MIXMAX & other RNGs for GeantV

J. Apostolakis (CERN) & S.Y. Jun (Fermilab)
3 July 2018

Conference and Workshop of the MIXMAX Consortium, NCSR Demokritos, 3-4 July 2018
https://indico.cern.ch/event/731433/

HEP experiment simulation

- Use **Radiation Transport** simulation ("Detector Simulation") for
  - design,
  - calibration and
  - analysis of experiments
- Simulation of LHC experiments are large user of computing resources
  - ~ 50% of total experiment CPU time (which use 100,000 CPU cores .)
- **Geant4** is the simulation engine in most recent HEP experiments
  - Including ATLAS, CMS and LHCb at LHC
- The **GeantV** Vector prototype investigating potential of computing speed gains using vectorisation & improved use of CPU cache
  - GeantV requires vectorised versions of good PRNG engines

Transport Monte Carlo & PRNGs

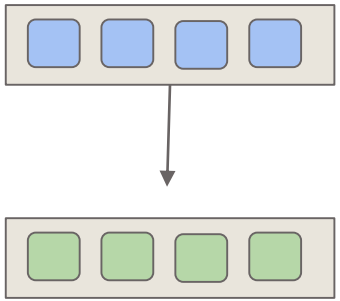
PRNGs are a key part of Radiation Transport Monte Carlo

- For each secondary particle, its **properties are sampled** from **distributions** using PRNG outputs (‘variates’)
- Typically O(10) variates used for **every** electromagnetic (EM) physics **interaction**. More for hadronic interactions.
- PRNGs take **2-3% of CPU time**, and the goal is to reduce it below 2.0% while using a PRNG with the best statistical properties
- The use of PRNGs is relatively **‘robust’** - few of the values are critical
- Use of **high quality PRNG** (within CPU cost ceiling) is necessary to avoid correlations between events, and to avoid very costly re-simulation

VecRNG and Simulation

```
Double_v lambda = ..
Double_v step= lamda * Uniform();
```

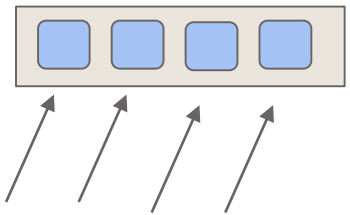
Uniform()



- A vector PRNG object can be used with vector physics models (or other classes)
- In the simple (non-reproducible) mode, a **per-thread** object is used.
- In order **to be reproducible**, the PRNG state must be owned by a track, and it must be used only for the simulation of that track.

- *Note: VecRNG is part of VecMath library - based on VecCore vector library*

VecMath git repository at https://github.com/root-project/vecmath



States of different **instances** of 'scalar' generator

Geant Vector Prototype & PRNGs

GeantV requires both scalar and vector PRNGs

- Some simulation is done in ‘scalar’ mode
- Most simulation is done in ‘vector’ mode, with one track in each lane of a (hardware) vector
  - E.g. X-coordinate (x0, x1, x2, x3) or energy (e0, e1, e2, e3) of tracks 0, 1, 2 & 3

Needs: (see full requirements in talk at GeantV community meeting, Oct 2016)

- Create an interface to extend sequential RNGs for vector use

VecRNG is our draft vector PRNG interface

- Static polymorphism: methods are not virtual
- Multiple outputs expected (one per lane) - from different ‘streams’

Choice of Generators: Not A Limited 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 ($\geq 2^{200}$) and fast with a small state (memory)
  - Exact repeatability (on various computing platforms)
  - Efficient ways of splitting the sequence into long disjoin steams
  - 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 64-bit int	136 bytes	10^{294}	$\sim \infty$

x2 for 64-bit

Generators in VecRNG

Implemented already:

- MRG32k3a (137 ns scalar, 58 ns AVX per output value)
- Random123's ThreeFry (74 ns scalar, 43 ns AVX per ouptut value)
- Random123's Philox (55 ns scalar, 77 ns AVX per output value)

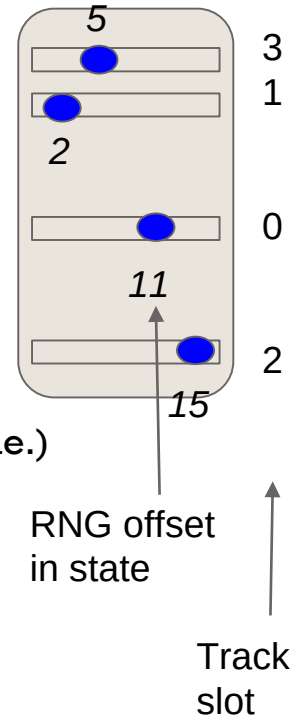
Underway / planned:

