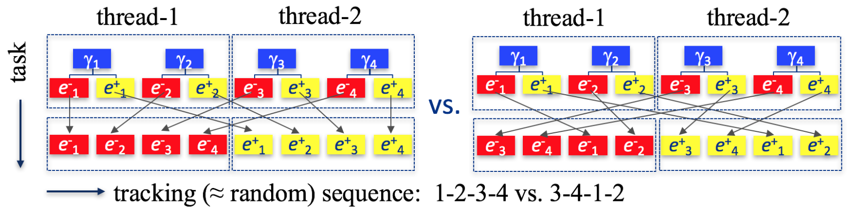
Vectorization of pRNG and reproducibility of concurrent particle transport simulation

Soon Yung Jun, Philippe Canal (Fermilab)
John Apostolakis, Andrei Gheata, Lorenzo Moneta (CERN)

ACAT19, Saas-Fee, Switzerland
March 11-15, 2019

Introduction

- Vectorization of pseudo random number generation (pRNG)
  - provides parallel pRNG for both SIMD and SIMT workflows
  - supports architecture-independent common kernels for pRNG
- Reproducibility of concurrent particle transport simulation
  - reproducibility is a stringent requirement for HEP simulation
  - the execution order of threads is fragile and unpredictable (example: pair-production and tracking particles by the type)



thread-1
thread-2
task
γ_1 γ_2 γ_3 γ_4
e^-_1 e^+_1 e^-_2 e^+_2 e^-_3 e^+_3 e^-_4 e^+_4
e^-_1 e^-_2 e^-_3 e^-_4 e^+_1 e^+_2 e^+_3 e^+_4
vs.
thread-1
thread-2
e^-_3 e^-_4 e^-_1 e^-_2 e^+_3 e^+_4 e^+_1 e^+_2
tracking (≈ random) sequence: 1-2-3-4 vs. 3-4-1-2

  - how to maintain absolute reproducibility of random number sequences for concurrent simulation tasks?

Contents

- Introduction
- Vectorization of pseudo random number generation
    - Choice of generators and implementation
    - Requirements and special features
    - Performance results
- Reproducibility of concurrent particle transport simulation
    - Reproducibility under concurrent tasks
    - Strategies for vector (SIMD) tasks
    - Verification and overhead
- Summary and Future Work

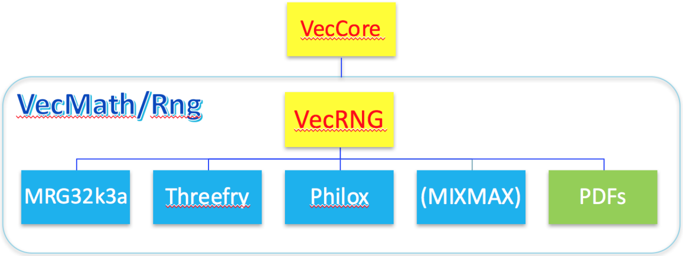
Choice of Generators: An initial list

- One of representative generators from major classes of pRNG
  - MRG32k3a [1]: Algorithm (Multiple Recursive Linear Congruent)
  - Random123 [2] : Counter based (Advance Randomization System)
  - MIXMAX [3]: Anosov C-system
- Meet general requirements for quality and performance
  - Long period ($\geq 2^{200}$), fast and with a small state
  - Repeatability in sequence on the same hardware configuration
  - Efficient ways of splitting the sequence into long disjoint streams
  - Crush-resistant: pass DIEHARD [4] and BigCrush of TestU01 [5]

Generator	Scalar State	Memory	Period	Stream
MRG32k3a	6 doubles	48 bytes	2^{191}	2^{64}
Threefry(4x32)	12 32-bit int	48 bytes	2^{256}	2^{128}
Philox(4x32)	10 32-bit int	40 bytes	2^{192}	2^{64}
MIXMAX(N=17)	17 32-bit int	68 bytes	10^{294}	$\sim \infty$

