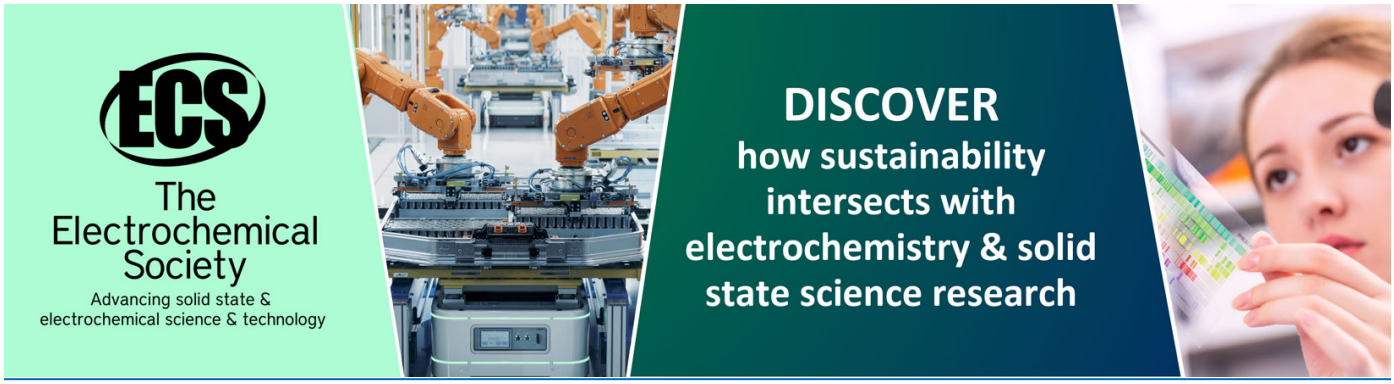
Vectorization of random number generation and reproducibility of concurrent particle transport simulation



Vectorization of random number generation and reproducibility of concurrent particle transport simulation

S Y Jun[1], P Canal[1], J Apostolakis[2], A Gheata[2], L Moneta[2]

[1]Fermi National Accelerator Laboratory†, MS234, P.O. Box 500, Batavia, IL, 60510, USA
[2]CERN, EP Department, Geneva, Switzerland

E-mail: syjun@fnal.gov, john.apostolakis@cern.ch

Abstract.

Efficient random number generation with high quality statistical properties and exact reproducibility of Monte Carlo simulations are important requirements in many areas of computational science. VecRNG is a package providing pseudo-random number generation (pRNG) in the context of a new library VecMath. This library bundles up several general-purpose mathematical utilities, data structures, and algorithms having both SIMD and SIMT (GPUs) support based on VecCore. Several state-of-the-art RNG algorithms are implemented as kernels supporting parallel generation of random numbers in scalar, vector, and Cuda workflows. In this report, we will present design considerations, implementation details, and computing performance of parallel pRNG engines on both CPU and GPU. Reproducibility of propagating multiple particles in parallel for HEP event simulation is demonstrated, using GeantV-based examples, for both sequential and fine-grain track-level concurrent simulation workflows. Strategies for efficient uses of vectorized pRNG and non-overlapping streams of random number sequences in concurrent computing environments is discussed as well.

1. Introduction

The stochastic nature of many physical systems is often well modeled by Monte Carlo techniques, which require pseudorandom number generators (pRNGs) with good statistical properties and generally the ability to create independent sub-sequences. Ever-increasing computing capacity is being provided by systems with modern multi-core CPU processors and especially by HPC systems equipped with GPUs or other accelerators. Owing to this, the precision and speed of a large scale scientific simulation of complex systems have been improved significantly. To enable these, new types of pRNGs with high statistical quality and a long period have been recently developed [1, 2, 3]. The recent evolution of hardware architectures towards wider vector pipelines, GPUs/accelerators and many-threads opens new opportunities for concurrent simulation models taking advantage of both SIMD and SIMT (GPU). These require pRNGs suitable for massively parallel and scalable computing [4].

