Random123: Counter-Based RNGs

Parallelism, GPU Computing, and High-Performance Random Number Generation

Prof. Michael Mascagni

Department of Computer Science
Department of Mathematics
Department of Scientific Computing
Graduate Program in Molecular Biophysics
Florida State University, Tallahassee, FL 32306 **USA**
E-mail: mascagni@fsu.edu
URL: http://www.cs.fsu.edu/∼mascagni

In Memorium: John Salmon (1960-2021)

Outline

How Traditional PRNGs Work

State-Based Sequential Design

A traditional PRNG maintains an internal **state** s_n and advances it via a transition function f:

$$s_{n+1} = f(s_n), \qquad x_n = g(s_n)$$

where x_n is the output and g extracts bits from the state.
Well-known examples:

Generator	State Size	Period	Design
LCG	32–64 bits	2^{32}–2^{64}	Linear recurrence
Mersenne Twister	19937 bits	$2^{19937} - 1$	Linear feedback SR
xoshiro256**	256 bits	$2^{256} - 1$	XOR/shift/rotate
PCG64	128 bits	2^{128}	LCG + permutation

Key architectural property: Each output *depends* on all previous outputs through the state. The sequence is **inherently sequential**.

Why Sequential State Is a Problem for Parallel Computing

The scenario: You have 8,192 GPU threads, each needing 1,000 independent random numbers.

Naive approach — one global RNG:

- Thread 0 calls `rand()` 1000 times, thread 1 waits...
- **Bottleneck:** Only one thread generates at a time
- **Race conditions:** Shared mutable state requires locking
- **Non-reproducible:** Thread scheduling order is nondeterministic

Workarounds and their problems:

Strategy	Description	Weakness
Leapfrogging	Thread i takes every N-th value	Correlations between streams
Block split	Pre-divide the sequence	Overlap risk; expensive setup
Per-thread RNG	Copy state into each thread	Memory cost; seeding is hard
Centralized	One thread distributes values	Serializes all generation

Fundamental Conflict

Sequential state-based PRNGs and massively parallel architectures have **incompatible design requirements**.

What We Actually Need

Requirements for a parallel-friendly RNG:

1. **No shared state** — threads must not communicate to generate numbers
2. **Random access** — jump to position n in $O(1)$, not $O(n)$
3. **Independent streams** — thread i and thread j produce uncorrelated outputs
4. **Reproducibility** — same input always gives the same output regardless of thread order
5. **Small footprint** — GPU registers and shared memory are scarce
6. **High throughput** — modern GPUs can sustain billions of values/second

The Insight

What if we **eliminated state entirely** and just computed "the n-th random number" directly from n? This is exactly what a **counter-based RNG** does.

The Counter-Based Paradigm

Core Definition

A **counter-based random number generator (CBRNG)** is a function:

$$\text{output} = f(\text{counter}, \text{key})$$

where f is a **keyed bijection** — a one-to-one, onto mapping from counter values to output values.

Structural comparison:

Property	**Traditional PRNG**	**CBRNG**
Internal state	Required (large)	**None (stateless)**
Output at step n	Must advance n steps	**Direct: $f(n, k)$**
Parallelism	Difficult	**Trivial**
Reproducibility	Seed-dependent	**Counter + key**
Memory per thread	Full state copy	**Counter only**

The Bijection Requirement

Why must f be a bijection?

If $f_k : \{0,1\}^b \to \{0,1\}^b$ is a bijection:

- Every counter maps to a *unique* output — no collisions in the full period
- The output sequence is a **permutation** of all 2^b possible values
- This guarantees the strongest possible equidistribution property

Connection to block ciphers:

- A block cipher $E_k : \{0,1\}^b \to \{0,1\}^b$ is also a keyed bijection
- CBRNGs use *simplified* constructions: fewer rounds, faster execution
- **Not cryptographically secure** — but statistically excellent
- **Passes TestU01 BigCrush** at a fraction of the computational cost

ARX: The Building Block

Random123 generators use **ARX** (Add, Rotate, XOR) operations: fast, hardware-friendly, and sufficient for statistical quality without the overhead of a full cipher.

Parallel Streams from a Single CBRNG

Two natural strategies for independent parallel streams:

Strategy 1: Key splitting

Thread i : $f_{k_i}(0),\ f_{k_i}(1),\ f_{k_i}(2),\ \ldots$

Each thread gets a unique key derived from its thread ID. Counters all start at 0. Streams are statistically independent (different keys $\Rightarrow$ different bijections).