Implementation under VecMath

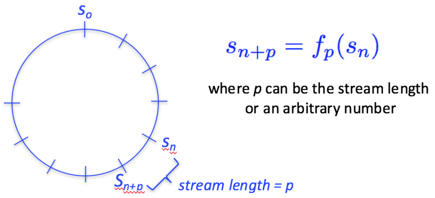
- VecMath: a collection of vectorized math utilities based on VecCore library [6] (https://github.com/root-project/vecmath)
- VecRNG under VecMath/Rng:



  - header only implementation with the static polymorphism
  - support SIMD and GPU with a common kernel using VecCore
  - support parallel (or concurrent) computing models

Requirements: Multiple Streams and Backends

- Multiple streams and an efficient skip-ahead algorithm (f_p): i.e., advancing a state by p-steps (sequences) to fully control parallel or multiple tasks independently



  - provide multiple streams for different tasks, which are potentially parallel (such as process id, task numbers, the mpi rank and etc.)

- Support both scalar and vector backend with a common kernel

```
rngEngine<Backend> rng(streamID);
Backend::Double_v rv = rng->Uniform<Backend>();
```

Special Features

- Vector backend uses N(=SIMD length) consecutive substreams and supports both vector and scalar return-type, for an example,

```
rngEngine<VectorBackend> rng();
double_v rv = rng->Uniform<VectorBackend>();
double   rs = rng->Uniform<ScalarBackend>();
```

- Working with states
  - The state is decoupled from the execution engine: passed as input for generation, but also updated in the process
  - an example use for GPU

```
State_t * states = ... CPU/GPU allocation ...
rngEngine<ScalarBackend> rng();
rng->Initialize(states, ntid);   // states[ntid]
double rs = rng->Uniform<ScalarBackend>(&states[tid]);
```

Preliminary Performance: AVX - Vc [7] Backend

- The average CPU time [ms] for generating 10M random numbers
  - std::rand() = (93.24 ± 0.03) ms

Generator	MRG32k3a	Threefry	Philox
Scalar	137.24 ± 0.05	74.13 ± 0.08	54.59 ± 0.03
Vector(AVX)	57.59 ± 0.02	43.06 ± 0.17	76.90 ± 0.22
CUDA Backend	0.45 ± 0.05	12.12 ± 0.02	12.19 ± 0.01
Curand [8]	0.51 ± 0.01	N/A	0.67 ± 0.05

  - Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (Ivy Bridge)
  - NVidia Tesla K40M-12GB (2880 cores)
  - The word size (W) and round (R) used for Random-123 were W4x32_R20 for Threefry and W4x32_R10 for Philox
- VecMath/Rng also supports vectorized PDFs
  - Gauss, Gamma, Poisson, etc.
  - Vector gain $\sim 3-4$ on Intel(R) Core(TM) i7-4510U CPU
  - For details, see the poster by Oscar Chaparro (IPN, Mexico)

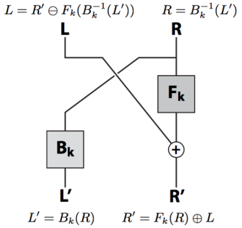
Note on Vector Performance of Philox

- Philox: Advance Randomization System (ARS)
  - iterated bijection with rounds of the Feistel function

$$L' = B_k(R) = \lfloor (R \times M)/2^W \rfloor$$

$$R' = F_k(R) \oplus L = (R \times M) \bmod 2^W \oplus k \oplus L$$

  - left and right bit of $(R \times M)$ and 2 XOR operations



- Poor vector performance of Philox is not due to ARS itself, but there is no directly conversion between 64 and 32 bit (SIMD) integers (size 4 vs. 8 in AVX) $\rightarrow$ scalar ops.

```
UInt64_v product = ((UInt64_v)R)*((UInt64_v)M); //error
left = product >> 32;
right = (UInt32_v)product;
```

- A common problem for generators that involve long integer operations: another ex: RanLux++ [9] (SWB)

Reproducibility under Concurrent Tasks

- Reproducibility between different modes and repeatability within the same mode are general requirements for HEP simulation
- Challenges for track-level parallelism under concurrent simulation work flows (the GeantV approach, talk by A. Gheata)
  - events are mixed and track processing order is not deterministic
- Strategy: a track owns a pRNG state (or object)
  - generate output variate and update the given state in a thread-independent way

