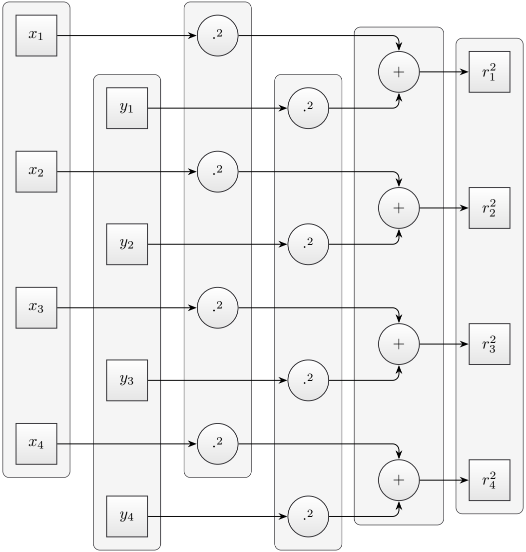
SIMD Example



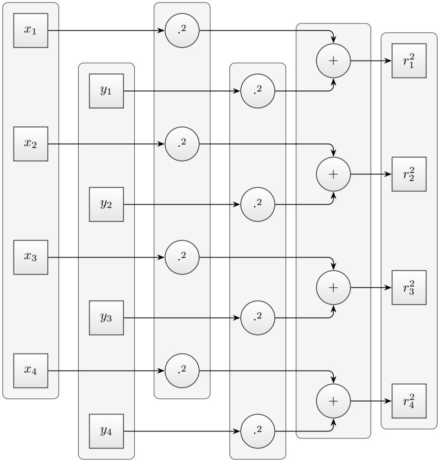
SIMD Example



SIMD Example



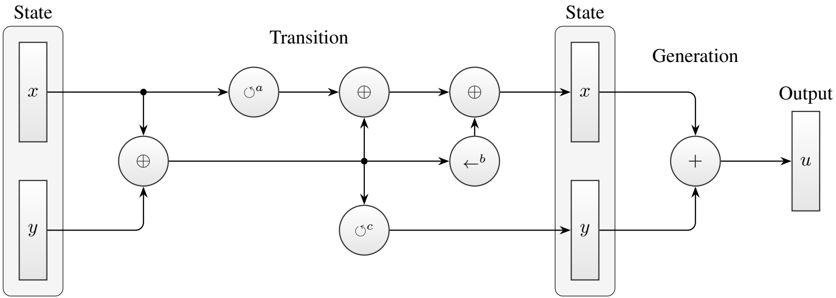
SIMD Example



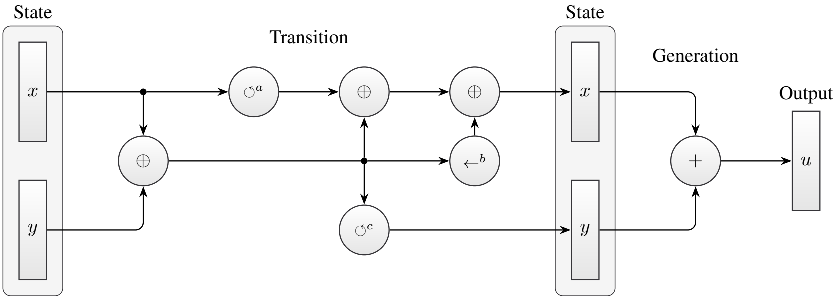

```
__m256d x = pxart::uniform<double>(vrng);
__m256d y = pxart::uniform<double>(vrng);

__m256d x2 = _mm256_mul_pd(x, x);
__m256d y2 = _mm256_mul_pd(y, y);
__m256d r2 = _mm256_add_pd(x2, y2);
```

