EPJ Web of Conferences **295**, 11001 (2024)
*CHEP 2023*
https://doi.org/10.1051/epjconf/202429511001

# A new portable random number generator wrapper library

*Tianle* Wang[1,*], *Mohammad* Atif[1], *Zhihua* Dong[1], *Charles* Leggett[2], and *Meifeng* Lin[1]

[1]Brookhaven National Laboratory, Upton, NY 11973, USA
[2]Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA

**Abstract.** Random number generator is an important component of many scientific projects. Many projects are written using programming models (like OpenMP and SYCL) to target different architectures. However, some programming models do not provide a random number generator. In this work, we introduce our random number generator wrapper. It is a header-only library that supports three distributions of random numbers: uniform, normal, and poisson. On the GPU backend, it wraps the cuRAND and rocRAND library, and supports various random number engines. It also wraps random123, a counter-based random number generator, on both CPU and GPU. With this library, we can generate random numbers with a few lines of code and target both GPU and multi-thread CPU with the same code. We also investigate the performance and scalability of this wrapper on different architectures with different engines and the number of cores.

## 1 Introduction

Computing is a central component of many aspects of experimental high energy physics (HEP), including but not limited to event reconstruction and detector simulation. The HEP software development was used to target traditional CPU architectures since CPUs used to be the predominant computational resource available. However, the capabilities of modern high-performance computing facilities have significantly expanded. They usually offer heterogeneous architecture with host multi-core CPUs and compute accelerators such as general-purpose Graphics Processing Units (GPU). HEP software needs to be rewritten to use those accelerators efficiently by running some of its components on them. However, adapting or rewriting older software to run correctly and efficiently on each new accelerator is time-consuming. To alleviate this difficulty, portable programming models are explored that allow users to write the software with a single API while keeping good performance over various CPU and GPU architectures.

During the implementation of HEP software, random number generation (RNG) plays an important role in many algorithms and applications (e.g., Wire-Cell-Gen[1] and FastCaloSim[2, 3]). However, many widely used programming models, including SYCL[4], OpenMP[5], and std::par, do not offer a general API for RNG for all architecture. This creates an unnecessary burden for developers since we must develop a wrapper that calls different vendor libraries on different architectures, a process that must be repeated for every RNG-related code. To address this issue, in this work, we propose a portable RNG wrapper compatible with multiple architectures that are widely used these days, including multi-threaded

*e-mail: twang3@bnl.gov

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

CPUs and NVIDIA/AMD GPUs (the Intel GPU support will come in the future), and multiple programming models commonly used, including but not limited to, OpenMP, SYCL and std::par (Kokkos can also use this library, but it also has its native API to do RNG). In Section 2, we explain our methodology and the implementing details of this library. In Section 3, we explain how to use this library in applications targeting different architectures. In Section 4, we document the performance evaluation of this library, including the choice of different RNG engines and scaling behavior. Finally, we summarize our experiences in Section 5.

## 2 Methodology and implementation detail

In many HEP software applications, it is a common practice to generate a large sequence of random numbers (usually larger than $O(10^4)$) and store them for subsequent use. This is because most RNG performs well in batch generation. Because of that, our library APIs allow users to generate a sequence of random numbers. Although different programming models (such as SYCL, std::par, and OpenMP) have their native sequential data containers, these containers fundamentally serve as wrappers for the C-array. All those containers provide API that enables interaction between the container and the underlying C-array. Because of that, to ensure portability, our API saves the random number outputs in a 1D array and returns the corresponding pointer, and it is the users' responsibility to create and initialize the data container in their chosen programming model with that pointer. Also, we implement this library as a header-only library: once the vendor library (for example, cuRAND on NVIDIA GPU) is installed, users can easily deploy our library by simply including the header.

Currently, we have developed wrappers for four architectures, namely:

- For single-thread CPU use case (basic), we wrap the mt19937 implemented in the C++ standard library. This serves as a backup if users want to use that RNG for a reproducibility check of their code with the single-worker (serial) pattern.
- For multi-threads CPU use case, we implement a set of APIs based on the random123 library[6] using OpenMP. The random123 library uses a counter-based random number generator engine, allowing massive parallelization with relatively small memory usage. This will usually not harm the portability since most compilers for programming models support basic OpenMP syntax on the CPU.
- For NVIDIA GPU, we wrap the host API of the cuRAND library. We support most random number engines and their settings. However, we currently do not support stream setting, which could interfere with most of the queue-based programming models (e.g., SYCL).
- For AMD GPU, we wrap the host API of the rocRAND library. We support most random number engines and their settings. Similar to NVIDIA GPU, we currently do not support stream setting.

On all those architectures, We support generating five kinds of random numbers:

- Uniform distribution of int type between 0 and INT_MAX.
- Uniform distribution of float/double type with a given range.
- Gaussian distribution of float/double type with given mean/std.