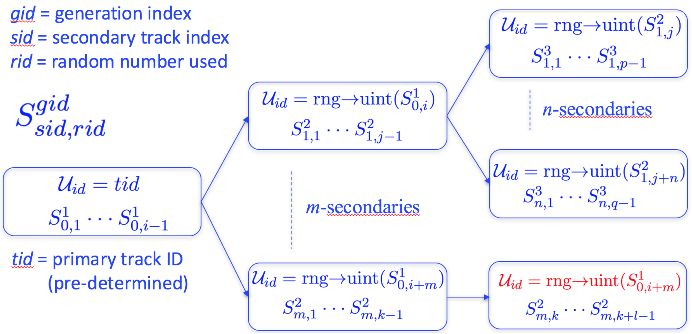
    ```
    r1 = rng->Uniform(track.RngState());
    ```

  - assign a unique sequence (stream) to each track in a collision resistant way (example: initialize the random state of a new secondary track with the random state of the parent)

    ```
    index = rng->UniformIndex(track.RngState());
    secondaryTrack.InitializeState(index);
    ```

Reproducibility: Stream Assignment (A Solution)

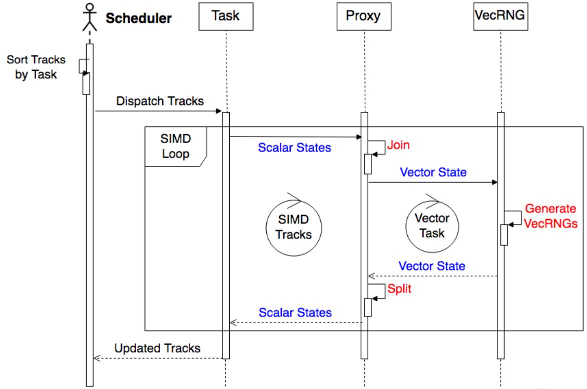
- Reproducibility of the random forest of tracking trees under task-level parallelism: $S=$ random state of the track
  - seed = run ID $\otimes$ event ID
  - unique stream ID, $\mathcal{U}_{id} = \text{rng}\rightarrow\text{UniformIndex}(S) \equiv \text{rng}\rightarrow\text{uint}(S)$ (ex: output of Random123 is a collision-resistant AES/ARS hash)


gid = generation index
sid = secondary track index
rid = random number used
$S^{gid}_{sid,rid}$
$\mathcal{U}_{id} = tid$
$S^1_{0,1} \cdots S^1_{0,i-1}$
tid = primary track ID (pre-determined)
$\mathcal{U}_{id} = \text{rng}\rightarrow\text{uint}(S^1_{0,i})$
$S^2_{1,1} \cdots S^2_{1,j-1}$
m-secondaries
$\mathcal{U}_{id} = \text{rng}\rightarrow\text{uint}(S^1_{0,i+m})$
$S^2_{m,1} \cdots S^2_{m,k-1}$
$\mathcal{U}_{id} = \text{rng}\rightarrow\text{uint}(S^2_{1,j})$
$S^3_{1,1} \cdots S^3_{1,p-1}$
n-secondaries
$\mathcal{U}_{id} = \text{rng}\rightarrow\text{uint}(S^2_{1,j+n})$
$S^3_{n,1} \cdots S^3_{n,q-1}$
$\mathcal{U}_{id} = \text{rng}\rightarrow\text{uint}(S^1_{0,i+m})$
$S^2_{m,k} \cdots S^2_{m,k+l-1}$

- Compare this to a different (pedigree) approach by D. Savin [10]

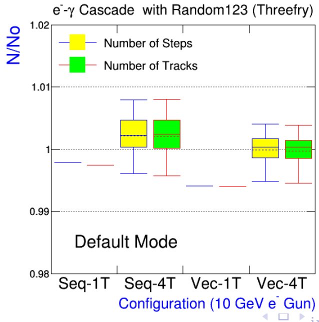
Reproducibility: Strategies for Vector Tasks