Implementation of the Xoroshiro128+

Xoroshiro128+ Scheme

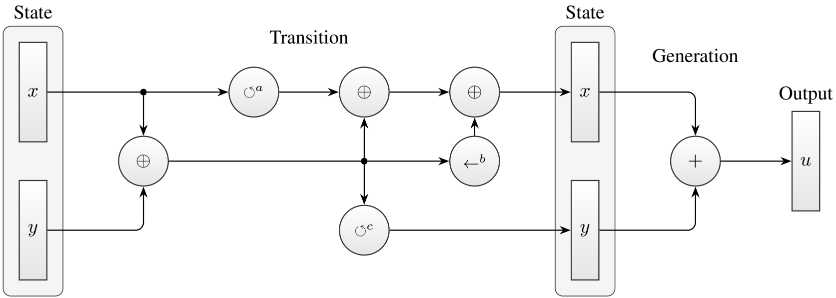


Xoroshiro128+ Scheme



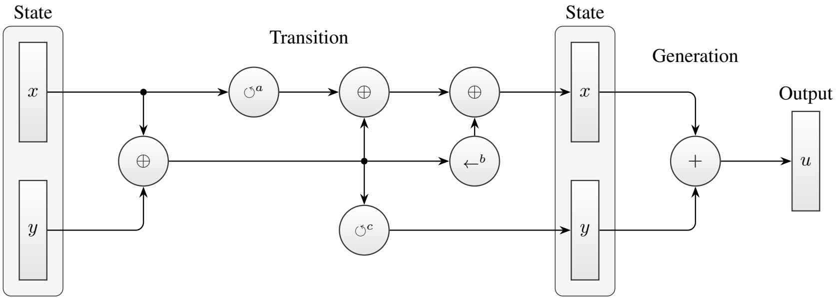
- scrambled linear PRNG

Xoroshiro128+ Scheme



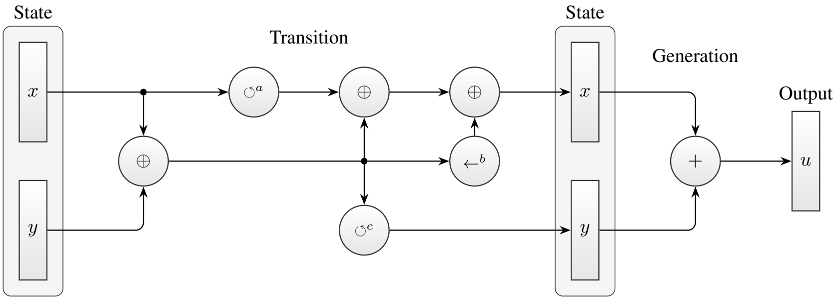
- scrambled linear PRNG
- 128-bit state, 64-bit output

Xoroshiro128+ Scheme



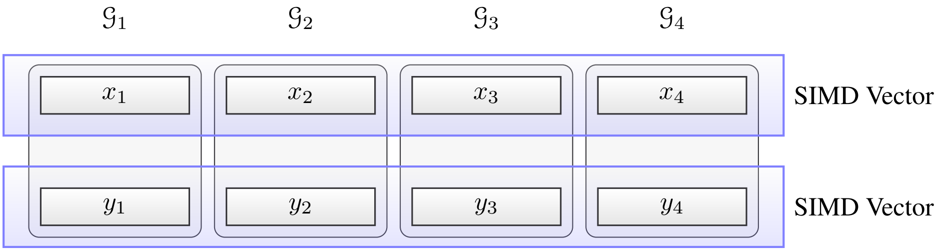
- scrambled linear PRNG
- 128-bit state, 64-bit output
- period: $2^{128} - 1$

Xoroshiro128+ Scheme

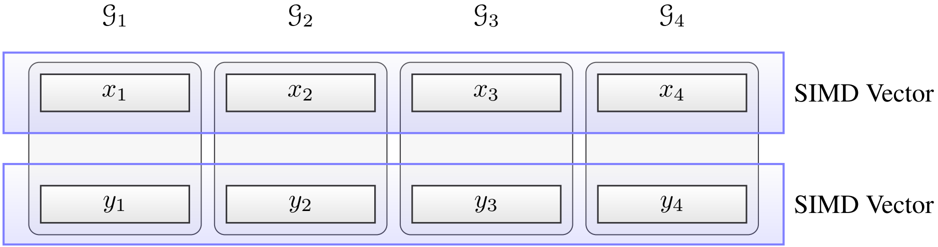


- scrambled linear PRNG
- 128-bit state, 64-bit output
- period: $2^{128} - 1$
- jump operations