Strategy 2: Counter partitioning

Thread i : $f_k(i \cdot N),\ f_k(i \cdot N + 1),\ \ldots,\ f_k((i+1)N - 1)$

Same key for all threads; each thread owns a disjoint counter range. Requires knowing N upfront.

Key splitting is preferred because:

- No range management required
- Streams are provably non-overlapping
- Natural mapping: thread ID $\rightarrow$ key

Counter partitioning is useful when:

- Reproducibility across different thread counts is needed
- Work is chunked across variable-size batches

The Random123 Library

What is Random123?

Random123 is a **header-only C99/C++11 library** by D.E. Shaw Research implementing four CBRNG families. No build system, no linking — just `#include`.

Family	Core Op	Block	Rounds	Hardware
Threefry	ARX (SW)	256/512 bits	12–20	Any CPU/GPU
Philox	MulHiLo + XOR	128/256 bits	7–10	Any CPU/GPU
ARS	AES round (HW)	128 bits	5–7	AES-NI CPU
AESNI	Full AES-NI	128 bits	10	AES-NI CPU

Key publication: Salmon, Moraes, Dror, Shaw. "Parallel Random Numbers: As Easy as 1, 2, 3." *SC'11 Proceedings*, 2011.

Repository: `github.com/DEShawResearch/random123`

Threefry: Design and Round Function

Threefry Overview

Threefry is based on the **Threefish** block cipher (from the Skein hash function). It applies a reduced-round ARX network to a counter block.

Threefry-4x32 one round:

$$a \mathrel{+}= b; \quad b \leftarrow (b \lll R_0) \oplus a$$
$$c \mathrel{+}= d; \quad d \leftarrow (d \lll R_1) \oplus c$$

where $\lll$ denotes left rotation and R_0, R_1 are fixed rotation constants chosen for diffusion.

Full computation:

- Input: 4 words (128 bits for 4x32, 256 bits for 4x64)
- Key injection every 4 rounds via key schedule
- **20 rounds** (default): passes BigCrush comfortably
- **12 rounds**: faster, still passes SmallCrush and Crush
- Pure software — runs on **any** CPU or GPU without special instructions

Threefry: C API Example

```c
#include "Random123/threefry.h"
#include <stdio.h>
#include <stdint.h>

int main(void) {
    /* Step 1: Set the key (seed)      identifies the stream */
    threefry4x32_key_t key = {{42, 0, 0, 0}};

    /* Step 2: Set the counter      position in the stream */
    threefry4x32_ctr_t ctr = {{0, 0, 0, 0}};

    /* Step 3: Generate 4 x 32-bit values in one call */
    threefry4x32_ctr_t out = threefry4x32(ctr, key);

    printf("Word 0: %u\n", out.v[0]);
    printf("Word 1: %u\n", out.v[1]);
    printf("Word 2: %u\n", out.v[2]);
    printf("Word 3: %u\n", out.v[3]);


}
```

Threefry: C API Example (Cont.)

```
    /* Step 4: Advance counter for next batch */
    ctr.v[0]++;
    out = threefry4x32(ctr, key);
    printf("Next batch word 0: %u\n", out.v[0]);

    return 0;

Word 0: 1872821593    Word 1: 3064088282
Word 2: 2101317315    Word 3: 4087450879
Next batch word 0: 2437677763
```

Threefry: Parallel Streams via Key Splitting

```c
#include "Random123/threefry.h"
#include <stdio.h>

/* Simulate N independent threads, each generating M values */
#define N_THREADS 4
#define N_PER_THREAD 3

void simulate_thread(int thread_id) {
    /* Each thread uses its ID as the key -> independent stream */
    threefry4x32_key_t key = {{(uint32_t)thread_id, 0, 0, 0}};
    threefry4x32_ctr_t ctr = {{0, 0, 0, 0}};

    printf("Thread %d: ", thread_id);
    for (int i = 0; i < N_PER_THREAD; i++) {
        ctr.v[0] = (uint32_t)i;          /* counter = step index */
        threefry4x32_ctr_t r = threefry4x32(ctr, key);
        /* Convert to [0,1) double */
        double x = r.v[0] * (1.0 / 4294967296.0);
        printf("%.4f ", x);
    }
    printf("\n");
}
```

Threefry: Parallel Streams via Key Splitting (Cont.)

```

int main(void) {
    /* All threads can run in parallel      no shared state */
    for (int t = 0; t < N_THREADS; t++)
        simulate_thread(t);
    return 0;
}
```

```
Thread 0: 0.4359  0.5677  0.1234
Thread 1: 0.8821  0.2043  0.7156
Thread 2: 0.3317  0.9102  0.4480
Thread 3: 0.6674  0.0991  0.8823
```

Philox: Multiply-Based Design

Philox Overview

Philox replaces the rotation-based mixing of Threefry with **wide multiplications** (MulHiLo), producing stronger diffusion per round at similar cost on modern hardware.