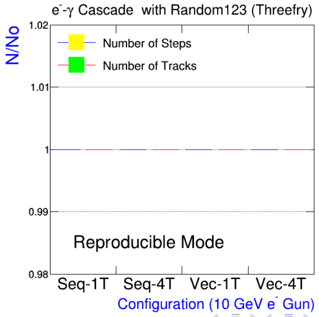
- Results of simulation should be reproducible (identical) between different modes (scalar vs. vector)
- Extension for deterministic vectorized sampling with vector rngs
  - Gather approach: generate scalar Rngs$\rightarrow$ gather to a SIMD array
  - Proxy approach: join–states $\rightarrow$ generate VecRngs $\rightarrow$ split-state



Reproducibility: Verification

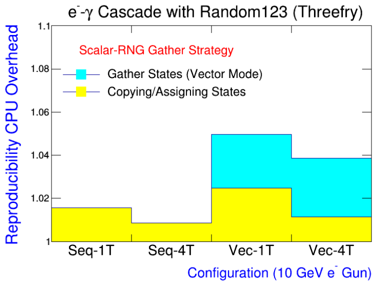
- Test setup with 10 GeV e^- with a subset of GeantV EM physics
  - passage through 50 layers of LAr-Pb calorimeter (TestEM3 [11])
  - $e^- - \gamma$ cascade (Bremstrahlung, Ionization and Compton)
  - 20 measurements (runs) of 1000 events, 10 e^-/event
  - configuration: (Sequential/Vector) $\otimes$ (1Thread/4Threads)
- The total number of steps (tracks) normalized to those of the first run of the reproducible Seq-1T mode, N_o ($\sim 5.6 \times 10^8$ steps)

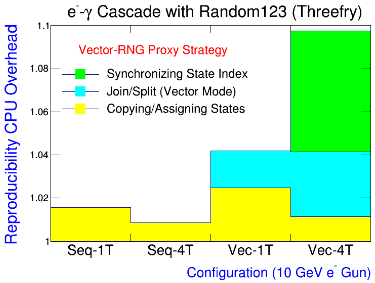




Reproducibility: Overhead

- Reproducibility introduces overhead due to
  - copying/assigning RNG states during workflows
  - gather RNGs to a SIMD array or join/split states
  - synchronizing the index of states in output (Random123 specific)
- Overhead: Time(Reproducibility)/Time(Fast Mode)





- the overhead of join/split of the proxy approach will be negiable if N(vector random numbers used) $\gg$ 1 per join/split.

Summary and Future Work

- Implemented several state-of-the-art RNG algorithms which support parallel generation of random numbers in both SIMD and SIMT workflows.
- Demonstrated reproducibility of propagating multiple particles in parallel in HEP event simulation with concurrent workflows.
- Discussed strategies for efficient uses of vectorized pRNG.
- Future Work under VecMath
  - add MIXMAX
  - add more probability distributions (random variates)

Acknowledgment

Thanks to the GeantV (vector prototype) development team.

- 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
- 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
- K. Savvidy and G. Savvidy, MIXMAX Random Number Generator, arXiv:1510.06274v3 (2011), K. Savvidy, The MIXMAX Random Number Generator, arXiv:1403.5355v2 (2014)
- G. Marsaglia, DIEHARD: A batter of tests of randomness (1996) `http://stat.fsu.edu/ geo/diehard.html`
- 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)
- G Amadio *et al* 2018 *J. Phys.: Conf. Ser.* **1085** 032034

- Kretz M and Lindenstruth V 2011 Vc: A C++ library for explicit vectorization *Software: Practice and Experience.* `http://dx.doi.org/10.1002/spe.1149`
- NVIDIA CUDA Toolkit 5.0 CURAND Guide `http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf`
- A. Sibidanov, A revision of the substract-with-borrow random number generators, arXiv:1705.03123v1 (2017)
- Savin D, Using Pseudo-Random Numbers Repeatably in a Fine-Grain Multithreaded Simulation *https://sd57.github.io/g4dprng/gsocPreprint.html*
- TestEM3, an example of GeantV applications *https://gitlab.cern.ch/GeantV/geant/tree/master/examples/physics*
- P. L'Ecuyer *et al*, Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs, Mathematics and Computers in Simulation 135 (2017) 3-17

pRNG: General Definition [2]

