Design and Implementation of Vectorized Pseudorandom Number Generators

Master's Thesis Defense and Presentation

Markus Pawellek

May 21, 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

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

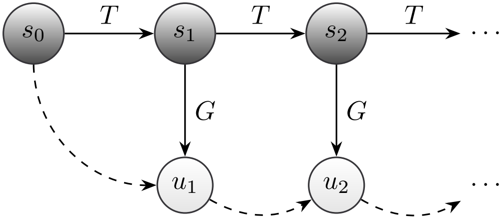
What is a random sequence?

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

Disadvantages:

- Unreproducibility
- Speed Limitations

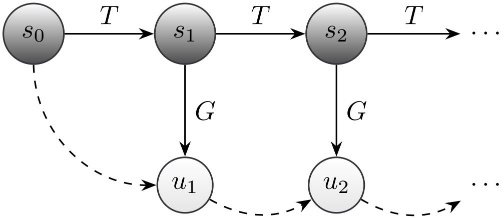
Pseudorandom Number Generator Definition



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

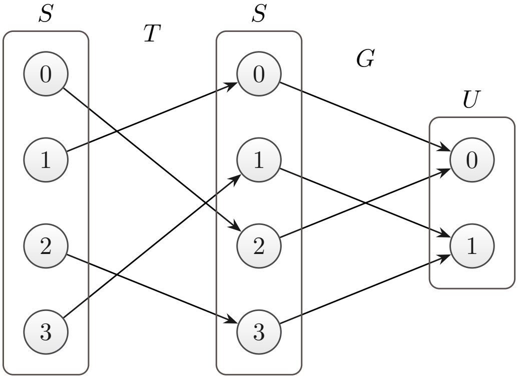
$$s_0 \in S, \qquad s_{n+1} := T(s_n), \qquad u_n := G(s_n)$$

