MRG8–Random Number Generation for the Exascale Era

Yusuke Nagasaka†, *Ken-ichi Miura* ‡, *John Shalf* ‡
Akira Nukada†, *Satoshi Matsuoka*§†
† *Tokyo Institute of Technology*
‡ *Lawrence Berkeley National Laboratory*
§ *RIKEN Center for Computational Science*

Tokyo Tech

BERKELEY LAB
Lawrence Berkeley National Laboratory

Random Number Generator

- **Pseudo random number generator (PRNG)** is a crucial component of numerous algorithms and applications
  - Quantum chemistry, molecular dynamics
  - Broader classes of Monte Carlo algorithms
  - Machine Learning field
    - Shuffling of training data
    - Initializing weights of neural network
    - cf.) Numpy employs Mersenne Twister
- **Pseudo** and **Real** random number
- What is a requirement for "**Good PRNG**"?

Random Number Generator

- **Pseudo random number generator (PRNG)** is a crucial component of numerous algorit
  - Quantum chem
  - Broader classes
  - Machine Learni
    - Shuffling of trai
    - Initializing weig
    - cf.) Numpy emp
- **Pseudo** and **Real** random n
- What is a requirement for "**Good PRNG**"?

> - **Long recurrence length**
> - **Good statistical quality**
> - **Deterministic Jump-ahead for parallelism**
> - **Performance (throughput)**

Recurrence Length

- PRNGs will eventually repeat themselves
  - Eg.) LCG in the C standard library repeat themselves in as few as $2.15 * 10^9$ steps (**too short**)
  - Much additional cost to erase the effect of auto-correlation
    - **Greatly reduce the effective performance of algorithm**
  - Minimum requirement is for an entire year of executing at full speed on a supercomputer

	MT19937	MRG32k3a	Philox	MRG8
Period	$2^{19937} - 1$	2^{191}	2^{130}	$(2^{31} - 1)^8 - 1$

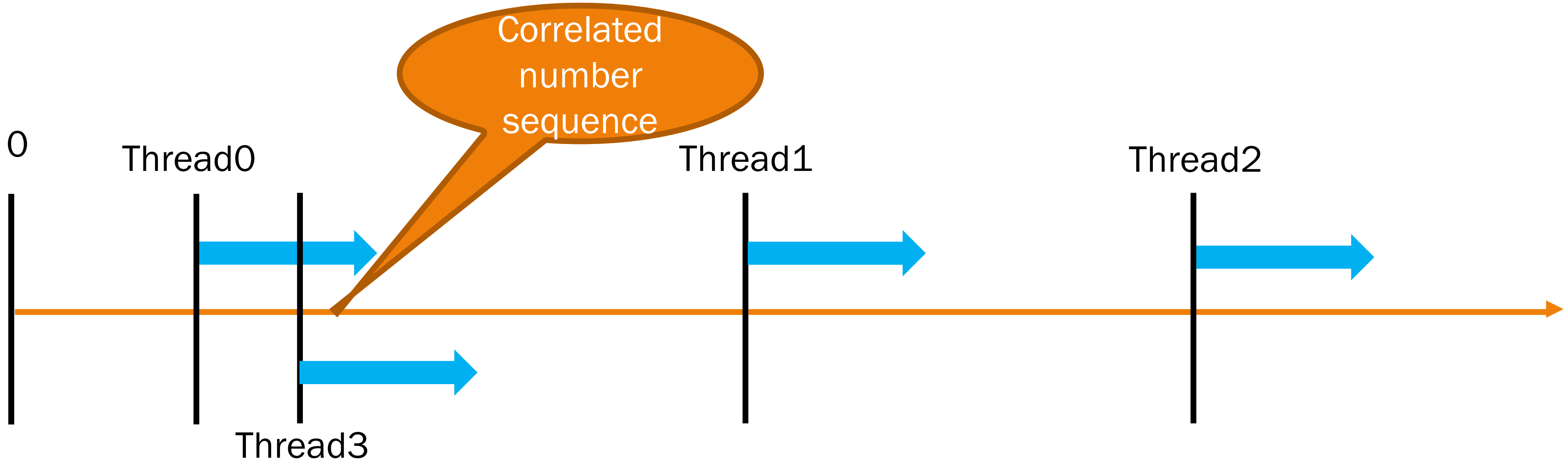
Statistical Quality

- Sequence must show no statistical bias
  - Otherwise, **PRNGs affect the outcome of a simulation**

- **TestU01** developed by L'Ecuyer
  - Benchmark set for empirical statistical testing of random number generators
  - Three pre-defined battery
    - Small Crush: 15 tests, using 2 random numbers
    - Crush: 186 tests, using 2 random numbers
    - Big Crush: 234 tests, using 2 random numbers