On different architectures, the list of supported RNG engines is different. This is summarized in Figure. 1. Here green cell represents the recommended RNG engines for that architecture as it gives the best performance, red cell represents that this engine is not supported on that architecture, yellow cell represents that this engine is supported but not optimal.

 https://doi.org/10.1051/epjconf/202429511001

| | Basic | Multi-threads CPU | NVIDIA GPU | AMD GPU |
|---|---|---|---|---|
| philox | red | green | green | green |
| xorwow | red | red | yellow | yellow |
| mrg32k3a | red | red | yellow | yellow |
| sobol32 | red | red | yellow | yellow |
| sobol64 | red | red | yellow | yellow |
| mtgp32 | red | red | yellow | yellow |
| mt19937 | green | red | yellow | red |

**Figure 1.** The hardware architecture support of RNG engines. green cell indicates the recommended engine, yellow cell indicates support but not optimal, and red cell indicates no support.

## 3 Usage

In this section, we show the reader how to use the library in their applications.

### 3.1 Usage Example

As described in Section 2, this RNG library will generate a uniform set of APIs for most programming models on four different hardware architectures. The user will first need to set up the random number generator with some engine. If the engine is not specified, the optimal one on that hardware architecture will be selected. The user can then call get_rng_<distribution_type>_<data_type> to generate a sequence of random numbers. Notice it is the user's responsibility to expose the raw pointer of the corresponding programming model's data container to that API. Examples are shown in Table 3.1 for OpenMP (use use_device_ptr construct), SYCL (call data() method of the SYCL container), and std::par (don't need to do anything).

```
auto rng = setup_rng(engine_name, seed, ...); //setup rng engine
-----
OpenMP:
#pragma omp target data use_device_ptr(m_normals)
get_rng_normal_double(m_normals, size, 0.0, 1.0, rng);
-----
SYCL:
auto m_normals = normals.data();
//Do not need to submit it using the queue
get_rng_normal_double(m_normals, size, 0.0, 1.0, rng);
-----
std::par:
double* m_normals = (double*)malloc(sizeof(double) * size);
get_rng_normal_double(m_normals, size, 0.0, 1.0, rng);
```

**Table 1.** An example of code snippet to show how to use this RNG library in different programming models (OpenMP, SYCL, and std::par) to generate a sequence of random numbers of normal distribution with double type.

### 3.2 Compile issue

To compile the code that uses this library, we also need to tell which hardware architecture we are targeting. Although some programming models can choose to target one specific

hardware architecture when compiling with some compiling options, they also allow users to compile once and generate a single binary that can target various hardware architectures. This library, however, can only target a single hardware architecture with a single compile, and the user needs to add an extra Dflag that represents the architecture the RNG will be run on. For example, we use the following Dflags to target the four hardware architectures mentioned in Section 2:

- -DARCH_CUDA for NVIDIA GPU
- -DARCH_HIP for AMD GPU
- -DUSE_RANDOM123 for multi-thread CPU
- If no flag is specified, use the default basic implementation with single thread on CPU

## 4 Performance Evaluations

For consistency of performance evaluations, we used the same workstation (lambda1) at Brookhaven National Laboratory which has an NVIDIA V100 GPU, an AMD Raedon Pro VII GPU, and an AMD 24-core Ryzen Threadripper 3960X CPU with 48 hyper threads.

### 4.1 Performance comparison

To compare the performance of each of the RNG engines in generating different types of random numbers on different architectures, we measured the average wall time of random number generation over 50 runs for each combination of random number type and engine. Here $4\times10^8$ random numbers are generated for each test. The results are shown in Figures 4.1 for NVIDIA GPU (left) and AMD GPU (right). For single-core and multi-cores CPU, we didn't compare the performance since only a single RNG engine is supported. We can see that the philox engine gives us the best performance in almost all cases, and different RNG engines have similar performance, except mt19937, whose performance is about 3-10 times worse than other engines.

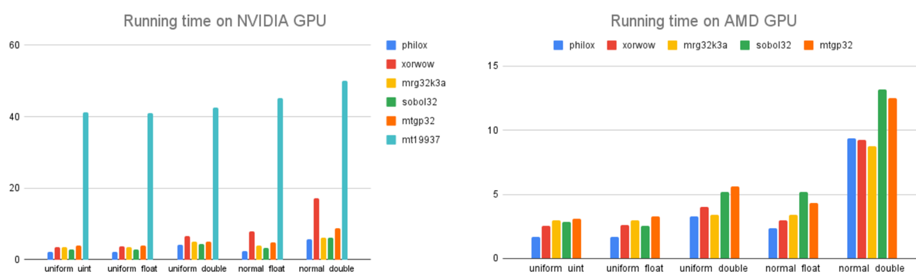


**Figure 2.** Running time of using different RNG engines to generate five different types of $4 \times 10^8$ random numbers on NVIDIA GPU (left) and AMD GPU(right).