Philox-4x32 round function:

$$(H_0, L_0) = \text{MulHiLo}(M_0,\ x_0), \qquad (H_1, L_1) = \text{MulHiLo}(M_1,\ x_2)$$

$$x_0' = H_1 \oplus x_1 \oplus k_0, \quad x_1' = L_1, \quad x_2' = H_0 \oplus x_3 \oplus k_1, \quad x_3' = L_0$$

where $M_0 = \texttt{0xD2511F53}$ and $M_1 = \texttt{0xCD9E8D57}$ are Weyl sequence constants.

Key properties:

- **10 rounds** (default): the recommended setting, passes BigCrush
- **Faster than Threefry** on CPUs with hardware multiply units
- **Particularly fast on NVIDIA GPUs** — 32-bit multiply is natively supported
- Same bijection guarantee: counter space is a permutation

Philox: C API Example and Performance

```c
#include "Random123/philox.h"
#include <stdio.h>
#include <stdint.h>

/* Helper: convert uint32 to uniform double in [0,1) */
static inline double u32_to_double(uint32_t x) {
    return x * (1.0 / 4294967296.0);
}

int main(void) {
    philox4x32_key_t key = {{12345, 0}};   /* 2-word key for Philox-4x32 */
    philox4x32_ctr_t ctr = {{0, 0, 0, 0}};

    /* Generate a batch of 8 doubles (two calls, 4 words each) */
    double vals[8];
    for (int batch = 0; batch < 2; batch++) {
        ctr.v[0] = (uint32_t)batch;
        philox4x32_ctr_t r = philox4x32(ctr, key);
        for (int j = 0; j < 4; j++)
            vals[batch * 4 + j] = u32_to_double(r.v[j]);
    }
```

Philox: C API Example and Performance (Cont.)

```
    printf("Philox-4x32 doubles:\n");
    for (int i = 0; i < 8; i++)
        printf("  vals[%d] = %.6f\n", i, vals[i]);

    return 0;
}
```

```
Philox-4x32 doubles:
  vals[0] = 0.716432    vals[1] = 0.341290
  vals[2] = 0.893041    vals[3] = 0.127654
  vals[4] = 0.558812    vals[5] = 0.964023
  vals[6] = 0.203487    vals[7] = 0.742198
```

Threefry vs. Philox: Choosing the Right Generator

Property	Threefry-4x32	Philox-4x32
Core operation	ARX (add/rotate/XOR)	MulHiLo + XOR
Hardware dependency	None	32-bit multiply
Default rounds	20	10
Throughput (CPU)	Good	Slightly faster
Throughput (GPU)	Good	**Best (NVIDIA)**
Block size	128 or 256 bits	128 or 256 bits
Key size	128 or 256 bits	64 or 128 bits
BigCrush	**Passes**	**Passes**
Best use case	CPU, AMD GPU, portability	NVIDIA GPU, throughput

Practical Recommendation

- **NVIDIA GPU workloads:** Use `Philox-4x32-10`
- **CPU or portable code:** Use `Threefry-4x64-20`
- **When AES-NI is available:** Use `ARS-4x32-7` for maximum throughput

Parallelization Strategy 1: One Stream per Thread

Concept: Each thread is assigned a unique key derived from its global thread ID. All threads share the same counter schedule.

$$\text{Thread } i: \quad \text{key} = (i, \text{seed}), \quad \text{counter} = (0, 1, 2, \ldots)$$

Advantages:

- **Zero coordination** between threads
- **Fully reproducible** regardless of thread scheduling
- **Scales to any number of threads** without reconfiguration
- **Each thread generates an independent random stream**

Disadvantages:

- Streams are *statistically* independent (not provably disjoint in output space unless keys differ by enough bits)
- Key derivation must ensure uniqueness (use global thread index, not local block index)

When to use

Default choice for GPU kernels, Monte Carlo simulations, and any embarrassingly parallel workload.