Pseudorandom Number Generator Concept



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

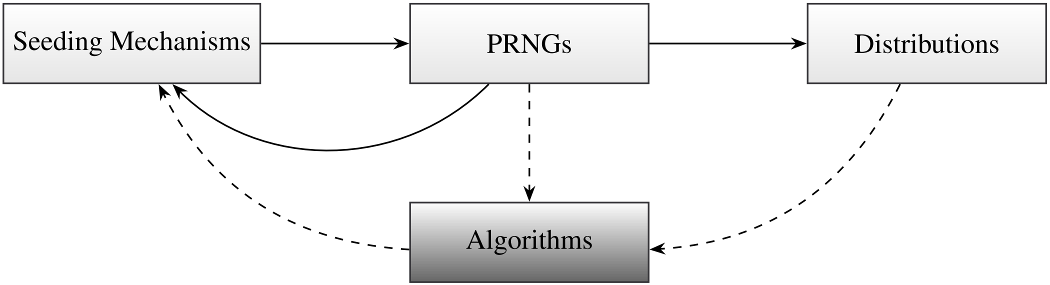
Pseudorandom Number Generator Example



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

Design of the Library

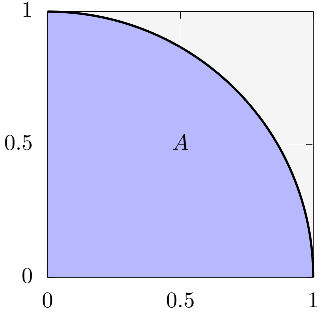
Design Components



Usage in C++

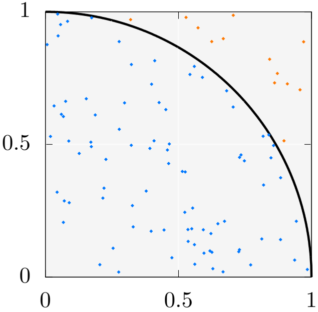
```cpp
#include <pxart/pxart.hpp>
//
std::random_device rd{};
//
pxart::mt19937 rng1{};
pxart::mt19937 rng1{rd};
pxart::mt19937 rng1{pxart::mt19937::default_seeder{rd()}};
//
pxart::xrsr128p rng2{rng1};
//
const auto x = pxart::uniform<float>(rng1);
//
const auto y = pxart::uniform(rng2, -1.0f, 1.0f);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

Computation of π



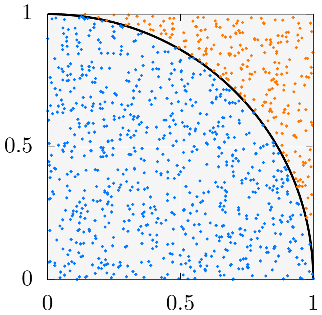
$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N}$$

Computation of π



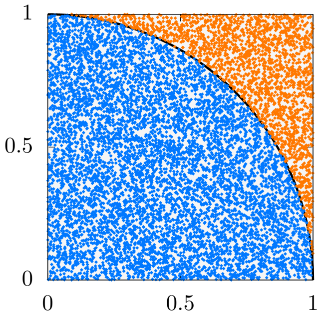
$$A = \frac{\pi}{4}, \qquad \hat{\pi} = \frac{4N_A}{N} = \frac{4 \cdot 87}{100} = 3.48$$

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

```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;

// ...
```

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

$$=$$

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

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

SIMD Features in C++

- SSE (128-bit registers) and AVX (256-bit registers) instruction set features
- Assembler vs. Automatic Vectorization vs. Manual Vectorization by 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?

- 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

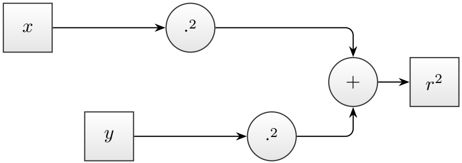
SIMD Architecture

What are conditions for good vectorization?

- no data dependency
- same processing pipeline
- branchless execution

SIMD Example

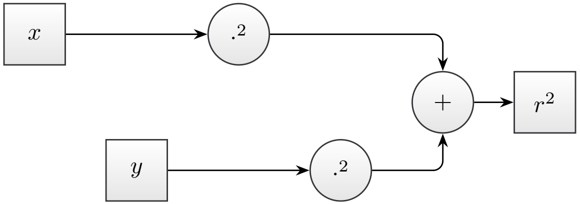
$$x, y \in \mathbb{R}, \qquad r^2 = x^2 + y^2$$



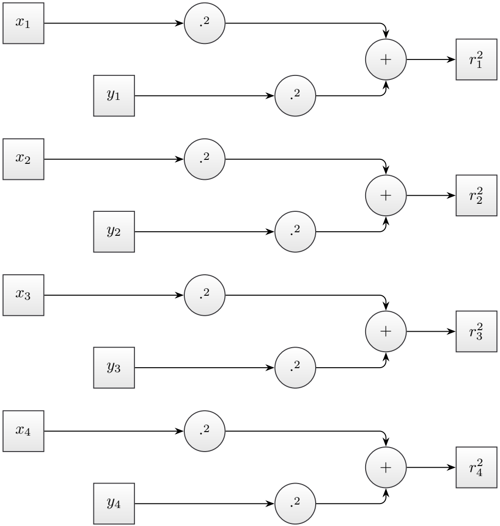

```
double x = pxart::uniform<double>(rng);
double y = pxart::uniform<double>(rng);