### 4.2 Scaling behavior

We also look at the scalability of our library. In Figure 4.2, we measure the average wall time of random number generation over 50 runs using the philox RNG engine with different numbers of random numbers generated. This is done for all five different combinations of

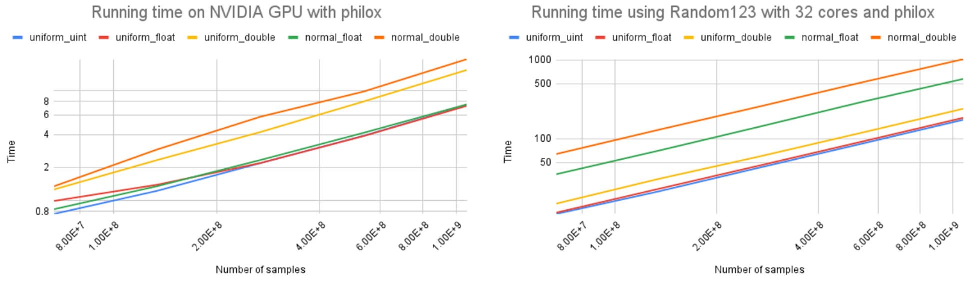


**Figure 3.** Scaling of running time using philox engines of generating different types of random numbers on NVIDIA GPU (left) and 32-cores CPU (right).

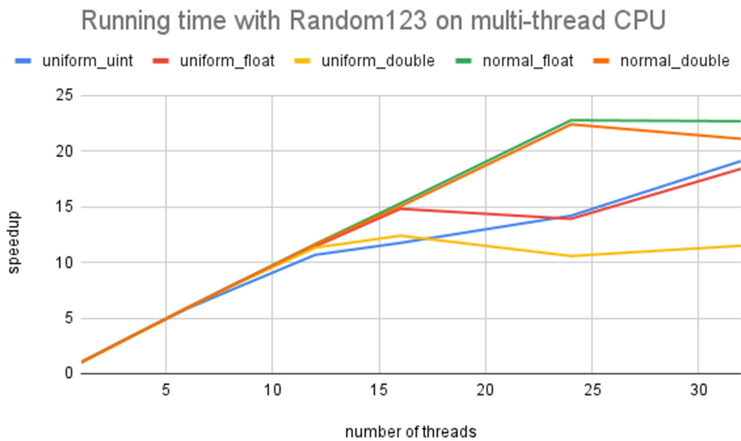


**Figure 4.** The strong scaling behavior using philox engines to generate different types of random numbers on multi-thread CPU.

random number types on NVIDIA GPU (left) and 32-cores CPU (right). The behavior on AMD GPU is almost identical to that on NVIDIA GPU, so we skip it here. We can see that the slopes on the log-log plot for all random number types are almost one, suggesting that the wall time is almost proportional to the size of the random numbers we generated.

In Figure 4, we show the average wall time on multi-thread CPU with different numbers of threads. When the number of threads is less than 12, we see almost linear strong scalability behavior. As we further increase the number of threads, the linear scaling behavior starts to break, and we see almost no performance improvement when the number of threads exceeds 24. This is because, although our machine has 48 hyper threads, it only has 24 physical cores, and for a computationally intense operation like RNG, introducing more than one thread per core will not help to improve the performance.

## 5 Summary and Outlook

We successfully implemented the header-only random number generation library that can be used with various programming models, including but not limited to OpenMP, SYCL, and std::par. This library can target four different hardware architectures: Single-core CPU, multi-core CPU, NVIDIA GPU, and AMD GPU. It also offers multiple choices of RNG engines for the user. We have shown that it can produce relatively good computational performance with good scalability. We are in the process of 1). adding Intel GPU as another supported hardware architecture; 2). adding support for asynchronous running with stream on GPU that is consistent with different programming models; 3). adding support for output memory layout that is more complicated than the simple 1D array.

## Acknowledgments

This work was supported by the U.S. Department of Energy, Office of Science, Office of High Energy Physics, High Energy Physics Center for Computational Excellence (HEP-CCE) under B&R KA2401045. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. We gratefully acknowledge the support of the Wire-Cell team of the Electronic Detector Group in the Physics department and the Scientific Data and Computing Center of Brookhaven National Laboratory, which is supported by the U.S. Department of Energy under Contract No. DE-SC0012704.

## References

[1] Lin, Meifeng, et al. "Portable Programming Model Exploration for LArTPC Simulation in a Heterogeneous Computing Environment: OpenMP vs. SYCL." arXiv preprint arXiv:2304.01841 (2023).

[2] The ATLAS Collaboration, The ATLAS Simulation Infrastructure, The European Physical Journal, **70**, 823-874 (2010)

[3] The ATLAS Collaboration, The new Fast Calorimeter Simulation in ATLAS, Tech. Rep. **ATL-SOFT-PUB-2018-002**, (2018)

[4] Khronos SYCL working group. 2020. SYCL 1.2.1 specification. Standard. Khronos Group, Inc, Beaverton, OR, USA.

[5] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J. (2001). Parallel programming in OpenMP. Morgan kaufmann.

[6] Salmon, John K., et al. "Parallel random numbers: as easy as 1, 2, 3." Proceedings of 2011 international conference for high performance computing, networking, storage and analysis. 2011. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf