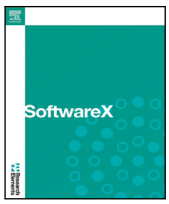
SoftwareX 27 (2024) 101773

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

OpenRAND: A performance portable, reproducible random number generation library for parallel computations

Shihab Shahriar Khan [a,*], Bryce Palmer [b,c], Christopher Edelmaier [d], Hasan Metin Aktulga [a]

[a] Department of Computer Science, Michigan State University, East Lansing, MI 48824, United States of America
[b] Department of Mechanical Engineering, Michigan State University, East Lansing, MI 48824, United States of America
[c] Department of Computational Mathematics, Science and Engineering, Michigan State University, East Lansing, MI 48824, United States of America
[d] Center for Computational Biology, Flatiron Institute, New York, NY 10010, United States of America

ARTICLE INFO

Keywords:
Pseudo random number generation
GPGPU
HPC
C++

ABSTRACT

We introduce OpenRAND, a C++17 library aimed at facilitating reproducible scientific research by generating statistically robust yet replicable random numbers in as little as two lines of code, overcoming some of the unnecessary complexities of existing RNG libraries. OpenRAND accommodates single and multi-threaded applications on CPUs and GPUs and offers a simplified, user-friendly API that complies with the C++ standard's random number engine interface. It is lightweight; provided as a portable, header-only library. It is statistically robust: a suite of built-in tests ensures no pattern exists within single or multiple streams. Despite its simplicity and portability, it remains performant—matching and sometimes outperforming native libraries. Our tests, including a Brownian walk simulation, affirm its reproducibility and ease-of-use while highlight its computational efficiency, outperforming CUDA's cuRAND by up to 1.8 times.

Code metadata

Current code version	V0.9
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00704
Permanent link to Reproducible Capsule	https://codeocean.com/capsule/0144704/tree
Legal Code License	MIT License.
Code versioning system used	Git
Software code languages, tools, and services used	C++ 17
Compilation requirements, operating environments & dependencies	A compiler with C++17 support, optionally CMake
If available Link to developer documentation/manual	https://msu-sparta.github.io/OpenRAND
Support email for questions	khanmd@msu.edu

1. Background

Generating random numbers in a reproducible manner is pivotal for ensuring the reliability and validity of scientific research outcomes, especially in domains fundamentally reliant on random number generation, such as stochastic simulations, machine learning, and computer graphics. This reproducibility permits the exact replication of simulations, facilitating meaningful comparisons devoid of unnecessary variance and ensuring that any disparities arising are solely attributable to external factors, such as discrepancies in floating-point arithmetic ordering. More explicitly, it allows simulations utilizing identical random seeds to be compared directly, eliminating reliance on statistical averages and significantly simplifying debugging and regression testing, as demonstrated in past works [1,2].

In single-threaded environments, the reproducibility of Pseudo Random Number Generators (RNGs) is straightforward since the same initial state (i.e., *seed*) results in an identical sequence of random numbers (i.e., *stream*). Reproducible random number generation for multi-threaded and multi-processed applications, on the other hand, is nontrivial due to the unpredictability of execution orders and the potential for race conditions. Strategies such as utilizing a single RNG instance with synchronization or distributing pre-allocated random

* Corresponding author.
E-mail address: khanmd@msu.edu (Shihab Shahriar Khan).

https://doi.org/10.1016/j.softx.2024.101773
Received 17 October 2023; Received in revised form 11 April 2024; Accepted 20 May 2024
Available online 5 June 2024

numbers to various threads are fraught with scalability issues [1,3]. Instead, contemporary research has predominantly focused on creating multiple independent streams using distinct RNG instances per thread. One approach is to split a single stream into multiple equally sized streams. For example, one could split a single stream with period 2^{64} to 2^{32} independent streams, each with period 2^{32}. This sub-stream approach, however, requires that the generator have a long period and efficient jump-ahead capability [4]. Moreover, statistical independence between these streams is a concern [5–7]. Another commonly employed technique is pre-generating a set of (pseudo)random seeds for each stream and employing them as starting points for generating multiple streams, known as a multi-stream approach. Although the probability of a direct seed collision, resulting in identical streams, can be minimized for RNGs with sufficiently large periods, there remains the potential for statistical correlation among streams, particularly when the bit representations of two seeds are closely related [4]. An additional intriguing class of generators are those based on chaotic dynamic systems [8]; some of these generators are noted for exhibiting strong parallel independence [9], offering a promising avenue for generating uncorrelated streams. For an extensive treatment of parallelization strategies and issues, please refer to [6,7].

The creation of performant Graphical Processing Unit (GPU) compatible RNGs introduces a distinct set of challenges as we shift from CPU to GPU, from dozens or, at most, hundreds of threads to potentially millions within a single node. Memory considerations serves as a prime illustration. In the GPU environment, threads access only a limited amount of high-speed private memory; hence, optimizing local memory usage is essential for maintaining performance. For instance, the default random engine in GNU's libstdc++, Mersenne Twister [10], requires approximately 624 32-bit words for internal state, exceeding by more than double the maximum number of 32-bit registers permitted per thread in CUDA. Further, the absence of certain instructions in GPUs, common in most CPUs, leads to distinct CPU vs. GPU performance characteristics for many generators [11], prompting the need for specialized parallelization strategies. For example, the GPU-adapted Mersenne Twister [12], requires a block of threads to share a single state.

2. Motivation and significance

In High Performance Computing (HPC), the efficiency and reproducibility of pseudo-random number generation are critical. Libraries like cuRAND and rocRAND excel at generating reproducible pseudo-random sequences in parallel environments but face challenges in portability, limiting their adaptability to new architectures. Conversely, while RandomCL [13] and clRNG [14] (though no longer actively developed) successfully address hardware portability, they are limited to the OpenCL software framework. Kokkos [15], on the other hand, offers portability, but its random module offers only one generator with known vulnerabilities [16]. Two challenges arise across these platforms: seeding intricacies and the demands of state management. For seeding, neither RandomCL nor clRNG allow arbitrary seeds; instead, they use a host-based sequential generator to produce seeds for a predetermined number of threads before transferring them to the device's global memory. cuRAND, rocRAND, and Kokkos allow arbitrary seeds to produce distinct random streams but necessitate complicated state initialization procedures. Moreover, these platforms obligate developers to consistently manage memory states throughout the lifecycle of a processing element or thread, adding computational overhead. Overall, while these libraries offer potent tools, they introduce multifaceted challenges, emphasizing the need for a streamlined solution.

Counter-based generators (CBRNGs) offer a promising alternative to traditional pseudo-random generators. Historically, these generators adapted cryptographic algorithms for HPC applications, albeit at reduced cryptographic strength [11,17,18]. By mirroring the counter mode used in block ciphers to encrypt messages exceeding one block, CBRNGs can naturally generate a unique stream within a kernel for a given seed [11,17]. They do so without requiring the complex state transformation functions of traditional sub-stream-based RNGs or the extraordinarily large periods of traditional multi-stream-based RNGs (which come at a high initialization and storage cost). Consequently, CBRNGs have compact states; for instance, OpenRAND's 96-bit state is over 200 times smaller than Mersenne Twister and well within CUDA's maximum 32-bit registers per thread. Furthermore, because of their lightweight footprint and ease of initialization, a separate CBRNG can be instantiated per processing element per kernel without significant overhead, the utility of which has been demonstrated by HOOMD-blue [19]. CBRNGs also champion the avalanche property, wherein a minimal bit alteration in the seed or counter cascades into a significantly altered, statistically independent new stream. This property allows developers to use any unique values for the seed or counter without worrying about correlations between streams, making CBRNGs well-suited for both sub-stream-based and multi-stream-based random number generation [11].

Building on this CBRNG foundation, Random123 is a pivotal library in the realm of counter-based random number generation [11]. It made significant strides by introducing three innovative generators and was at the forefront in terms of performing comprehensive statistical testing. Nevertheless, despite its contributions, Random123 exhibits notable drawbacks. The library does not incorporate more recent CBRNGs [20,21]. Its API skews heavily toward lower-level implementations, exposing developers to the algorithmic details underpinning its generators, making its use clunky and error-prone. Furthermore, its reliance on intrinsics for performance enhancements compromises portability and inflates the codebase.