Despite the fact that there are many parallel pRNGs developed for specific hardware architectures, portable libraries of random number services which can be used across different architectures and in hybrid computing models are not commonly available. It is pertinent to use common pRNGs not only for the purpose of validation, but also for the full reproducibility of

simulation chains across different platforms, discussed in detail in Section 4. In this paper, we report on design considerations and an initial implementation of a small set of pRNG algorithms portable for SIMD and SIMT which rely on a common kernel. The reproducibility of propagating multiple particles in parallel for high energy physics (HEP) event simulation is also demonstrated.

2. Vectorization of pseudo random number generation

VecRNG is a part of VecMath which is a collection of vectorized math algorithms and utility functions for HEP applications, based on the VecCore library [5]. It provides parallel pRNGs implementations for both SIMD and SIMT workflows via architecture-independent common kernels. In this section, we describe the design and implementation of vectorized algorithms in the context of VecRNG.

2.1. Choice of Generators

We sought generators that meet strict quality requirements, belonging to families of generators which have been examined in depth [6], or have evidence from ergodic theory of exceptional decorrelation properties [3]. All must pass major crush-resistant tests such as DIEHARD [7] and BigCrush of TestU01 [8]. In addition we took into consideration constraints in size of the state and performance: 1) a very long period ($\geq 2^{200}$), obtained from a small state (in memory), 2) fast implementations and repeatability in sequence on the same hardware configuration, 3) efficient ways of splitting the sequence into long disjoint streams.

For the first phase of implementation, we selected the following representative generators from major classes of pRNG: MRG32k3a [1], Random123 [2], and MIXMAX [3]. Table 1 shows key properties of the generators selected for vectorization.

Table 1. The list of generators under VecMath and some of their properties.

Generator	class	Scalar State	Memory	Period	Stream
MRG32k3a	Algorithm-based	6 doubles	48 bytes	2^{191}	2^{64}
Threefry(4x32)	Cryptographic-based	12 32-bit int	48 bytes	2^{256}	2^{128}
Philox(4x32)	Cryptographic-based	10 32-bit int	40 bytes	2^{192}	2^{64}
MIXMAX(N=17)	Anosov C-system	17 64-bit int	68 bytes	10^{294}	10^{250}

2.2. Design Consideration

VecRNG is designed to support concurrent computing models as well as SIMD and SIMT (GPU) with common kernels using backends provided by VecCore. The primary purpose of the use of backends is to provide access to the full performance of vector hardware, while hiding both hardware-specific details (i.e. width of vector registers) and the complexity of low-level instruction sets behind abstraction layers. In this way, generic kernels are highly portable across different hardware architectures or dependencies to external libraries. Today a vector backend can use either the Vc library [9] or UME::SIMD [10] for explicit SIMD vectorization. The CUDA and scalar backends share standard types; in addition there are cuda-specific extensions. Detailed descriptions of VecCore and its backends can be found elsewhere [11, 12].

Another design choice was the exclusive use of static polymorphism, motivated by performance considerations. Every concrete generator inherits through the CRTP (curiously recurring template pattern) from the VecRNG base class, which defines mandatory methods and common interfaces.

Last but not least, the implementation of VecRNG is header only and provides a minimal set of member methods. This approach allows more flexibility to higher level interfaces for specific computing applications, but minimizes the overhead compilation time.

2.3. Implementation Details

VecRNG is the base class for the static polymorphism and has only one protected data member (*State_t *fState*) where *State_t* is defined in each concrete generator and provided through RNG_traits. An example for MRG32k3a is:

```
template <typename Backend>
struct RNG_traits<MRG32k3a<Backend> > {
  struct State { typename Backend::Double_v fCg[MRG::vsize]; };
  using State_t = State;
};
```

The essential components of VecRNG interfaces are `Uniform<Backend>()` and `Uniform<Backend>(State_t& s)` which generate the backend type of double precision u.i.i.d (uniformly independent and identically distributed) in [0,1), and update the internal *fState* and the given state s, respectively. The latter shown below is originally introduced for GPU, but also is useful for reproducibility which will be discussed in the next section.

```
template <typename Backend>
VECCORE_ATT_HOST_DEVICE
typename Backend::Double_v Uniform(State_t& state) {
  return static_cast<DerivedT *>(this)->template Kernel<Backend>(state);
};
```