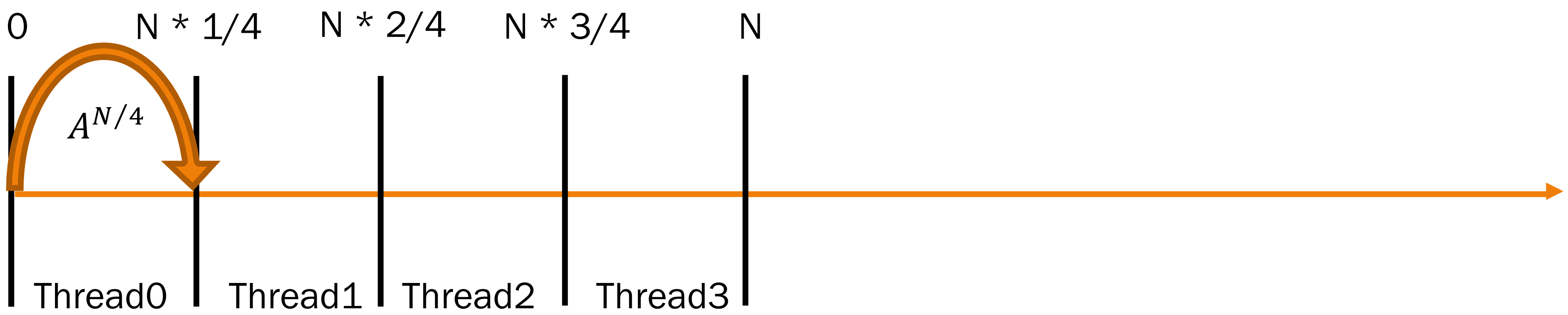
Jump-ahead for Parallelism

- Two primary approaches for parallelization of PRNG
  - **Multistream**
    - Different random “seed” to produce different random number sequence
    - Overhead of setting the start point is not expensive
    - Chance of correlated number sequences is not so low
      - cf.) birthday paradox



Jump-ahead for Parallelism

- Two primary approaches for parallelization of PRNG
  - **Substream (Jump-ahead)**
    - Each worker get a sub-sequence that is guaranteed to be non-overlapping with its peers
      - Parallelization does not break the statistical quality of PRNGs
    - Cost of jump-ahead may hurt parallel scalability



MRG8

- 8^{th}-order full primitive polynomials
  - One of multiple recursive generators
  - Next random number is generated from previous random numbers with polynomial
    - $x_n = a_1x_{n-1} + a_2x_{n-2} + a_3x_{n-3} + a_4x_{n-4} + a_5x_{n-5} + a_6x_{n-6} + a_7x_{n-7} + a_8x_{n-8} \bmod (2^{31} - 1)$
    - Modulo operation can be executed by “bit shift”, “bit and” and “plus” operation
- **Long period**
  - $(2^{31} – 1)^8 \sim 4.5*10^{74}$
- **Good statistical quality**
  - Pass Big crash of TestU01

Contribution

- We reformulate the MRG8 for Intel's KNL and NVIDIA's GPU
  - Utilize wide 512-bit register
  - Exploit parallelism of many-core processors
- Huge performance benefit from existing libraries
  - MRG8-AVX512 achieves a substantial **69% improvement**
  - MRG8-GPU shows a maximum **x3.36 speedup**
- Secure the statistical quality and long period of original MRG8

Reformulating to Matrix-Vector Operation

- Compute multiple next random numbers in one matrix-vector operation

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \boldsymbol{y}_{n-1} = \begin{pmatrix} x_{n-1} \\ x_{n-2} \\ x_{n-3} \\ x_{n-4} \\ x_{n-5} \\ x_{n-6} \\ x_{n-7} \\ x_{n-8} \end{pmatrix}$$

$$\boldsymbol{y}_n = A\, \boldsymbol{y}_{n-1} \bmod p$$

$$\boldsymbol{y}_{n+8} = A^8\, \boldsymbol{y}_n \bmod p$$

$$\begin{pmatrix} \boldsymbol{y}_{n+8} \\ \boldsymbol{y}_{n+16} \\ \boldsymbol{y}_{n+24} \\ \boldsymbol{y}_{n+32} \end{pmatrix} = \begin{pmatrix} A^8 \\ A^{16} \\ A^{24} \\ A^{32} \end{pmatrix} \boldsymbol{y}_n \bmod p$$