- A finite set of states space: $\mathcal{S}\{i = 0, n : s_i\}$
  (state: a set of parameters to generate random numbers)
- A key space: $\mathcal{K}\{k\}$ and an integer output multiplicity: $J \in \mathbb{Z}$
- A typical u.i.i.d output space: $\mathcal{U}\{u_{k,i}\} \in (0, 1)$

$$\begin{aligned} f &: \mathcal{S} \to \mathcal{S} \\ g &: \mathcal{S} \times \mathcal{K} \times \mathbb{Z}_J \to \mathcal{U} \end{aligned}$$

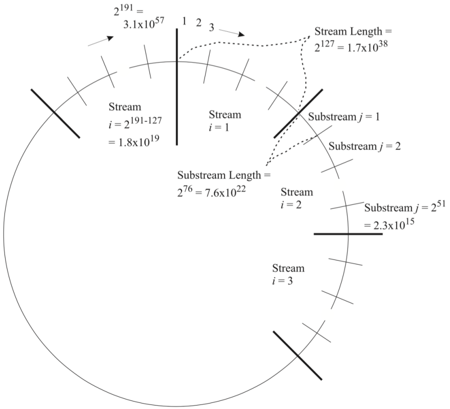
  - transition function f: $s_{i+1} = f(s_i)$ with a seed s_o
  - output function g: $u_{k,i} = g_{k,(i \bmod J)}(s_{\lfloor i/J \rfloor})$ for $i \geq 0$

$$g_{k,j} = h_j \circ b_k \tag{1}$$

  - h_j : a selector (simple)
  - b_k: a keyed bijection (complicated)
- A period $\rho \leq |\mathcal{S}| \leq 2^m$ where m is the bits represent the state

An Example of pRNG with Multiple Streams

- State $\{s_j\}$ of MRG32k3a: $s_{j,n+\nu} = (A_j^\nu \bmod m_j) s_{j,n} \bmod m_j$ (sequence index n, period $\rho = 2^{191}$, stream length $\nu = 2^{127}$)



2^{191} = 3.1×10^{57}
1 2 3
Stream Length = $2^{127} = 1.7 \times 10^{38}$
Stream $i = 2^{191-127}$ $= 1.8 \times 10^{19}$
Stream $i = 1$
Substream $j = 1$
Substream $j = 2$
Substream Length = $2^{76} = 7.6 \times 10^{22}$
Stream $i = 2$
Substream $j = 2^{51}$ $= 2.3 \times 10^{15}$
Stream $i = 3$

Figure: Number of Streams (2^{64}) and substreams (2^{51}) of MRG32k3a [1]

Overlap Probability of Streams with a Hash

- The probability of non-overlap with a multi-stream RNG with a random seed [12]
  - period: ρ
  - number of streams : s
  - length of stream: l

$$p = (1 - \frac{sl}{\rho})^{s-1} \tag{2}$$

- For small sl/ρ, the overlap probability $1 - p \approx s^2 l/\rho$
  - example: $1 - p = 2^{-68}$ for $s = l = 2^{20}, \rho = 2^{128}$
- Alternatives: a combination of pRNGs
  - select a seed for each stream by another pRNG (reproducibility)
  - tree of random streams (created dynamically)
- Overlap probability of multiple streams using the hash mechanism is nearly zero for typical HEP detector simulation.

Notes on Reproducibility Overhead

- Overhead of assigning a unique stream ID (s) depends on efficiency of the skip ahead
  - Random123: no cost by assigning the key $= s$
  - MRG32k3a: (A^s mod m) using the binary decomposition of s
  - MIXMAX: $O(N^2 log(p))$
- Reproducibility of Random123 sequences under the vector and multi-threaded workflow requires reinitializing the state index for each random number generation since
  - tracks undergo either vector or scalar loop undeterministically
  - there is no way to keep tracking the index between the scalar and the vector backend (example of the Philox state)

```
R123::array_t<BackendT,4> ctr;
R123::array_t<BackendT,2> key;
R123::array_t<BackendT,4> ukey;
unsigned int index;
```

  - generate ukey only when index $= 0$ and output $=$ ukey[index] ;