Each derived pRNG class is only responsible for implementing the generic Kernel method and its own auxiliary member functions,

One of associated requirements for each generator in VecRNG is to provide an efficient skip-ahead algorithm, $s_{n+p} = f_p(s_n)$ (i.e., advancing a state, s_n, by p-sequences where p is the unit of the stream length or an arbitrary number) in order to assign disjointed multiple streams for different tasks. For an example, the mandatory method, `Initialize(long n)`, moves the random state at the beginning of the given n^{th} stream. Each generator supports both scalar and vector backends with a common kernel. Random123 has an extremely efficient stream assignment without any additional cost since the key serves as the stream index while MRG32k3a uses transition matrices (A) which recursively evaluate (A^s mod m) using the binary decomposition of s. The vector backend uses N(=SIMD length) consecutive substreams and also supports the scalar return-type which corresponds to the first lane of the vector return-type.

For SIMT (GPU) applications, multiple independent streams are assigned to threads of all blocks with `Initialize(State_t *states, unsigned int n)`, where n is the multiplication of the number of blocks times the number of threads per block and `state` is a pre-allocated array of `State_t` of which the size is n. Then the state is used for each thread of a block to generate a random number and update its state using `Uniform<Scalar>(state[tid])` where `tid` is the thread index in CUDA kernel (i.e, `tid = threadIdx.x + blockIdx.x * blockDim.x`). This approach can be also used for applications on the host side which require statistically disjointed streams for n-parallel tasks.

Besides the Uniform method, some commonly used random variates are also provided. Implementation details of probability distribution functions are discussed separately [13].

3. Performance

Three important metrics to evaluate performance of pRNGs are speed (CPU), memory requirement, and the quality of random numbers. As pRNGs selected for this report are already

satisfactory for two latter criteria, we concentrate on the relative performance between different backends. To have a minimal validation check for quality, the fluctuation of the sum within each sample is required to meet the condition $|\sum^{N/n_v} \boldsymbol{u} - N/2| < \sqrt{N/12}$ where $\boldsymbol{u}$ is the vector of random numbers and n_v is the size of vector lanes for *double* (e.g., 4 for AVX).

Table 2 shows preliminary performance results of implemented pRNGs on Intel® Xeon(R) E5-2620 CPU (12 cores @ 2.00GHz). Performance and quality of the Intel MKL/VSL [14] random number generation depend on the size of output array (N) - we used N = 32 for this comparison.

Table 2. Performance of generators implemented in VecRNG on Intel® Xeon(R) E5-2620 CPU (Sandy Bridge) is compared to std::rand() and the Intel MKL/VSL library. The time [ms] to generate $N = 10^7$ output doubles was averaged over 200 measurements.

Generator	std::rand()	MRG32k3a	Threefry	Philox
Scalar	139.98 ± 0.06	209.25 ± 0.07	129.65 ± 0.06	100.78 ± 0.04
Vector(SSE)		123.58 ± 0.06	123.47 ± 1.19	225.82 ± 0.46
Vector(AVX)		110.97 ± 0.06	82.88 ± 0.15	141.54 ± 0.10
Intel MKL/VSL		145.50 ± 0.06	N/A	N/A

The performance of the CUDA backend and the Curand library [15] (cuda 8.0, arch=sm_3.5) in Table 3 were measured using the kernel configuration with 26 blocks x 192 threads on NVidia Tesla K20M GPU (2496 CUDA cores @ 0.71GHz). The Word size (W) and round (R) used for Random123 were W4x32_R20 for Threefry and W4x32_R10 for Philox, respectively.

Philox uses the Advance Randomization System (ARS) which iterates bijection with rounds of the Feistel function and a couple of XOR operations. Poor vector performance of Philox is due to a lack of a native vector operation for conversion between 64 and 32 bit (SIMD) integers which forces it to use scalar operations. Performance of Random123 (Threefry and Philox) on GPU will be further optimized.

4. Reproducibility of simulation in parallel computing

HEP experiments have a goal for the reproducibility for detector simulation: simulations with the same initial configuration (primary particles and pRNG choice and seed) must give the same results. This must hold true even if different choices are made during a run, e.g. using vector kernels for a set of physics processes of selected tracks. A key practical reason is the need to reproduce and debug problems which occur during the simulation of a particular event or initial particle. In addition, the reliability of a simulation which cannot be repeated is more difficult to assess.

In the Geant Vector Prototype (GeantV) [16], baskets of tracks undergoing the same interaction are accumulated to enable computations on vectors of track properties. The aim is to

Table 3. Performance of VecRNG generators compared with the NVidia Curand library on NVidia Kepler K20M GPU (Tesla). The time [ms] to generate $N = 10^7$ output doubles was averaged over 200 measurements.

Generator	MRG32k3a	Threefry	Philox
Cuda backend	2.03 ± 0.03	10.22 ± 0.02	10.17 ± 0.01
NVidia Curand	2.05 ± 0.02	N/A	1.92 ± 0.01

use vectors for the bulk of the computational work. The remaining tail of tracks is treated with serial (non-vectorized) code, using the same algorithms as the vector code. Multi-threading is used to gather larger populations of tracks having similar properties and enable (wider) vectors, targeting better use of vector code and higher performance. Due to the out-of-order execution in multi-threading, different tracks are collected in baskets in each run. In addition, a different set of remaining 'unbasketised' tracks is run in scalar code in each run, in particular during phases of basket starvation and the ramping down of the simulation. So a particular algorithm must obtain the same pRNG output value (variate) for a track, whether it is processed as part of a vector in a basket of tracks (in 'vector' mode) or as a single track using the non-vectorized code (in 'scalar' mode).

To obtain the same results for a track's physics interactions (or other operations), the same sequence of output values of a pRNG is needed. This is accomplished by associating a single state of a pRNG with each track. When a new track is created either as a primary particle or in an interaction, a deterministically-defined new state of the pRNG must be generated and associated to it. This idea, called 'pseudo-random' trees, was first proposed in the 1980s in a particle transport application [17]. A first implementation was also created using linear congruential generators. Applications in other parallel and branching computations have been proposed since - see the recent review of Schaathun[18] for an overview and an evaluation of methods proposed. One such method, the pedigree method, for constructing seeds was developed by Leiserson *et al.* [19] exploiting deterministic parallel computations written in Cilk. This method was demonstrated in particle transport simulation [20] using Geant4 [21] as a testbed.

We demonstrate how such splittable/tree pRNGs can be used in practice within the constraints of a particle transport program which mixes vector and 'scalar' (non-vector) code which can be run in either a multi-threaded or serial implementation. We seek also to measure the overhead, compared to simulations which do not use these methods, and, as a result, do not reproduce the same results between runs. We discuss a number of considerations and key aspects of the implementation.

The approach adopted for GeantV depends on two parts: First, an initial seed for the scalar mode or a set of seeds for the vector mode are assigned by the run number or other units of parallel tasks. Second, a unique stream index is determined for each track. The stream index for the primary track consists of high precision bits set by the event number and low precision bits by the track index. For the secondary (or daughter) track, the stream index is generated in a collision resistant way using the current state of the pRNG carried by the (mother) track that undergoes an interaction.

To enable vector code for physics processes, a vector pRNG must be created in order to generate the output in each vector lane of the PRNG corresponding to that track. In our design, this is the role of an instance of a proxy class, which acts as a vector pRNG. The proxy both provides all the expected outputs in each vector lane (as through from the pRNG of its track) and advances the state of each track's pRNG accordingly. Our first implementation of a proxy class gathers the contents of the scalar pRNGs into an instance of the corresponding VecRng class (e.g. gathering `MRG32k3a<double>` into `MRG32k3a<Double_v>`). The proxy instance is reusable, by explicitly attaching and detaching the set of track pRNG states. We have tested this capability using a limited set of GeantV physics processes, including in particular bremsstrahlung, ionisation, and Compton scattering, which undergo a self-contained $e^- - \gamma$ cascade process. The pRNG used in the tests is ThreeFry from the Random123 package. This counter based generator was chosen because the stream is easily split into, the size of the state is moderate (128 bits), and the method of initialization from a seed is trivial.

