Parallel Random Number Generation for SIMD/SIMT

Soon Yung Jun[1], **Philippe Canal**[1], **John Apstolakis**[2]

[1] **Fermi National Accelerator Laboratory**   [2] **CERN**

Contact: syjun@fnal.gov

Abstract

Efficient random number generation with high quality statistical properties is one of important components of Monte Carlo simulation in many areas of computational science. As high performance parallel computing systems equipped with deeper vector pipelines or many-cores technologies are widely used, a variety of parallel pseudo-random number generators (PRNGs) have been developed for specific hardware architectures such as SIMD or GPU (SIMT). However, concurrent random number services for hybrid computing models are not commonly available. In this report, we present design considerations, implementation details and preliminary computing performance of parallel PRNG engines which support both SIMD and SIMT with multiple independent streams of random number sequences.

Pseudo-Random Number Generation

$$s_{i-1} \longrightarrow \boxed{f} \longrightarrow s_i \longrightarrow \boxed{g_{k,j}} \longrightarrow u_{k,j}$$
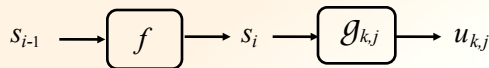
f (transition function): $\mathcal{S} \to \mathcal{S}\{s_i\}$ (state space),

g (output function) : $\mathcal{K} \times \mathcal{S} \to \mathcal{U}\{u_{k,j}\}$ (output space $\in [0,1)$)

$\mathcal{K}\{k\}$ (key space), $g_{k,j} = b_k \circ h_j$ (b_k: bijection, h_j: selector)

Algorithm-basis vs. Counter-basis Generators

Ex. $\mathbb{F}_2$-Linear modulo m

$\vec{x}_i = \mathbf{A}\,\vec{x}_{i-1} \mod 2$
$\vec{y}_i = \mathbf{B}\,\vec{x}_i \mod 2$
$u_i = \sum_{l=1}^{w} y_{i,l-1}\,2^{-l}$

LFSR, GFSRMT, WELL,...

	Algorithm	Counter
f	complicated	simple
b	-	complicated
h	simple	medium

Strong block cipher

AES, ARS

Parallel pRNG for SIMD/SIMT

$$\vec{s}_{i-1} \longrightarrow \boxed{F} \longrightarrow \vec{s}_i \longrightarrow \boxed{G_{k,j}} \longrightarrow \vec{u}_{k,j}$$
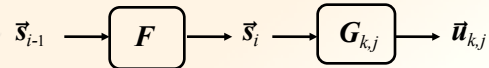
Parallel Random Number Generation

$\vec{s}_{i-1} = (s^1, s^2, \ldots s^m)_{i-1}$

$\vec{u}_{k,j} = (u^1, u^2, \ldots u^m)_{k,j}$

m = SIMD vector, or SIMT threads

128 bits: xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7

Vectorization
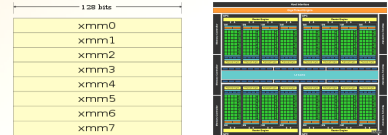
• Common Kernels are used for both SIMD and SIMT using VecCore[1]

Backend VecCore[1] → F / $G_{k,j}$ → **Common Kernels** → SIMD / SIMT → Common pRNG
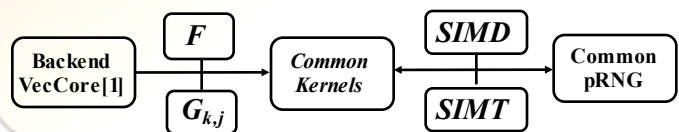
Examples of Implementation

MRG32k3a [2]

Multiple Recursive Generator (MRG) with $\mathbb{F}_2$-Linear pRNG
State $(s_{i,n})$, Linear recurrence vector $(x_{i,n})$ and Output (u_i)