Xoroshiro128+ SIMD Scheme

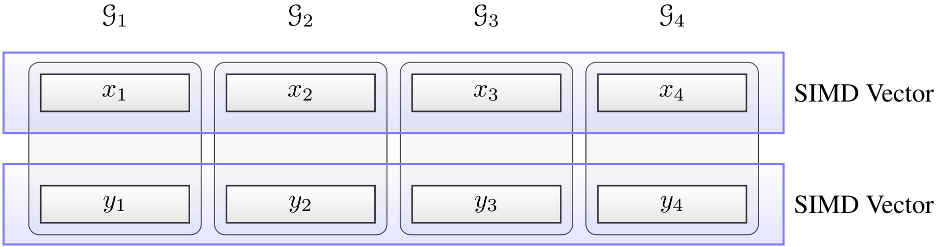


Xoroshiro128+ SIMD Scheme



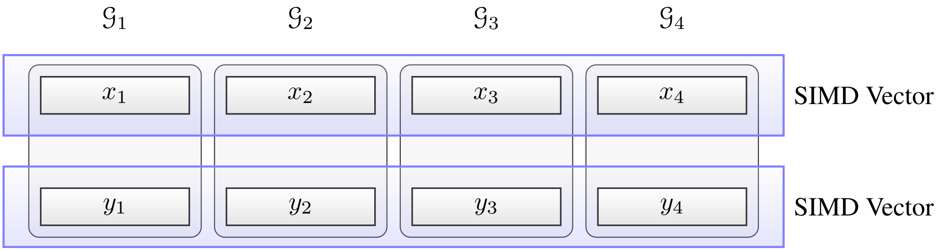
- several parallelization techniques for multiple streams

Xoroshiro128+ SIMD Scheme



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

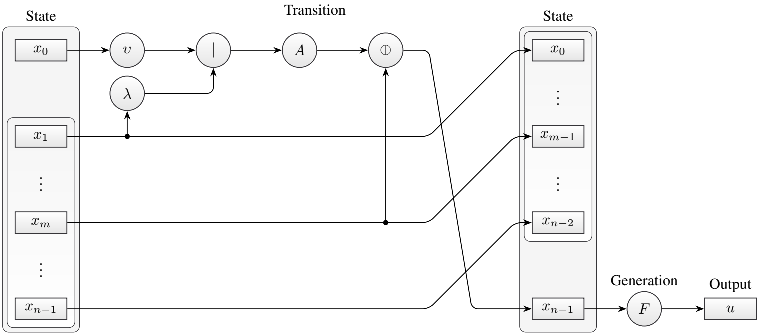
Xoroshiro128+ SIMD Scheme



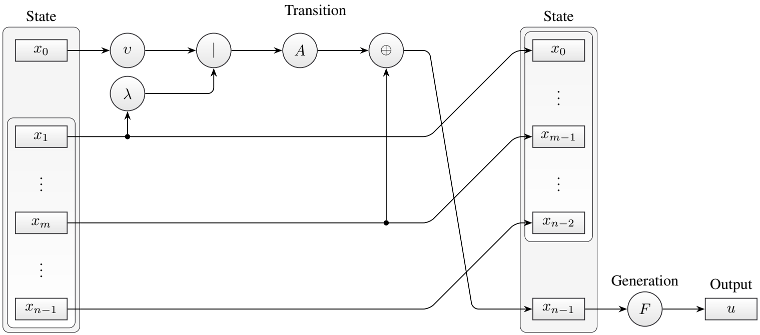
- several parallelization techniques for multiple streams
- here: multiple instances of the same generator
- seeding and parameter variations for multiple streams

Implementation of the MT19937

MT19937

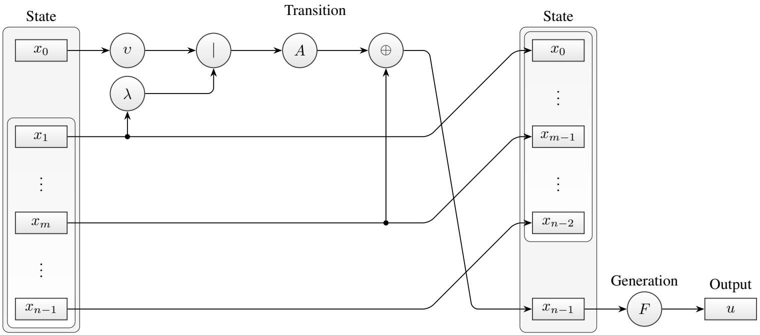

State
Transition
State
x_0
v
$|$
A
$\oplus$
λ
x_1
$\vdots$
x_m
$\vdots$
x_{n-1}
x_0
$\vdots$
x_{m-1}
$\vdots$
x_{n-2}
x_{n-1}
Generation
F
Output
u

MT19937



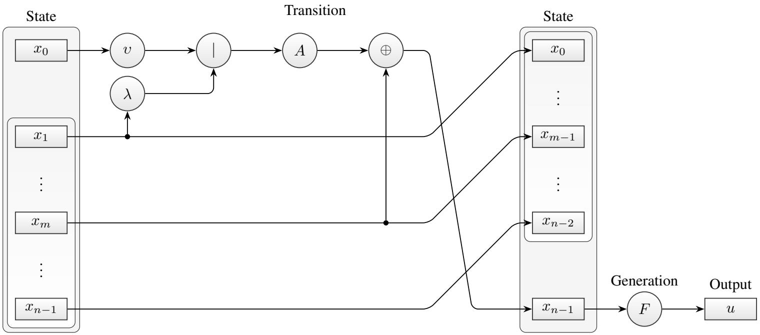
- de-facto standard

MT19937



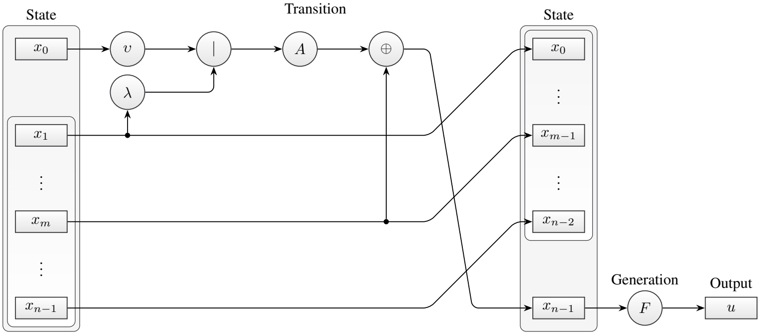
- de-facto standard
- linear PRNG

MT19937



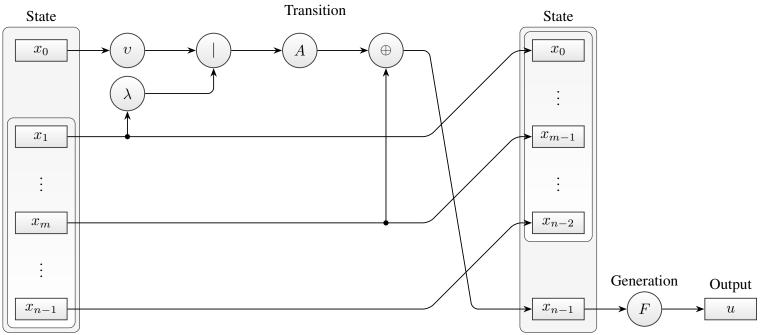
- de-facto standard
- linear PRNG
- 19937-bit state, 32-bit output

MT19937



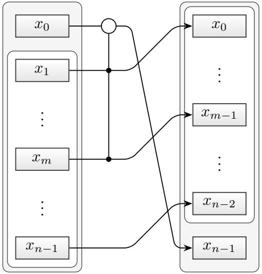
- de-facto standard
- linear PRNG
- 19937-bit state, 32-bit output
- period: $2^{19937} - 1$

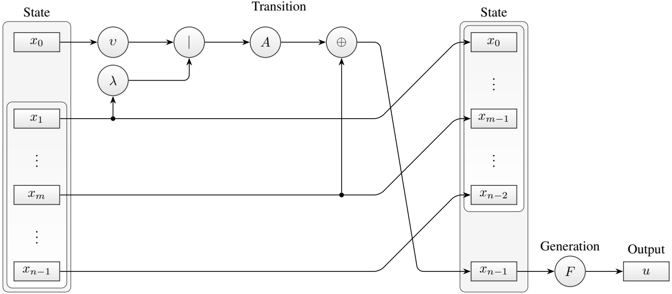
MT19937



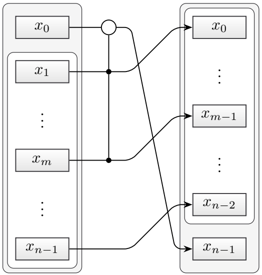
- de-facto standard
- linear PRNG
- 19937-bit state, 32-bit output
- period: $2^{19937} - 1$
- 623-dimensional equidistributed