We compare the number of tracks and steps of a simulation of 1000 events, each comprised of ten 10 GeV electrons impinging on a 50 layer lead and liquid argon calorimeter. The number of tracks and steps of scalar ('non-vector') and vector configurations, with either 1 or 4 threads

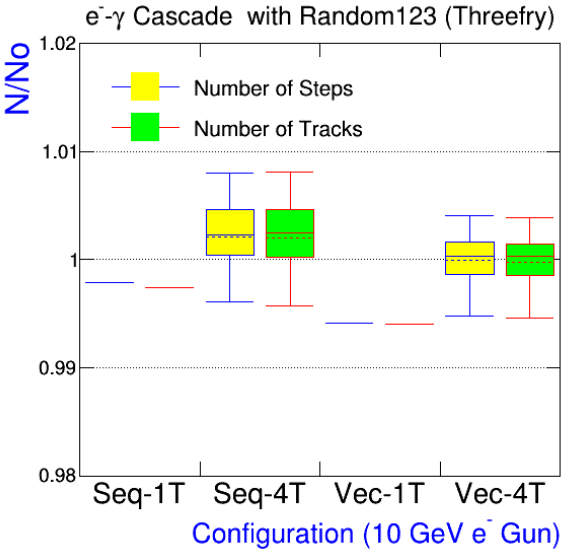


Figure 1. The ratio of the total number of tracks (steps) of the default (non-reproducible) mode normalized to the reproducible configuration of which the total number of tracks (steps) is N_o.

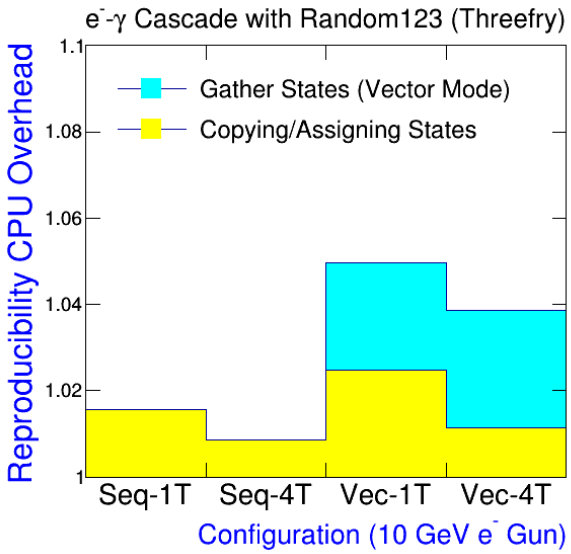


Figure 2. The overhead of the reproducibility in simulation (CPU) time for the strategy using gathering scalar states to a vector state for different configurations and splitting states visa versa.

each has been compared.

The simulation is run in two modes: the default mode in which a per-thread state of one serial pRNG and one vector pRNG are used in each thread, and the 'reproducible' mode in which our method is used. Using the values for the 'reproducible' mode run with 1 thread as baseline (Seq-1T), the ratios of the number of tracks and steps can be seen in Figure 1.

It is verified that the reproducible mode maintains the constant number of tracks and steps for all tested configurations, as required. In addition, those numbers are different from the single threaded mode for each of the 1-thread (Seq-1T) and vector 1-thread (Vec-1T) modes by $0.2 - 0.6\%$. In the multi-threaded mode, the number of tracks and steps fluctuates, as expected with averages (and variances) within one sigma of the values of the reproducible mode.

Reproducibility introduces an overhead in simulation time due to copying and assigning pRNG states during simulation workflows, gathering scalar states to a SIMD vector state or joining-splitting states for the proxy approach, and synchronizing the index of states in output (Random123 specific). Figure 2 shows an example of the CPU overhead as the fraction of Time(Reproducibility)/Time(Default Mode) using gathering scalar PRNG states to a vector state and splitting states visa versa. Another approach using the join-split method shows a similar (2-5%) performance degradation for the reproducibility mode.