Addressing the drawbacks of existing libraries in terms of usability, flexibility, and performance (especially on GPUs) and in light of the advantages of CBRNGs, we developed OpenRAND. OpenRAND amalgamates diverse counter-based generators with complementary strengths under a unified, user-centric API that makes generating statistically robust random numbers in parallel trivial. We designed OpenRAND with an emphasis on simplicity and adaptability, without compromising on performance or relying on intrinsics. As a header-only library with its core header files comprising 470 source lines of code, it touts a lightweight footprint and can be incorporated into existing projects with ease. This simplicity facilitates the development of reproducible parallel code, allowing developers to circumvent the complications of API boilerplate and state maintenance. Still, despite our emphasis on simplicity and ease-of-use, OpenRAND matches or surpasses the speed of native CPU and GPU libraries. Most importantly, OpenRAND has undergone extensive statistical testing to ensure its reliability and robustness. Together, these features make OpenRAND a viable alternative to existing more complex, less performant RNG libraries.

3. Software description

3.1. API design

OpenRAND implements a core set of counter-based random number generators complemented by examples, benchmarks, and tests. Complying with the C++17's random engine interface, generators in OpenRAND are compatible with standard library functions, including randomly sampling various distributions. Each generator is created via a constructor that requires two arguments: a 64-bit seed and a 32-bit counter,[1] which, together, produce a unique stream with a period of 2^{32}. Typically, the seed is chosen as the identity of a logical thread or a processing element in the program [6], whereas the counter is used to create multiple streams *per seed* as needed. Currently, OpenRAND

[1] Except for one generator, Squares, which currently accepts 32-bit seeds.

supports a variety of counter-based generators, including Philox [11], Threefry [11], Squares [21], and Tyche [20]. They offer high-quality streams in compact sizes, efficient construction/destruction of RNG objects, and accept arbitrary seeds.

3.2. Seed and counter selection

It is important to emphasize that the seed/counter design of OpenRAND allows developers the flexibility to achieve different RNG parallelization strategies. Consider a particle dynamics simulation where, at each timestep, each particle needs to generate a single random number. In this case, the seed could be the particle's global id and the counter the timestep number, as demonstrated in Section 4. If, on the other hand, one wished to generate multiple random numbers per particle per timestep within different parallel regions, then one could simply store a counter on the particle itself and increment it after each random number is generated. In either case, users can sidestep *all* random state maintenance. In practical terms and in contrast to Nvidia's cuRAND library, this means that developers can often forgo the hassles and performance degradations related to storing states in global memory, launching a separate kernel on GPUs to initialize the states, or the overhead of loading and saving states inside each kernel for every thread. However, one could achieve the same design pattern as cuRAND or Kokkos by initializing a different OpenRAND generator and counter per thread within a global state and using the thread id as the generator seed. To demonstrate this flexibility, we have implemented each of these parallelization strategies within the examples directory of our public repository.[2]

4. Illustrative examples

To demonstrate OpenRAND's advantages, we employ the Brownian Dynamics macro-benchmark from [1], re-implemented in CUDA across three RNG libraries. In this macro-benchmark the velocity of each particle is perturbed by random fluctuations, requiring two random numbers per particle per time step. Fig. 1 highlights OpenRAND, Fig. 2 cuRAND, and Fig. 3 Random123. Unlike cuRAND, both OpenRAND and Random123 use CBRNGs, allowing us to use the unique particle IDs as generator seeds and the time step number as the counter. This choice ensures that the time series of random numbers generated per particle is deterministic and replicable, independent of the number of threads. It also circumvents the state maintenance associated with cuRAND (seen in lines 3–6 and 33–37 of Fig. 2), eliminating the need for memory allocation, state initialization, and continuous memory operations within each kernel thread. Nevertheless, random number generation with Random123 requires excessive boilerplate for initialization and random sampling, as seen in lines 14–29 of Fig. 3, burdening developers with extra coding demands and amplifying the risk of inadvertently introducing bugs. On the other hand, OpenRAND's API required just two lines of code (lines 14 and 15) for both generator initialization and random number computation—over 14 fewer lines than the competing libraries.

```
typedef openrand::Philox RNG;

__global__ void apply_forces(Particle *particles,
                             int counter){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= N) { return; }

  Particle p = particles[i];
  // Apply drag force
  p.vx -= GAMMA / mass * p.vx * dt;
  p.vy -= GAMMA / mass * p.vy * dt;

  // Apply random motion
  RNG local_rand_state(p.pid, counter);
  rnd::double2 r = local_rand_state.draw_double2();
  p.vx += (r.x  * 2.0 - 1.0) * sqrt_dt;
  p.vy += (r.y  * 2.0 - 1.0) * sqrt_dt;
  particles[i] = p;
}

int main(){
  // Initialize particles
  init_particles<<<nblocks, nthreads>>>(particles,
                                        /*counter*/ 0);

  // Simulation loop
  int iter = 0;
  while (iter++ < STEPS) {
    apply_forces<<<nblocks, nthreads>>>(particles,
                                        iter);
    ...
  }
}
```

Fig. 1. Illustrative example of OpenRAND's API: A 2D Brownian walk simulation using OpenRAND's Philox generator.

5. Empirical results

5.1. Performance benchmarks

To assess OpenRAND's performance, we employ two benchmark tests focusing on micro and macro performance metrics across CPU and GPU platforms, respectively.

For our micro-benchmark, we measured the raw random number generation speed for streams of varying sizes across all generators within a CPU, similar to the experiments in [11,13,20,21]. We employed Google benchmark for evaluation, comparing against widely used baselines: GNU libstdc++'s mt19937 [10], the default random engine in GNU's libstdc++, and PCG [4], a widely used, performant RNG library. As Fig. 4(a) shows, OpenRAND is competitive with state-of-the-art CPU-optimized generators for all stream sizes. OpenRAND's edge over mt19937 in smaller streams is likely due to mt19937's complex initialization routine. Threefry and Philox are generally slower as they involve multiple diffusive "rounds", while the rest use a single-round approach.

Transitioning to GPU performance, we employ the previously discussed macro-benchmark, a 2D Brownian dynamics simulation in CUDA. This simulation involved one million independent particles diffusing according to a Brownian random walk. Particles were monitored over 10,000 steps, with the particles influenced by both a velocity-proportional drag force and a random uniform motion. To maintain consistency, pseudo-random number generation for all libraries used their respective Philox generators (For details, refer to code[3]). This benchmark was executed on two Nvidia GPUs: a Tesla V100 PCIe and an A100 SXM.

As seen in Fig. 4(b), OpenRAND outperformed cuRAND by 1.8x and Kokkos' random number module by 1.2x, while saving ~64 MB of GPU memory per million particles and performed on par with Random123. Given the simplistic nature of the kernels used in the program, where random number generation dominates computational cost, such a performance margin between OpenRAND and cuRAND was unanticipated. This comparison between cuRAND, a native library specifically optimized for these platforms, Kokkos, a performance

[2] https://github.com/msu-sparta/OpenRAND

[3] https://github.com/Shihab-Shahriar/brownian-dynamics

```cpp
typedef curandStatePhilox4_32_10_t RNG;

__global__ void rand_init(RNG *rand_state) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i >= N) { return; }
  curand_init(1234, i, 0, &rand_state[i]);
}


template <typename RNG>
__global__ void apply_forces(Particle *particles,
                             RNG* rand_state){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= N) { return; }

  Particle p = particles[i];
  // Apply drag force
  p.vx -= GAMMA / mass * p.vx * dt;
  p.vy -= GAMMA / mass * p.vy * dt;

  // Apply random motion
  RNG local_rand_state = rand_state[i];
  double2 r = curand_uniform2_double(
                  &local_rand_state);
  p.vx += (r.x  * 2.0 - 1.0) * sqrt_dt;
  p.vy += (r.y  * 2.0 - 1.0) * sqrt_dt;
  rand_state[i] = local_rand_state;
  particles[i] = p;
}

int main(){
  // Random number generator setup
  RNG *d_rand_states;
  cudaMalloc((void **)&d_rand_states, N*sizeof(RNG));

  // Initialize random number generators
  rand_init<<<nblocks, nthreads>>>(d_rand_states);

  // Initialize particles
  init_particles<<<nblocks, nthreads>>>(particles,
                                  d_rand_states);

  // Simulation loop
  int iter = 0;
  while (iter++ < STEPS) {
    apply_forces<<<nblocks, nthreads>>>(particles,
                                  d_rand_states);
    ...
  }
}
```

Fig. 2. Illustrative comparison with cuRAND's API: Implementing the Brownian walk simulation from Fig. 1.

```cpp
typedef r123::Philox4x32 RNG;

__global__ void apply_forces(Particle *particles,
                             int counter){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= N) { return; }

  Particle p = particles[i];
  // Apply drag force
  p.vx -= GAMMA / mass * p.vx * dt;
  p.vy -= GAMMA / mass * p.vy * dt;

  // Apply random motion
  RNG rng;
  RNG::ctr_type c={{}};
  RNG::ukey_type uk={{}};
  uk[0] = p.pid;
  RNG::key_type k=uk;

  c[0] = counter;
  c[1] = 0;
  RNG::ctr_type r = rng(c, k);

  uint64_t xu = (static_cast<uint64_t>(r[0]) << 32)
      | static_cast<uint64_t>(r[1]);
  uint64_t yu = (static_cast<uint64_t>(r[2]) << 32)
      | static_cast<uint64_t>(r[3]);
  auto x = r123::u01<double, uint64_t>(xu);
  auto y = r123::u01<double, uint64_t>(yu);

  p.vx += (x  * 2.0 - 1.0) * sqrt_dt;
  p.vy += (y  * 2.0 - 1.0) * sqrt_dt;
  particles[i] = p;
}

int main(){
  // Initialize particles
  init_particles<<<nblocks, nthreads>>>(particles,
                                  /* counter*/ 0);
  // Simulation loop
  int iter = 0;
  while (iter++ < STEPS) {
    apply_forces<<<nblocks, nthreads>>>(particles,
                                        iter);
    ...
  }
}
```

Fig. 3. Illustrative comparison with Random123's API: Implementing the Brownian walk simulation from Fig. 1.

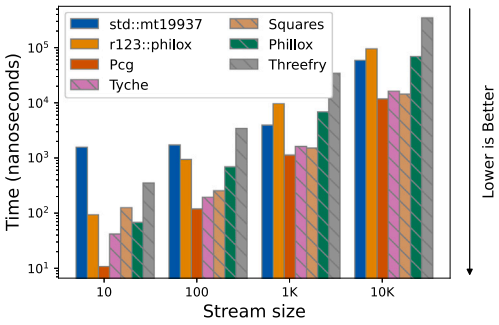
portable library, and Random123, a library that utilized intrinsic instructions to enhance performance, offers confidence that the simplified API and platform-independent code of OpenRAND do not compromise its performance.

5.2. Statistical evaluation

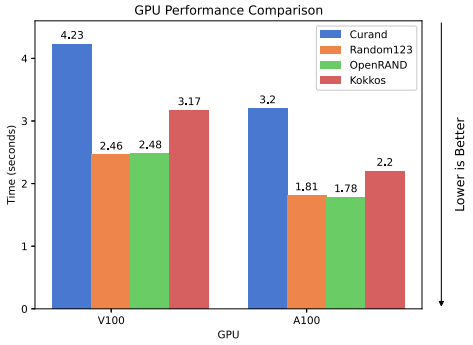
To ensure the quality of our random number generation, OpenRAND exclusively incorporates generators with rigorous empirical validations and long-standing use. Even with this foundation, maintaining statistical integrity requires careful implementation, as subtle bugs can compromise randomness. As such, we perform rigorous quality assurance as part of our Continuous Integration pipeline: every generator within OpenRAND was subjected to statistical testing using the popular frameworks TestU01 [22] and PractRand [23]. These tools offer a suite of complementary statistical tests designed to identify any underlying patterns or irregularities in random streams of data. An example of these tests is the Birthday Spacing test from TestU01 [22], which contrasts empirical results against known analytical solutions to detect potential discrepancies.

We began testing by evaluating individual data streams, probing them to their theoretical limit of 2^{32} integers using PractRand across a comprehensive range of keys and counters. To perform our parallel stream tests, we followed the procedure outlined in [19]—we simulated a scenario with 16,000 particles, generating micro-streams comprising three random numbers for each particle. These individual micro-streams for each particle were first combined into a single concatenated stream. This unified stream was then lengthened over successive iterations to examine correlations across the entire system.