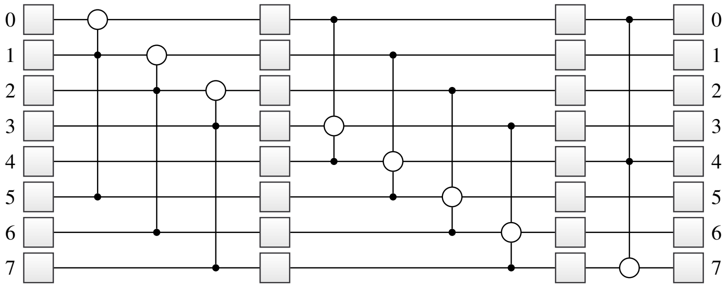
MT19937 Abbreviation



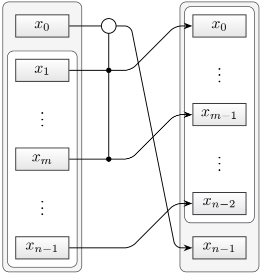


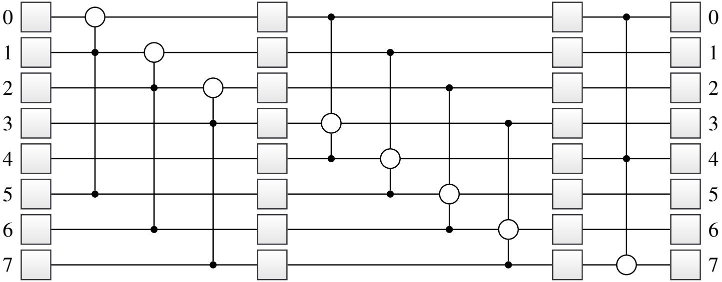
MT19937 Scalar Loop Scheme





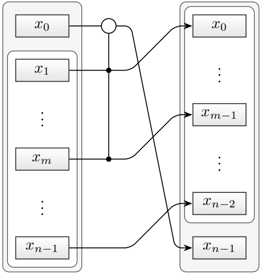
MT19937 Scalar Loop Scheme

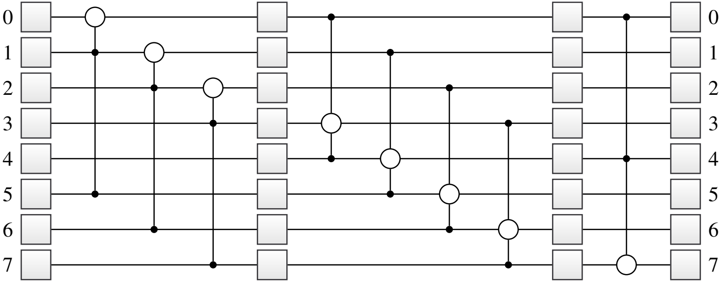




- moving all elements with one transition is inefficient

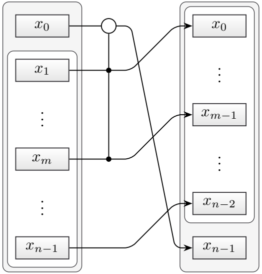
MT19937 Scalar Loop Scheme

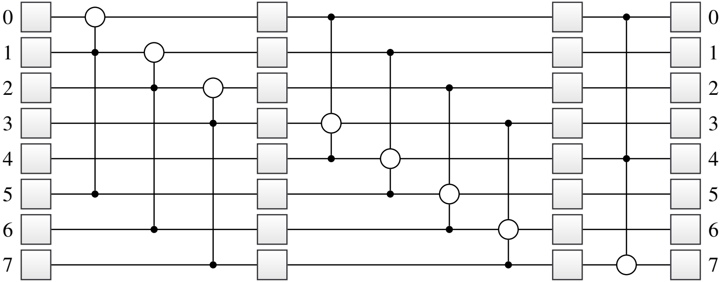




- moving all elements with one transition is inefficient
- instead do n transitions at once