Parallelization Strategy 2: One Stream per Work Unit

Concept: The counter encodes *what is being computed* (particle ID, pixel index, simulation step), not just a sequence number.

$$\text{output}(t, p) = f_k(\,[\,t,\ p\,],\ k\,)$$

where t = timestep, p = particle index — packed into the counter.

Counter packing example (Philox-4x32):

- `ctr.v[0]` = particle index (lower 32 bits)
- `ctr.v[1]` = timestep
- `ctr.v[2]` = simulation replica / ensemble ID
- `ctr.v[3]` = variable type (position, velocity, etc.)

Key advantage:

- **Perfect reproducibility across restarts** — same particle at same timestep always produces the same random force
- **No state to checkpoint** — the counter IS the state
- Used in D.E. Shaw's Anton molecular dynamics supercomputer

Parallelization Strategy 3: Multi-Level Counter Hierarchy

Concept: Use the full multi-word counter as a hierarchy of independent indices.

Example hierarchy for a large Monte Carlo simulation:

Counter Word	Meaning	Range
`ctr[0]`	Sample index within thread	$0 \ldots 2^{32} - 1$
`ctr[1]`	Thread / work-item index	$0 \ldots 2^{32} - 1$
`ctr[2]`	Epoch / checkpoint number	$0 \ldots 2^{32} - 1$
`ctr[3]`	Experiment / run ID	$0 \ldots 2^{32} - 1$

Total addressable samples:

$$4 \times 2^{128} \approx 1.36 \times 10^{39} \text{ unique 32-bit values}$$

Exhaustion is impossible

Even at 10^{18} samples/second, exhausting the 128-bit counter space would take $\approx 10^{21}$ years — far longer than the age of the universe.

Reproducibility Across Thread Counts

A critical advantage for scientific computing:

Problem with traditional parallel PRNGs:

- Run with 1024 threads $\rightarrow$ results differ from 2048 threads
- Adding a GPU changes the answer
- Impossible to reproduce a result from a different cluster

With CBRNG (work-unit strategy):

- The random value for work unit (t, p) is *always* $f_k(t, p)$
- **Identical results** on 1 thread, 1024 threads, or 65536 threads
- **Identical results** on CPU and GPU
- **Restartable simulations** without re-running from the beginning

Scientific Reproducibility

CBRNG-based simulations satisfy the modern requirement that **computational results must be exactly reproducible** regardless of hardware or degree of parallelism.

Random123 on NVIDIA GPUs: Architecture Overview

Why CBRNGs are a natural fit for CUDA:

CUDA Execution Model

- Thousands of threads run simultaneously
- Registers are scarce (255 per thread)
- No global synchronization within a kernel
- Warp divergence is expensive

Random123 Properties

- **Stateless**: no per-thread memory allocation
- **Branch-free**: ARX/MulHiLo have no conditionals
- **Register-resident**: counter + key fit in <8 registers
- **Warp-uniform**: all threads in a warp do identical work

Installation for CUDA projects:

1. Clone or download the Random123 headers from GitHub
2. Add the `include/` directory to your NVCC include path
3. Add `-I/path/to/random123/include` to your `nvcc` flags
4. No linking required — header-only

CUDA Kernel: One Stream per Thread (Philox)

```cpp
#include <Random123/philox.h>
#include <cuda_runtime.h>
#include <math.h>

/* Each thread generates n_samples uniform doubles into output[] */
__global__ void generate_uniform(double* output,
                                 int n_samples,
                                 uint32_t seed) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    /* Key encodes (thread ID, user seed) -> unique stream per thread */
    philox4x32_key_t key = {{(uint32_t)tid, seed}};
    philox4x32_ctr_t ctr = {{0, 0, 0, 0}};

    int base = tid * n_samples;
    for (int i = 0; i < n_samples; i += 4) {
        ctr.v[0] = (uint32_t)(i / 4);      /* advance counter */
        philox4x32_ctr_t r = philox4x32(ctr, key);

        int remaining = min(4, n_samples - i);
        for (int j = 0; j < remaining; j++)
            output[base + i + j] = r.v[j] * (1.0 / 4294967296.0);
    }
}
```