All generators were successfully tested for at least 1TB of data using PractRand and TestU01's comprehensive BigCrush battery of tests. Out of BigCrush's suite of 160 tests, the Threefry and Tyche generators passed all 160 tests, and Philox and Squares passed 159. It is worth noting that during repeated trials with multiple global seeds, occasional failures are expected. This is not unique to OpenRAND; the authors of



(a) Time taken by OpenRAND generators versus baselines (std::mt19937 and r123::philox) to produce specified stream lengths on the host.



(b) Wall time for various libraries executing the Brownian Dynamics benchmark on different GPUs, using the Philox generator [11] in each library.

Fig. 4. Performance of OpenRAND on host and device respectively.

cuRAND[4] noted similar failures. For an exhaustive breakdown of our statistical results, we direct readers to our documentation.[5] To the best of our knowledge, this is the first time Tyche [20] and Squares [21] generators have undergone correlation tests for parallel streams.

6. Impact

Random number generation plays a fundamental role in the efficiency and reliability of larger software systems across fields such as stochastic simulations, machine learning, and computer graphics. Ideally, there would exist a good off-the-shelf RNG library that could be used in various contexts, including multi-threaded/multi-process applications, without introducing excessive boilerplate code, unnecessary complexity, or restrictions on applicable architecture. The existing software landscape is, however, fraught with challenges. Some good options expose low-level algorithmic and implementation details (e.g., Random123, cuRAND), leading to increased complexity; others are intrinsically bound to specific hardware (like cuRAND) or software platforms (such as rocRAND, OneAPI MKL). Several once-popular, platform-agnostic alternatives are now abandonware (clRNG, RandomCL), and even universal options, like the C++ Standard library, prove ill-suited for GPGPU programs. This landscape has led many open-source platforms to either layer atop a low-level library, like HOOMD-Blue's use of Random123, or to write custom random generators—as seen in Tensorflow, Pytorch, VTK, Jax, Alpaka, Kokkos, and others—sometimes without the benefit of thorough statistical validation.

In light of the deficiencies of the existing software landscape, we designed OpenRAND to be that off-the-shelf solution, a simplistic library that developers could pull into their projects with ease and use to generate random numbers in parallel in as little as two lines of code. Meeting the criteria for such a library, OpenRAND is trustworthy—validated for statistical robustness for single and parallel streams across CPU and GPU platforms (including standard C++, CUDA, and HIP). It is as performant, if not better, than existing libraries such as cuRAND, Kokkos, Random123, and the C++ Standard library while maintaining portability across different architectures. Most importantly, it has a clean, intuitive API that complies with the C++ standard's random number engine interface, making its use familiar and compatible with existing libraries. As such, OpenRAND is uniquely positioned as a solution to elevate various projects by simplifying development, boosting performance, and enhancing portability while ensuring statistical validity and reproducibility for single and parallel streams.

7. Conclusion

To summarize, while the realm of random number generation presents numerous software options riddled with deficiencies—from being difficult and error-prone to use to being tightly bound to specific architectures—OpenRAND provides an off-the-shelf solution with a clean, intuitive API and competitive performance on CPUs and GPUs, without introducing boilerplate code, unnecessary complexity, or restrictions on applicable architecture. Its focus on counter-based generators was made strategically due to their low-memory footprint, flexibility with respect to RNG parallelization strategy, and avalanche property. Together, these features simplify the development of reproducible parallel code, avoiding the complications and redundancies faced by other libraries. Given its features, rigorous statistical validation, and performance benchmarks, OpenRAND has the potential to significantly aid developers in various scientific fields, ensuring that random number generation remains reliable, efficient, and easy to use.

CRediT authorship contribution statement

Shihab Shahriar Khan: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Methodology, Formal analysis, Conceptualization. **Bryce Palmer:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Investigation, Conceptualization. **Christopher Edelmaier:** Writing – review & editing, Validation, Investigation, Conceptualization. **Hasan Metin Aktulga:** Writing – review & editing, Validation, Supervision, Software, Project administration, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code for the library and the benchmarks have been linked in the paper.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to improve language and readability. The authors reviewed and edited the final content and take full responsibility for the content of the publication.

[4] https://docs.nvidia.com/cuda/curand/testing.html
[5] https://msu-sparta.github.io/OpenRAND/md_statistical_results.html

Acknowledgments

This material is based upon work supported by the National Science Foundation Office of Advanced Cyberinfrastructure, United States of America under Grant 2007181 and used resources provided by Michigan State University's High-Performance Computing Center, United States of America.