$s_{1,n} = (x_{1,n}, x_{1,n+1}, x_{1,n+2})$
$s_{2,n} = (x_{2,n}, x_{2,n+1}, x_{2,n+2})$
$x_{1,n} = [a_{12}x_{1,n-2} - a_{13}x_{1,n-3}] \mod m_1$
$x_{2,n} = [a_{21}x_{1,n-1} - a_{23}x_{1,n-3}] \mod m_2$
$u_n = [(x_{1,n} - x_{2,n}) \mod m_1]/(m_1 + 1)$
period $= (m_1^3 - 1)(m_2^3 - 1)/2 \approx 2^{191} \approx 3.1 \times 10^{57}$

```
// struct MRG32k3a (random state of MRG32k3a)
template <typename BackendT>
struct MRG32k3a_t {
  typename BackendT::Double_v fCg[MRG::vsize];
};

//class MRG32k3a<BackendT>
template <typename BackendT>
class MRG32k3a : public VecRNG<MRG32k3a<BackendT>, BackendT, MRG32k3a_t<BackendT> >
{
  VECCORE ATT HOST DEVICE template <typename ReturnTypeBackendT>
  typename ReturnTypeBackendT::Double_v Kernel(MRG32k3a_t<BackendT>& state)
  {
    using Double_v = typename ReturnTypeBackendT::Double_v;
    Double_v k, p1;

    // Component 1
    p1 = MRG::a12 * state.fCg[1] - MRG::a13n * state.fCg[0];
    k = math::Floor (p1 / MRG::m1); p1 -= k * MRG::m1;
    Mask_v<Double_v> negative = (p1 < 0.);
    MaskedAssign(p1, negative, p1 + MRG::m1);
    state.fCg[0] = state.fCg[1]; state.fCg[1] = state.fCg[2]; state.fCg[2] = p1;

    // Component 2
    p2 = MRG::a21 * state.fCg[5] - MRG::a23n * state.fCg[3];
    k = math::Floor (p2 / MRG::m2); p2 -= k * MRG::m2;
    negative = (p2 < 0.);
    MaskedAssign(p2, negative, p2 + MRG::m2);
    state.fCg[3] = state.fCg[4]; state.fCg[4] = state.fCg[5]; state.fCg[5] = p2;

    // Combination
    return Blend((p1 > p2),(p1 - p2) * MRG::norm, (p1 - p2 + MRG::m1) * MRG::norm);
  }
}
```

Random123 [3]

Threefry: N rounds of AES with a user key (Weyl sequence)

$\text{round\_key}_0 = \text{user\_key}$
$\text{round\_key}_i = \text{round\_key}_{i-1} + c$

Philox: iterated bijection with complex round functions

$L = B_k(R) = \lfloor (R \times M)/2^W \rfloor$
$R = F_k(R) \oplus L = (R \times M) \mod 2^W \oplus k \oplus L$

SP-Network

```
// struct Threefry t (random state of Threefry-4x32-20)
template <typename BackendT> struct Threefry_t {
  unsigned int index;
  R123::array4_t<BackendT> ctr;
  R123::array4_t<BackendT> key;
  R123::array4_t<BackendT> ukey;
};

//class Threefry<BackendT>
template <typename BackendT>
class Threefry : public VecRNG<Threefry<BackendT>, BackendT, Threefry_t<BackendT> >
{
  VECCORE ATT HOST DEVICE inline
  typename BackendT::UInt32_v RotL_32(typename BackendT::UInt32_v x, unsigned int N)
  { return (x << (N & 31)) | (x >> ((32-N) & 31)); }

  VECCORE ATT HOST DEVICE
  void BijectAndShuffle(R123::array4_t<BackendT> X, R123::array_t<BackendT,5> ks,
                        unsigned int start, unsigned int index)
  {
    X[0] += X[1]; X[1] = RotL_32(X[1],R123::R_32x4[start+0][0]); X[1] ^= X[0];
    X[2] += X[3]; X[3] = RotL_32(X[3],R123::R_32x4[start+0][1]); X[3] ^= X[2];
    X[0] += X[3]; X[3] = RotL_32(X[3],R123::R_32x4[start+1][0]); X[3] ^= X[0];
    X[2] += X[1]; X[1] = RotL_32(X[1],R123::R_32x4[start+1][1]); X[1] ^= X[2];
    X[0] += X[1]; X[1] = RotL_32(X[1],R123::R_32x4[start+2][0]); X[1] ^= X[0];
    X[2] += X[3]; X[3] = RotL_32(X[3],R123::R_32x4[start+2][1]); X[3] ^= X[2];
    X[0] += X[3]; X[3] = RotL_32(X[3],R123::R_32x4[start+3][0]); X[3] ^= X[0];
    X[2] += X[1]; X[1] = RotL_32(X[1],R123::R_32x4[start+3][1]); X[1] ^= X[2];
    // InjectKey (r=index)
    for(size_t ir= 0; ir<4; ++ir){ int kpos = (ir+index) % 5; X[ir] += key[kpos]; }
    X[3] += index;  // X[WCNT4-1] += r
  }
}
```