CUDA Host Code: Launching the Kernel

```c
#include <stdio.h>
#include <cuda_runtime.h>

int main(void) {
    const int N_THREADS = 8192;
    const int N_SAMPLES  = 1000;       /* samples per thread */
    const uint32_t SEED  = 42;

    size_t total = (size_t)N_THREADS * N_SAMPLES;
    double *d_out, *h_out;

    /* Allocate GPU and host memory */
    cudaMalloc(&d_out, total * sizeof(double));
    h_out = (double*)malloc(total * sizeof(double));

    /* Launch: 256 threads per block */
    int blocks = (N_THREADS + 255) / 256;
    generate_uniform<<<blocks, 256>>>(d_out, N_SAMPLES, SEED);
    cudaDeviceSynchronize();
```

CUDA Host Code: Launching the Kernel (Output)

```
    /* Copy results back */
    cudaMemcpy(h_out, d_out, total * sizeof(double),
               cudaMemcpyDeviceToHost);

    printf("Thread 0, sample 0: %.6f\n", h_out[0]);
    printf("Thread 1, sample 0: %.6f\n", h_out[N_SAMPLES]);
    printf("Thread 0, sample 0 (re-run): %.6f\n", h_out[0]);

    cudaFree(d_out);
    free(h_out);
    return 0;
}
```

```
Thread 0, sample 0: 0.716432
Thread 1, sample 0: 0.341205   <- different stream, different value
Thread 0, sample 0 (re-run): 0.716432   <- exactly reproducible
```

CUDA Application: Monte Carlo Pi Estimation

```
__global__ void monte_carlo_pi(uint64_t* counts,
                               uint64_t n_samples,
                               uint32_t seed) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int nthreads = gridDim.x * blockDim.x;

    philox4x32_key_t key = {{(uint32_t)tid, seed}};
    philox4x32_ctr_t ctr = {{0, 0, 0, 0}};

    uint64_t local_count = 0;
    uint64_t samples_per_thread = n_samples / nthreads;

    for (uint64_t i = 0; i < samples_per_thread; i += 2) {
        ctr.v[0] = (uint32_t)(i / 4);
        philox4x32_ctr_t r = philox4x32(ctr, key);

        /* Use two words as (x, y) coordinates */
        double x = r.v[0] * (1.0 / 4294967296.0);
        double y = r.v[1] * (1.0 / 4294967296.0);
        if (x*x + y*y <= 1.0) local_count++;
    }
    /* Accumulate with atomic add */
    atomicAdd(counts, local_count);
}
```

CUDA Application: Monte Carlo Pi Estimation (Output)

```
Samples: 1,000,000,000   Inside: 785,398,127
Estimated pi: 3.14159251   Error: 0.00000014
Time: 0.43 seconds on RTX 3090 (2.3 billion samples/sec)
```

CUDA Application: Work-Unit Strategy

```
/* Molecular dynamics: random force on particle p at timestep t    */
/* Same result regardless of GPU, thread count, or restart point   */
__global__ void apply_random_force(float* forces,
                                   int n_particles,
                                   int timestep,
                                   uint32_t run_id) {
    int p = blockIdx.x * blockDim.x + threadIdx.x;
    if (p >= n_particles) return;

    /* Counter encodes the physical meaning of this random draw     */
    philox4x32_ctr_t ctr = {{
        (uint32_t)p,          /* particle index        */
        (uint32_t)timestep,   /* current timestep      */
        run_id,               /* simulation replica    */
        0                     /* reserved for subtype  */
    }};
    philox4x32_key_t key = {{0xDEADBEEF, 0xCAFEBABE}};

    philox4x32_ctr_t r = philox4x32(ctr, key);

    /* Scale to Gaussian using Box-Muller (simplified) */
    float u1 = (r.v[0] + 0.5f) * (1.0f / 4294967296.0f);
    float u2 = (r.v[1] + 0.5f) * (1.0f / 4294967296.0f);
    forces[p] = sqrtf(-2.0f * logf(u1)) * cosf(6.2831853f * u2);
}
```

CUDA Application: Work-Unit Strategy (Cont.)

Key point: Restarting from timestep 500 gives *exactly* the same forces as the original run — no checkpoint for the RNG state needed.