References

[1] Phillips CL, Anderson JA, Glotzer SC. Pseudo-random number generation for Brownian dynamics and dissipative particle dynamics simulations on GPU devices. J Comput Phys 2011;230(19):7191–201.
[2] Dura-Bernal S, Suter BA, Gleeson P, Cantarelli M, Quintana A, Rodriguez F, et al. NetPyNE, a tool for data-driven multiscale modeling of brain circuits. Elife 2019;8:e44494.
[3] L'Ecuyer P, Nadeau-Chamard O, Chen Y-F, Lebar J. Multiple streams with recurrence-based, counter-based, and splittable random number generators. In: 2021 winter simulation conference. WSC, 2021, p. 1–16. http://dx.doi.org/10.1109/WSC52266.2021.9715397.
[4] O'Neill ME. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. ACM Trans Math Software 2014.
[5] De Matteis A, Pagnutti S. Parallelization of random number generators and long-range correlations. Numer Math 1988;53:595–608.
[6] L'Ecuyer P, Munger D, Oreshkin B, Simard R. Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. Math Comput Simulation 2017;135:3–17.
[7] Fog A. Pseudo-random number generators for vector processors and multicore processors. J Modern Appl Stat Methods: JMASM 2015;308–34. http://dx.doi.org/10.22237/jmasm/1430454120.
[8] Matthews R. On the derivation of a "chaotic" encryption algorithm. Cryptologia 1989;13(1):29–42.
[9] Tutueva AV, Nepomuceno EG, Karimov AI, Andreev VS, Butusov DN. Adaptive chaotic maps and their application to pseudo-random numbers generation. Chaos Solitons Fractals 2020;133:109615.
[10] Matsumoto M, Nishimura T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simul (TOMACS) 1998;8(1):3–30.
[11] Salmon JK, Moraes MA, Dror RO, Shaw DE. Parallel random numbers: as easy as 1, 2, 3. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis. 2011, p. 1–12.
[12] Saito M, Matsumoto M. Variants of Mersenne twister suitable for graphic processors. ACM Trans Math Softw 2013;39(2):1–20.
[13] Ciglarič T, Češnovar R, Štrumbelj E. An OpenCL library for parallel random number generators. J Supercomput 2019;75:3866–81.
[14] L'Ecuyer P, Munger D, Kemerchou N. clRNG: A random number API with multiple streams for OpenCL. 2015, URL https://api.semanticscholar.org/CorpusID:56334678.
[15] Trott CR, Lebrun-Grandié D, Arndt D, Ciesko J, Dang V, Ellingwood N, et al. Kokkos 3: Programming model extensions for the exascale era. IEEE Trans Parallel Distrib Syst 2022;33(4):805–17. http://dx.doi.org/10.1109/TPDS.2021.3097283.
[16] Lemire D, O'Neill ME. Xorshift1024*, Xorshift1024+, Xorshift128+ and Xoroshiro128+ fail statistical tests for linearity. 2018, CoRR abs/1810.05313, arXiv:1810.05313.
[17] Bernstein DJ, et al. ChaCha, a variant of Salsa20. In: Workshop record of SASC, vol. 8, (no. 1):Citeseer; 2008, p. 3–5.
[18] Zafar F, Olano M, Curtis A. GPU random numbers via the tiny encryption algorithm. In: Proceedings of the conference on high performance graphics. 2010, p. 133–41.
[19] Anderson JA, Glaser J, Glotzer SC. HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations. Comput Mater Sci 2020;173:109363.
[20] Neves S, Araujo F. Fast and small nonlinear pseudorandom number generators for computer simulation. In: Parallel processing and applied mathematics: 9th international conference, PPAM 2011, torun, Poland, September 11-14, 2011. revised selected papers, part i 9. Springer; 2012, p. 92–101.
[21] Widynski B. Squares: A fast counter-based RNG. 2020, arXiv preprint arXiv:2004.06278.
[22] L'Ecuyer P, Simard R. TestU01: A C library for empirical testing of random number generators. ACM Trans Math Softw 2007;33(4):1–40.
[23] Doty-Humphrey C. Practically random: C++ library of statistical tests for RNGs. 2010, https://sourceforge.net/projects/pracrand.