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

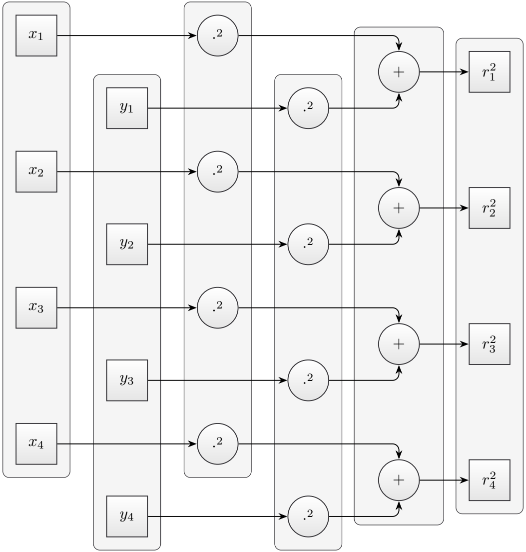
SIMD Example



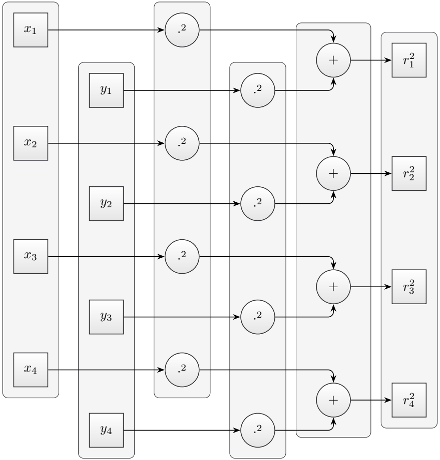
SIMD Example



SIMD Example



SIMD Example

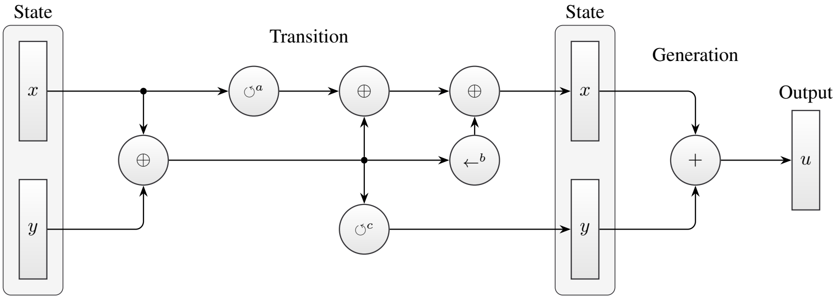



```
__m256d x = pxart::uniform<double>(vrng);
__m256y y = pxart::uniform<double>(vrng);

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

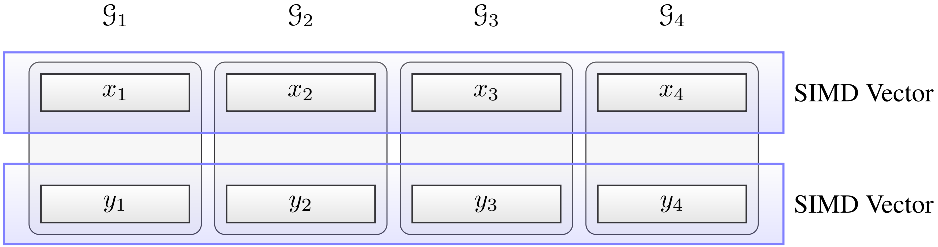
Implementation

Xoroshiro128+



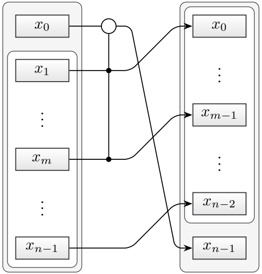
- scrambled linear PRNG
- 128-bit state, 64-bit output
- period: $2^{128} - 1$
- jump operations

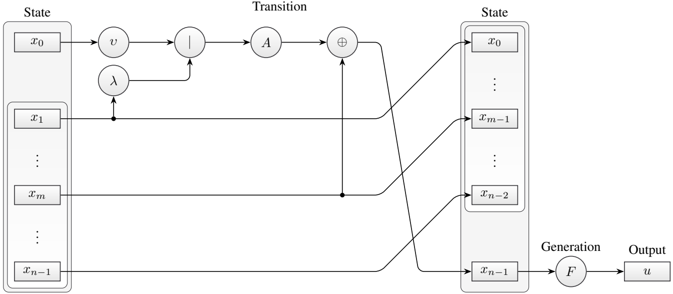
Xoroshiro128+ Scalar and Vectorized



- several parallelization techniques
- multiple instances of the same generator
- seeding variations

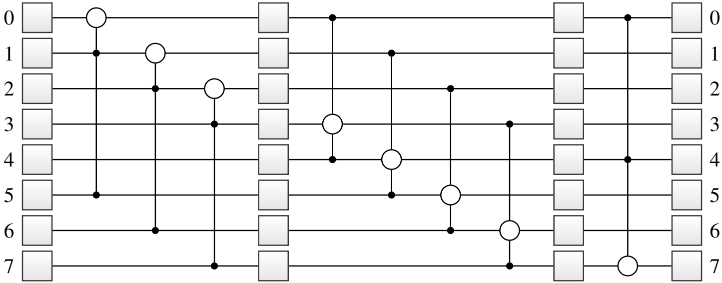
Mersenne Twister MT19937



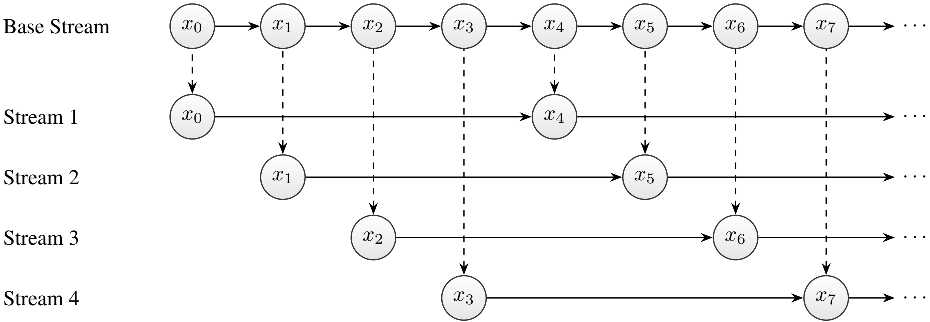


- de-facto standard
- linear PRNG
- 19937-bit / 19968-bit $\approx 2.4\,\mathrm{KiB}$ state, 32-bit output