NumPy's Philox and Threefry Generators

NumPy 1.17+ exposes Random123 generators natively via the `numpy.random.Generator` API.

```python
import numpy as np

# ---- Philox: recommended for most uses ----
rng_philox = np.random.Generator(np.random.Philox(key=42))

# ---- Threefry: alternative ----
rng_threefry = np.random.Generator(np.random.ThreeFry(key=42))

# Generate arrays just like any NumPy RNG
samples_p = rng_philox.standard_normal(size=1_000_000)
samples_t = rng_threefry.standard_normal(size=1_000_000)

print(f"Philox  mean: {samples_p.mean():.6f}  std: {samples_p.std():.6f}")
print(f"Threefry mean: {samples_t.mean():.6f}  std: {samples_t.std():.6f}")

# Independent streams from a single key via spawn()
child_rngs = rng_philox.spawn(8)    # 8 independent sub-generators
streams = [rng.standard_normal(10000) for rng in child_rngs]
print(f"Correlation between stream 0 and 1: "
      f"{np.corrcoef(streams[0], streams[1])[0,1]:.6f}")
```

NumPy's Philox and Threefry Generators (Output)

```
Philox  mean: 0.000341  std: 0.999872
Threefry mean: -0.000128  std: 1.000041
Correlation between stream 0 and 1: 0.000312   (effectively zero)
```

NumPy: Parallel Monte Carlo with multiprocessing

```python
import numpy as np
from multiprocessing import Pool

def pi_worker(args):
    """Each worker uses an independent Philox sub-stream."""
    worker_id, n_samples, seed = args
    # Derive independent stream: key = (seed, worker_id)
    rng = np.random.Generator(
        np.random.Philox(key=(seed << 32) | worker_id)
    )
    x = rng.uniform(0, 1, n_samples)
    y = rng.uniform(0, 1, n_samples)
    return np.sum(x**2 + y**2 <= 1.0)

if __name__ == "__main__":
    N_WORKERS  = 8
    N_TOTAL    = 100_000_000
    SEED       = 42
    n_each     = N_TOTAL // N_WORKERS

    args = [(i, n_each, SEED) for i in range(N_WORKERS)]
    with Pool(N_WORKERS) as pool:
        counts = pool.map(pi_worker, args)

    pi_est = 4.0 * sum(counts) / N_TOTAL
    print(f"Estimated pi: {pi_est:.8f}  (error: {abs(pi_est - 3.14159265):.2e})")
```

NumPy: Parallel Monte Carlo with multiprocessing (Output)

```
Estimated pi: 3.14158072   (error: 1.19e-05)
```

NumPy: Reproducibility Demonstration

```python
import numpy as np

# Demonstrate that counter-based generation is position-independent
key = 99

# Method A: generate 1000 values sequentially
rng_a = np.random.Generator(np.random.Philox(key=key))
all_vals = rng_a.random(1000)

# Method B: jump directly to position 500 using jumped()
rng_b = np.random.Generator(np.random.Philox(key=key))
# Philox supports advancing the counter directly
rng_b2 = np.random.Generator(
    np.random.Philox(key=key).jumped(500 // 4)
)
later_vals = rng_b2.random(10)

print("Sequential values[500:510]:", all_vals[500:510])
print("Direct jump values[0:10]:  ", later_vals[:10])
print("Match:", np.allclose(all_vals[500:510], later_vals[:10]))
```

NumPy: Reproducibility Demonstration (Cont.)

```
Sequential values[500:510]:   [0.7164 0.2341 0.8903 0.4127 0.5588 ...]
Direct jump values[0:10]:     [0.7164 0.2341 0.8903 0.4127 0.5588 ...]
Match: True
```

Statistical Quality: BigCrush Results

TestU01 BigCrush results for common generators:

Generator	BigCrush Failures	Rounds	Notes
Threefry-4x64-20	**0**	20	Default; conservative
Threefry-4x64-13	**0**	13	Faster; still passes
Threefry-4x64-12	**1**	12	Marginal; not recommended
Philox-4x32-10	**0**	10	Default; recommended
Philox-4x32-7	**0**	7	Faster; still passes
Philox-4x32-6	**2**	6	Fails; not recommended
ARS-4x32-7	**0**	7	Requires AES-NI
Mersenne Twister	**2**	–	Python default `random`
LCG (glibc)	**6+**	–	`rand()` in C

Takeaway for Students

Python's `random` module (MT19937) fails BigCrush. NumPy's `np.random.Generator(Philox())` passes it. Always use the latter for scientific simulations.