Alternative proxy implementations are under development, including one that avoids the cost of copying the data. This is of most interest for the cases in which the average number of variates required is small and/or the PRNG state is large.

5. Conclusion

We implemented common kernels of pseudorandom number generation for SIMD and SIMT architecture using VecCore. We also demonstrated reproducibility of propagating multiple particles in parallel for HEP event simulation with concurrent workflows.

Acknowledgments

† Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

References

[1] P. L'Ecuyer, R. Simard, E.J. Chen, W.D. Kelton, An object-oriented random number package with many long streams and substreams, Operations Research 50 (2002) 1073-1075
[2] J.K. Salmon, M.A. Moraes, R.O. Dror, D.E. Shaw, Parallel random numbers: as easy as 1, 2, 3, International Conference for High Performance Computing, Networking, Storage and Analysis, ACM (2011) pp. 16:1-16:12
[3] K. Savvidy and G. Savvidy, MIXMAX Random Number Generator, arXiv:1510.06274v3 (2011), K. Savvidy 2015, The MIXMAX random number generator, Comp. Phys. Comm. , **196**, 161 and arXiv:1403.5355v2 (2014).
[4] P. L'Ecuyer, D. Munger, B. Oreshkin, R. Simard, "Random numbers for parallel computers: Requirement and methods, with emphasis on GPUs", *Math. and Computers in Simulation* **135** 3-17 (2017).
[5] Amadio G *et al* 2018 *J. Phys.: Conf. Ser.* **1085** 032034
[6] P. L'Ecuyer "Combined Multiple Recursive Random Number Generators", Op. Research **44**, 816
[7] G. Marsaglia, DIEHARD: A batter of tests of randomness (1996) `http://stat.fsu.edu/ geo/diehard.html`
[8] P. L'Ecuyer, R. Simard, TestU01: A C Library for Empirical Testing of Random Number Generators ACM Transaction on Mathematical Software, Vol. 33, No.4, Article 22 (2007)
[9] Kretz M and Lindenstruth V 2011 Vc: A C++ library for explicit vectorization *Software: Practice and Experience*. Online at `http://dx.doi.org/10.1002/spe.1149`
[10] UME::SIMD A library for explicit simd vectorization. Online at `https://github.com/edanor/umesimd`
[11] Wenzel S 2014 Towards a high performance geometry library for particle-detector simulation *16th International workshop on Advanced Computing and Analysis Techniques in physics research (ACAT)*
[12] de Fine Licht J 2014 First experience with portable high-performance geometry code on GPU *GPU Computing in High Energy Physics 2014*
[13] Amaro O *et al.* 2019 Vectorization techniques for probability distribution function using VecCore, the 19^{th} ACAT conference
[14] Intel MKL/VSL library, Intel Parallel Studio 2016
[15] NVIDIA Curand library, http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf
[16] Amadio G *et al.* 2015 J. Phys.: Conf. Ser.664 072006
[17] Halton J H 1989 Pseudo-random trees: Multiple independent sequence generators for parallel and branching computations", *J. of Comp. Physics* **84** 1.
[18] Schaathun, H G 2015 Evaluation of Splittable Pseudo-Random Generators. J. of Functional Programming 25, e6.
[19] Leiserson C E 2012 *et al.* Deterministic Parallel Random-number Generation for Dynamic-multithreading Platforms, *SIGPLAN Not.*, **47**, 193.
[20] Savin D, Using Pseudo-Random Numbers Repeatably in a Fine-Grain Multithreaded Simulation https://sd57.github.io/g4dprng/gsocPreprint.html
[21] Allison J *et al.* 2016 Recent developments in Geant4 *Nucl. Instrum. Methods Phys. Res.* A **835** 186-225
[22] Ahn S *et al.* 2014 Geant4-MT: bringing multi-threading into Geant4 production *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)* 04213