- period: $2^{19937} - 1$
- 623-dimensional equidistributed

MT19937 Scalar

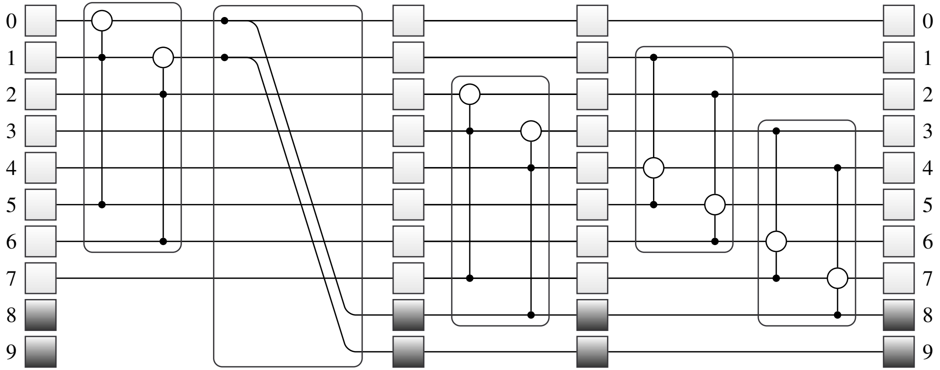


MT19937 SIMD



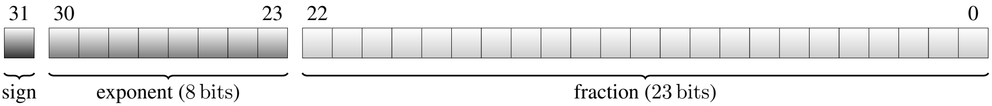
- vectorized generator will give same output as scalar one, only faster

MT19937 SIMD



- implementation could be tested for the same output as the standard

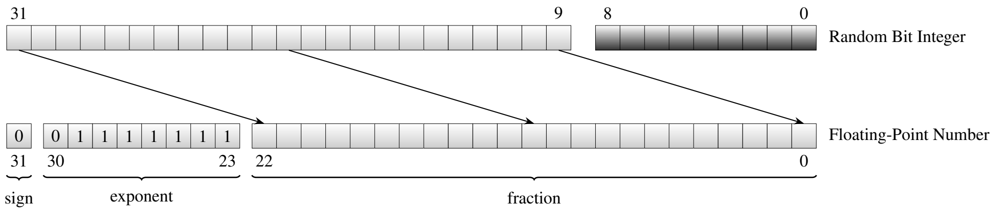
Real Uniform Distribution: Floating-Point Encoding



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

- IEEE 754
- we use only normalized numbers

Real Uniform Distribution



- get random integer
- shift bits with highest entropy into fraction part
- set sign and exponent to generate random floating-point value in $[1, 2)$
- subtract one
- due independent operations easily vectorizable

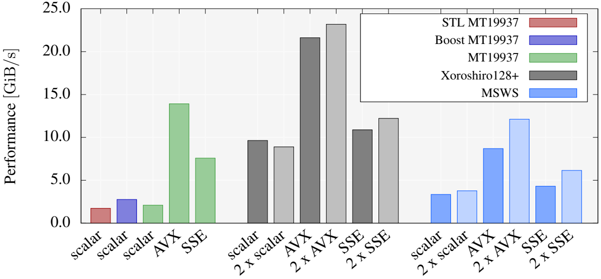
Integer Uniform Distribution

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

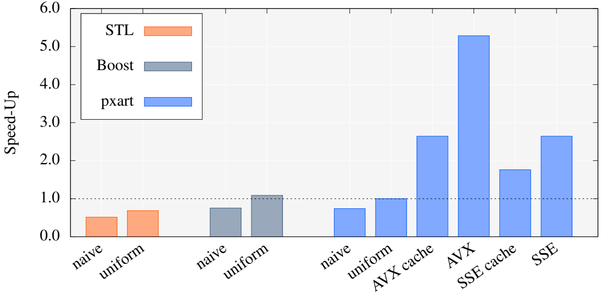
- only approximation possible with vectorization
- first use bound by next more accurate multiplication
- then add offset

Evaluation and Results

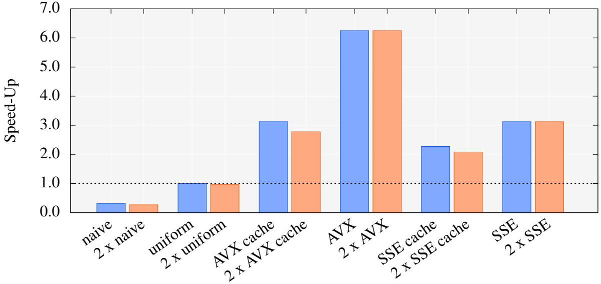
Evaluation and Results



- Tests: TestU01, dieharder, Unit Tests, API Tests
- Performance: Benchmarks



Speed-Up
6.0
5.0
4.0
3.0
2.0
1.0
0.0
STL
Boost
pxart
naive
uniform
naive
uniform
naive
uniform
AVX cache
AVX
SSE cache
SSE



Speed-Up
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
naive
2 x naive
uniform
2 x uniform
AVX cache
2 x AVX cache
AVX
2 x AVX
SSE cache
2 x SSE cache
SSE
2 x SSE

Conclusions and Future Work

Comparison

	pXart	RNGAVXLIB	Intel MKL
Portable	✔	✘	✘
User-Friendly API	✔	✘	✘
Header-Only	✔	✘	✘
Open Source	✔	✔	✘
Documentation	✔	✘	✔
Distributions	✘	✔	✔
CMake and build2 Support	✔	✘	✘
Dependency-Free	✔	?	?

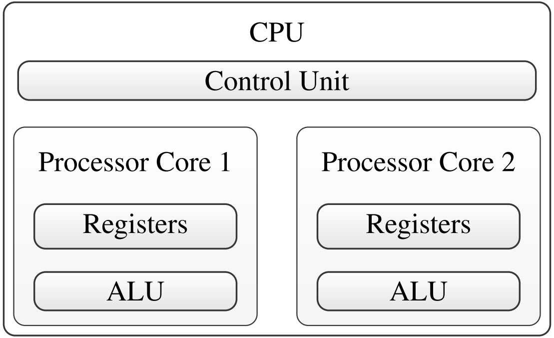
Conclusions and Future Work

- 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