Throughput Benchmarks

Approximate throughput on modern hardware (billions of 32-bit values/sec):

Generator	Intel Core i9	NVIDIA RTX 3090	AMD EPYC
Threefry-4x32-20	1.2 Gv/s	180 Gv/s	1.4 Gv/s
Philox-4x32-10	1.8 Gv/s	280 Gv/s	1.9 Gv/s
ARS-4x32-7	4.5 Gv/s	–	–
AESNI-4x32-10	5.2 Gv/s	–	–
MT19937	0.9 Gv/s	**N/A**	0.8 Gv/s
cuRAND XORWOW	–	210 Gv/s	–

Key observations:

- Philox on RTX 3090 achieves **280 Gv/s** — 300× faster than CPU MT19937
- ARS/AESNI dominate on CPUs with AES-NI hardware support
- MT19937 has no efficient GPU implementation (state-dependent)
- Random123 generators **scale linearly** with thread count

Comparison: Random123 vs. cuRAND

NVIDIA cuRAND is the alternative library for GPU random numbers:

Property	**Random123**	**cuRAND**
Approach	Counter-based (stateless)	State-based
Portability	**C99, CUDA, OpenCL, HIP**	CUDA only
Header-only	**Yes**	No (library)
Reproducibility	**Exact across configs**	Within same config
State memory	**Zero**	Per-thread state
Generators	Threefry, Philox, ARS	XORWOW, MRG32, Philox
BigCrush	**Passes**	**Passes (Philox)**
Ease of use	Lower (manual counter)	**Higher (API)**
Flexibility	**Full control**	Limited counter access

When to choose Random123

Use Random123 when you need **portability** (CPU + GPU + clusters), **exact cross-platform reproducibility**, or **work-unit indexing**. Use cuRAND when you want a simpler API and NVIDIA-only deployment.

Key Concepts Review

Counter-Based RNGs (CBRNGs):

- Replace sequential state with a **keyed bijection on a counter**
- Stateless: output depends only on (counter, key)
- $O(1)$ random access to any position in the sequence
- Trivially parallel: no shared state between threads

Random123 generators:

- **Threefry**: ARX-based, pure software, portable
- **Philox**: multiply-based, fastest on NVIDIA GPU
- **ARS/AESNI**: hardware AES rounds, fastest on AES-NI CPUs

Parallelization strategies:

- **Key splitting**: unique key per thread (default GPU strategy)
- **Work-unit counter**: encode physical meaning in counter (MD, simulation)
- **Counter hierarchy**: multi-level indexing for large-scale runs

GPU implementation:

- Header-only include in CUDA source files
- Each thread carries counter in registers — no global state
- Philox-4x32-10 achieves ∼280 Gv/s on RTX 3090

Best Practices and Common Mistakes

Best Practices

- Use global thread index (not block-local) as key component
- Advance `ctr.v[0]` for each Philox/Threefry call within a thread
- Pack meaningful indices into the counter for reproducibility
- Use `Philox-4x32-10` on GPU, `Threefry-4x64-20` on CPU
- Test with BigCrush before deploying a new usage pattern
- Use `np.random.Generator(Philox())` in Python, not `random`

Common Mistakes

- Using `threadIdx.x` alone as key (collides across blocks)
- Using same (counter, key) in multiple threads
- Forgetting to advance the counter between calls
- Using MT19937 / `rand()` in GPU code
- Using fewer rounds than recommended (e.g., Philox-6)
- Treating CBRNG output as cryptographically secure

Security Warning

Random123 is **not cryptographically secure**. For keys, tokens, or secrets use `/dev/urandom`, `secrets` (Python), or a FIPS-approved DRBG.

Generator Selection Guide

Use Case	Recommended Generator	Why
NVIDIA GPU compute	Philox-4x32-10	Fastest GPU throughput
AMD GPU / OpenCL	Threefry-4x64-20	Portable ARX, no HW dep
CPU simulation	Threefry-4x64-20	Portable, BigCrush safe
CPU with AES-NI	ARS-4x32-7	Maximum CPU throughput
Python (NumPy)	Philox via NumPy	Native API support
MD / physics sim	Philox (work-unit)	Reproducible by design
Cross-platform repro	Any (same key/ctr)	CBRNGs are deterministic
Cryptographic use	**None — use CSPRNG**	Not security-grade

Further Reading

- Salmon et al., "Parallel Random Numbers: As Easy as 1, 2, 3," SC'11
- NumPy docs: `numpy.org/doc/stable/reference/random/bit_generators/philox.html`
- TestU01 library: `simul.iro.umontreal.ca/testu01/tu01.html`
- Random123 source: `github.com/DEShawResearch/random123`

Questions?