Design Consideration

➢ Template Approach (Static Polymorphism)
➢ **Support Scalar, Vector (SIMD) and GPU**
➢ **Common kernels (portability)**
➢ Statistical properties
➢ Performance (CPU and memory)
➢ Reproducibility

Base: VecRNG

VecRNG → MRG32k3a, Threefry, Philox, Others

Class Derived: public VecRNG<Derived>

Associated requirements

• A common pRNG service for hybrid computing models
• **Multiple streams (threads) and sub-streams (tasks)**
• Efficient jumping ahead and easy to assign streams
• Reproducibility for multiple threaded applications or concurrent tasks
• Several pRNGs may cooperate together for independent tasks

Performance Measurement

CPU : Intel(R) Xeon(R) CPU E5-2620 (12 cores @ 2.00GHz)
GPU : NVIDIA Tesla K20m (2496 CUDA cores @ 0.71GHz)
Average Time [ms] for generating 10M random numbers (200 measurements)

std::rand() = (139.98 ± 0.06) ms

	Scalar	Vector (SSE)	CUDA	MKL/VSL ✳	Curand ◆
MRG32k3a	209.25 ± 0.07	123.58 ± 0.03	2.03 ± 0.03	145.50 ± 0.06	2.05 ± 0.02
Threefry ☞	129.65 ± 0.06	123.47 ± 1.19	10.22 ± 0.02	N/A	N/A
Philox ☞	100.78 ± 0.04	225.82 ± 0.46	10.17 ± 0.01	N/A	1.92 ± 0.01

✳ Perofrmance and quality of the Intel MKL/VSL[4] random number generation depend on the size of output array: using N=32 for this measurement
❖ Curand library[5] (cuda 8.0, arch=sm_3.5), using (26 blocks x 192 threads)
☞ Word size (W) and round (R) used for Random-123: Threefry (W4x32_R20) and Philox (W4x32_R10)

Work in Progress

• Optimizing performance of Random123 (XOR operation)
• Testing with SIMD/SIMT tasks (ex. vectorized physics processes and models [6])
• **Reproducibity tests under multi-threaded workflows**

Conclusions

• Demonstrated a comom implementation of puedo random number generation for SIMD and SIMT architecture using VecCore
• Measured peformance of parallel random number generation for example generators (MRG32k3a, Threefry and Philox of Random123) and compared their performance to corresponding generators from MKL/VSL and NVIDIA cudrand libraries
    • Sizable gain by an explicit vectorization for MRG32k3a
    • GPU performance comparable to Curand [4] for MRG32k3a
    • Philox (32x10) shows good performance on CPU
    • SIMD Vectorization of XOP operations need to be studied further
• Future work: Reproducibility under hybrid computing environments

References

[1] VecCore: https://github.com/root-project/veccore
[2] 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
[3] 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
[4] Intel® MKL/VSL library, Intel ParallelStudio2016
[5] NVIDIA® Curand library, http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf
[6] Amadio G et al, *Electromagnetic Physics Models for Parallel Computing Architectures*, J. Phys.: Conf., Series 523 01 2004