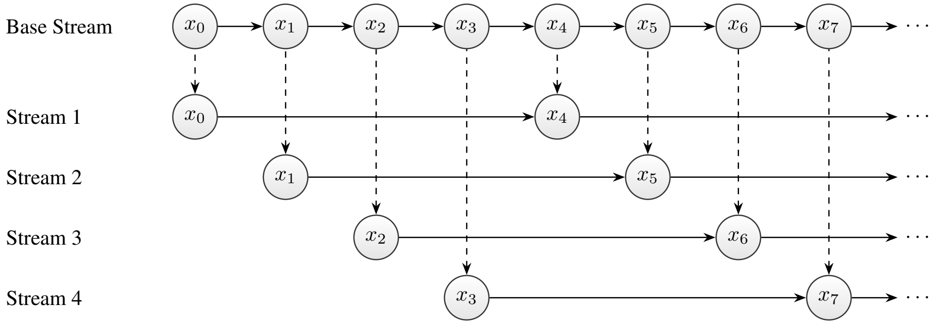
MT19937 Scalar Loop Scheme





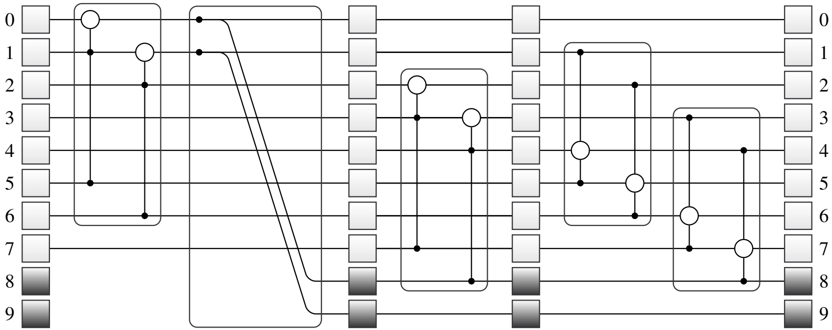
- moving all elements with one transition is inefficient
- instead do n transitions at once
- example with $n = 8$ and $m = 5$; reality with $n = 624$ and $m = 397$

MT19937 SIMD Leap Frogging

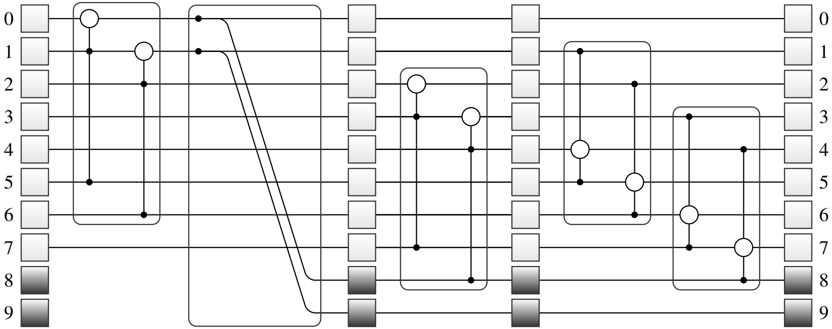


- vectorized generator will give same output as scalar one, only faster

MT19937 SIMD Loop Scheme

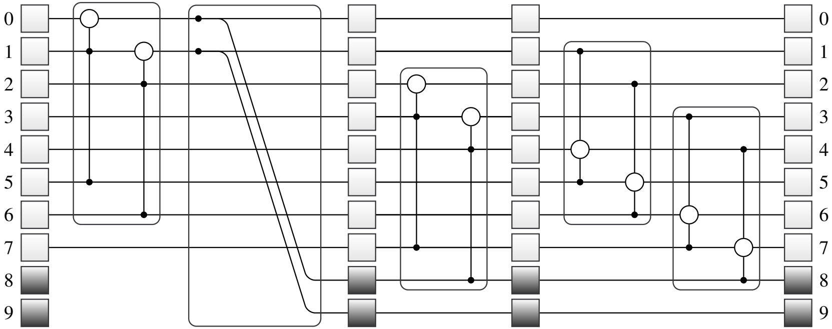


MT19937 SIMD Loop Scheme



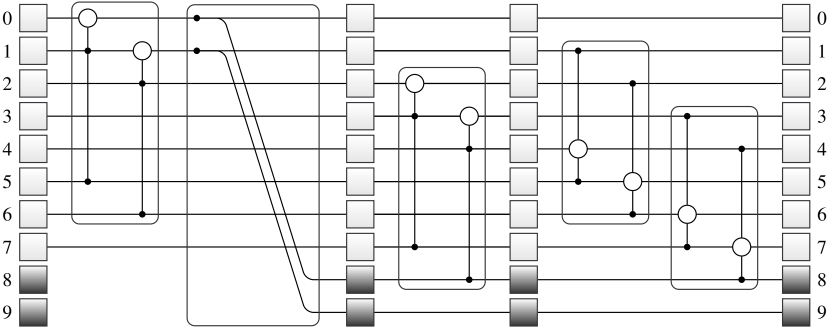
- example: two-element-vector; reality: up to eight-element-vector