- MIXMAX N=17 (and potentially N=8)
- Ranlux

Benchmark for 10^6 values on Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (Ivy Bridge). The word size (W) and round (R) used for Random-123 were W4x32 R20 for Threefry and W4x32 R10 for Philox

Needs of “per-track state”

- GeantV reproducible mode means per-track ‘owned’ PRNG object/state
- Requirement(s) / constraints
  - Keep the same interface when moving to reproducible mode (as much as possible.)
  - Ensure that after its use, the per-track PRNG’s state is correctly updated
- Constraints
  - Speed - as low overhead as possible compared with ‘simpler’ VectorRNG use
  - Each PRNG state will be at a different offset/location in its output (eg MIXMAX, RANLUX) because its track had a different history, and number of variates used.
  - Avoid copying of significant amount of data



Reproducibility for Vector Tasks

- A typical workflow for a vector task with tracks[ntracks]

```
pRNG<VectorBackend> *vrng = new pRNG<VectorBackend>; //per task
pRNG_t<VectorBackend> *state = vrng->GetState();
size_t numChunk = ntracks/vecSize; size_t base = 0;

for (int i = 0; i < numChunk ; ++j) { //vector loop
 //gather
 for(int j=0; j<vecSize; ++i) state[j] = track[base+j]->rng->GetState();
 vrng->SetState(state);

 //Do a vectorized task: use as many vrng->Uniform<VectorBackend>()

 //scatter
 state = vrng->GetState();
 for(int j=0; j<vecSize; ++j) track[base+j]->rng->SetState(state[j]);
 ibase += vecSize;
}
//sclar loop for the remaining (ntracks-numChunk*vecSize) tracks
```

- Reproducibility with a re-usable buffer/proxy

VecRNG Proxy Design/Interface

- New approach: create a ‘proxy’ object that appears like a vector-RNG object
  - The proxy will provide all the methods of VecRNG object
- Constructor **VecRNGproxy(prngState* trackPrng[VecLen], numVariatesExpected)** takes
  - a set of RNG states (corresponding to the tracks in the lanes)
  - the expected number of outputs (variates)
- Destructor ensures that per-track RNG objects are correctly updated
- Its implementation will hide the complexity of the previous page code into the proxy RNG class

Newest results / ongoing

First Proxy implementation ‘JoiningProxyMRG32k3a’’ copying state of MRG32k3a

Initial benchmarks:

- overhead of copying requires about 10 output values per stream to compensate cost of copying state (in and out) of proxy

Investigating vectorisation of ‘partial sum’ operation

- Key operation for MIXMAX generator

Summary / next steps

Design of vector RNG interface

First t of vector RNGs using MRG32k3a and Random123

Per-track state of generator will be used for reproducible simulation.

First implementation to adapt VecRNG generators

Next steps:

- Full performance evaluation, used with vectorised EM physics models.
- Vector extension to MIXMAX
- Investigation of ‘pinning’ events to threads - can be fully repeatable with simple RNG state (1 vector + 1 scalar) and thus reproducible without per-track state.

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 (201), K. Savvidy, The MIXMAX Random Number Generator, arXiv:1403.5355v2 (2014)

4. G. Marsaglia, DIEHARD: A batter of tests of randomness (1996) `http://stat.fsu.edu/ geo/diehard.html`

5. 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)

6. 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

7. UME::SIMD, A library for explicit simd vectorization. https://github.com/edanor/umesimd

8. VCL (C++ vector class library), www.agner.org/optimize

Reproducibility & use of RNG

To ensure reproducibility the number of PRNG output values (variates) consumed in a method must be constant.

- It must NOT depend on whether scalar or vector call is made,
- It must NOT depend on what happens to tracks in other vector lanes.

All algorithms that consume PRNG variates **must** ensure that the **number** of variates consumed depends only on the values of the track itself and the PRNG output values.

So if there are variations in 'consumption', it seems that one of two strategies must be used:

- To consume exactly as many as necessary for every track.
- To "normalize" the number of PRNG variates to be the equal to a fixed maximum (potentially wasteful.)