MT19937 SIMD Loop Scheme



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

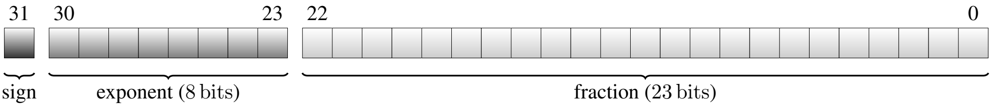
MT19937 SIMD Loop Scheme



- 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

Implementation of Uniform Distribution Functions

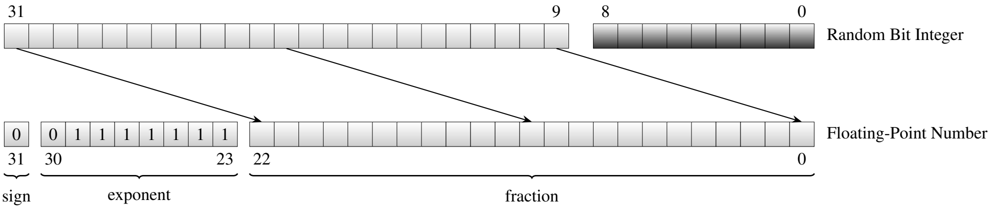
Real Uniform Distribution: Floating-Point Encoding



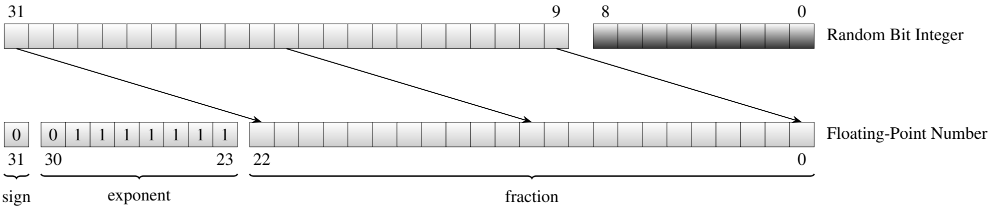
$$x = (-1)^s \cdot m \cdot 2^{e-o}$$

- IEEE 754
- we use only normalized numbers

Real Uniform Distribution

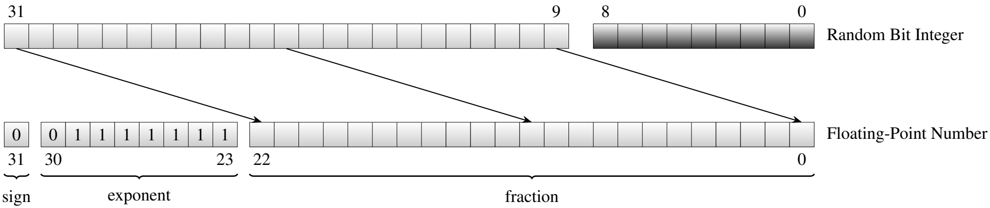


Real Uniform Distribution



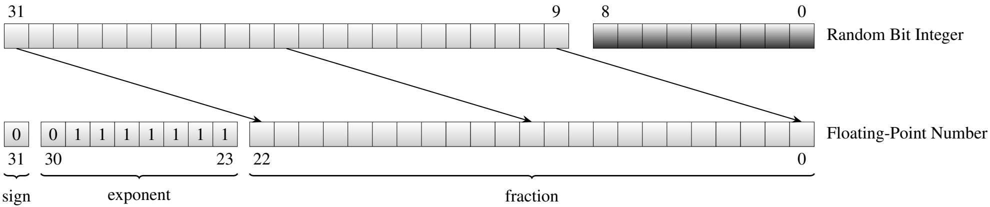
- get random integer

Real Uniform Distribution



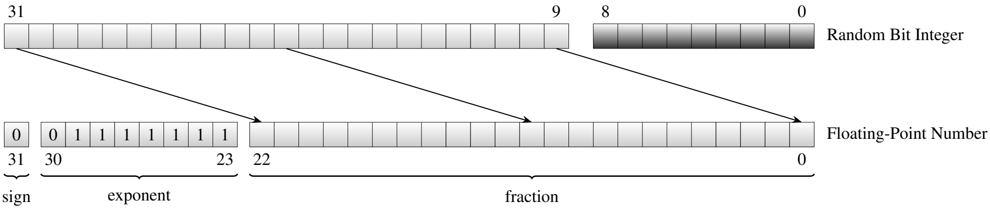
- get random integer
- shift bits with highest entropy into fraction part

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

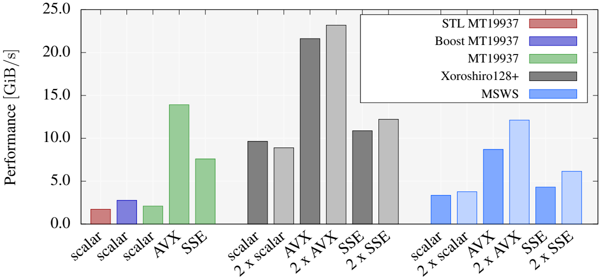
- Consistency and Correctness: Unit Tests, API Tests, Examples

Tests and Performance

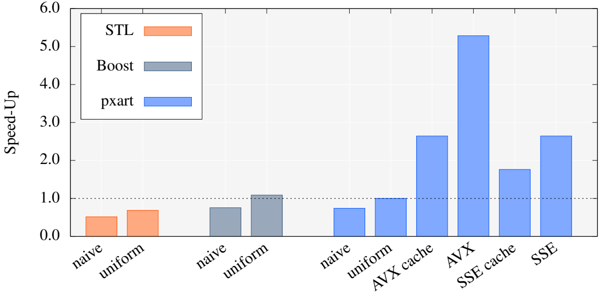
- Consistency and Correctness: Unit Tests, API Tests, Examples
- Statistical Performance: TestU01, dieharder

Tests and Performance

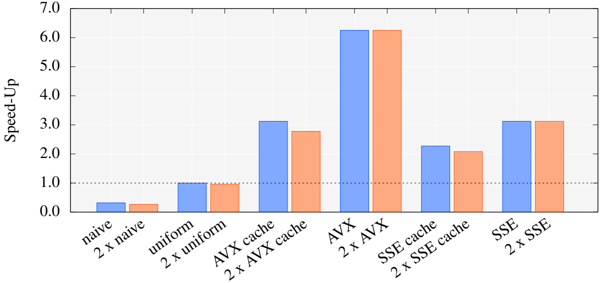
- Consistency and Correctness: Unit Tests, API Tests, Examples
- Statistical Performance: TestU01, dieharder
- Performance: Filling a Cache, Monte Carlo π



MT19937 Speed-Up Monte Carlo π



Xoroshiro128+ Speed-Up Monte Carlo π



Comparison to Intel MKL VSL and RNGAVXLIB

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

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 when applied in Monte Carlo π benchmark

Comparison to Intel MKL VSL and RNGAVXLIB

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 when applied in Monte Carlo π benchmark
- scalar interface reduces performance

Comparison to Intel MKL VSL and RNGAVXLIB

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 when applied in Monte Carlo π benchmark
- scalar interface reduces performance
- Intel MKL VSL always fills vector of data

Comparison to Intel MKL VSL and RNGAVXLIB

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 when applied in Monte Carlo π benchmark
- scalar interface reduces performance
- Intel MKL VSL always fills vector of data
- benchmarks are biased

Conclusions and Future Work

Comparison

	pXart	RNGAVXLIB	Intel MKL
Portable	✔	✘	✘
User-Friendly API	✔	✘	✘
Header-Only	✔	✘	✘
Open Source	✔	✔	✘
Documentation	✔	✘	$\sim$
Distributions	✘	✔	✔
CMake and build2 Support	✔	✘	✘
Dependency-Free	✔	✔	$\sim$
Easy-to-get	✔	$\sim$	$\sim$
AVX512	✘	$\sim$	✔

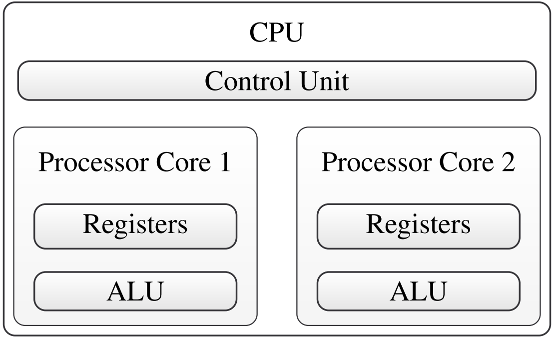
Conclusions and Future Work

- possible applications in simulations
- mt19937 vs. xoroshiro128+

Thank you for Your Attention!

References

Processor


CPU
Control Unit
Processor Core 1
Registers
ALU
Processor Core 2
Registers
ALU

Memory Hierarchy