A^8, A^{16}, A^{24} and A^{32} can be precomputed

Easily apply vector/parallel processing to Mat-vec op

Jump-ahead Random Sequence in MRG8

- Jump-ahead to arbitrary point
  - When jump to i-th point, compute $A^i\,\boldsymbol{y}_0 \; mod \; p$
  - Implementation: Matrix-vector multiplication
    - Precompute A^{2^j} $(j = 0, 1, 2, \dots, 246)$
    - Compute $A^i\,\boldsymbol{y}_0 \; mod \; p$
      - $A^i = e_1 A^1 * e_2 A^2 * e_3 A^4 * \dots * e_{246} A^{2^{246}} \; (e_j \in \{0, 1\})$
      - In the implementation, **executed as mat-vec**, not mat-mat

Jump-Ahead(A, $\boldsymbol{y}$, i)
    for j = 0 to 246
        do if (i & (0x1)) == 1
            then $\boldsymbol{y}$ = A^(2^j) $\boldsymbol{y}$ mod $2^{31} - 1$
        i = (i >> 1)

MRG8-AVX512: Optimization for KNL

- Efficiently compute $\boldsymbol{y}_{n+8} = A^8\, \boldsymbol{y}_n \; mod \; p$ with **wide 512-bit vector register**
  - Generate 8 double elements in parallel
  - Executed as outer product
- **Low cost of jump-ahead function**
  - Exploit high parallelism (up to 272 threads)

MRG8-AVX512($A_{KNL}, \boldsymbol{y_n}$)

// Generate eight 64-bit floating point random numbers
$MASK \leftarrow (2^{31} - 1)$
$\mathbf{s1} \leftarrow \mathbf{0}$
$\mathbf{s2} \leftarrow \mathbf{0}$
for $q \leftarrow 0$ **to** 3
  do $s_1 \leftarrow s_1 + \boldsymbol{a_q y_n}$
    // PERMUTE($\boldsymbol{x}$) returns $\boldsymbol{w}$ s.t. $w[i] = x[(i+1)\%8]$
    $\boldsymbol{x} \leftarrow$ PERMUTE($\boldsymbol{y_n}$)
for $q \leftarrow 4$ **to** 7
  do $s_2 \leftarrow s_2 + \boldsymbol{a_q y_n}$
    $\boldsymbol{x} \leftarrow$ PERMUTE($\boldsymbol{y_n}$)
$\boldsymbol{s} \leftarrow (\mathbf{s1} \& MASK) + (\mathbf{s1} >> 31) + (\mathbf{s2} \& MASK) + (\mathbf{s2} >> 31)$
$\boldsymbol{s} \leftarrow (\boldsymbol{s} \& MASK) + (\boldsymbol{s} >> 31)$
$\boldsymbol{s} \leftarrow (\boldsymbol{s} \& MASK) + (\boldsymbol{s} >> 31)$
$\boldsymbol{y_{n+8}} \leftarrow \boldsymbol{s}$
$\boldsymbol{r} \leftarrow$ (double)$(\boldsymbol{s} - 1)/MASK$

MRG8-GPU: Optimization for GPU

- **Efficiently compute 32 x 8 matrix-vector operation**
  - Computed as outer product
    - 1 threads compute one random number
  - __umulhi() instruction
    - Multiplication between 32-bit unsigned integers and output is upper 32-bit of result
    - Reduce expensive mixed-precision integer multiplications

$$\begin{pmatrix} \boldsymbol{y}_{n+8} \\ \boldsymbol{y}_{n+16} \\ \boldsymbol{y}_{n+24} \\ \boldsymbol{y}_{n+32} \end{pmatrix} = \begin{pmatrix} A^{8} \\ A^{16} \\ A^{24} \\ A^{32} \end{pmatrix} \boldsymbol{y}_n \bmod p$$

- Too many threads require many “jump-ahead” procedure
  - **Carefully select best number of total threads with keeping high occupancy of GPU**

API of MRG8-AVX512/-GPU

- **Single generation:** double rand();
  - Each function call returns a single random number
  - follows C and C++ standard API
  - **Low throughput** due to the overhead of function call

- **Array generation:** void rand(double *ran, int n);
  - User provides a pointer to the array with the array size
  - Array is filled with random numbers
  - Adopted by Intel MKL and cuRAND

Model for Performance Upper Bound -1-

- Performance upper bound for the Array generation
  - Determined as **min(p_m, p_c)**; memory-bound vs compute-bound use case
  - **Memory-bound case**
    - Restricted by storing the generated random numbers to memory
    - Upper bound is estimated by memory bandwidth of STREAM benchmark
  - **Compute-bound case**
    - Count the number of instructions
    - Only consider the kernel part excluding jump-ahead overhead

Model for Performance Upper Bound -2-

- Intel KNL (MRG8-AVX512)
  - Memory bandwidth is 166.6GB/sec => **p_m = 22.4 billion RNG/sec**
  - Compute-bound: **p_c = 34.6 billion RNG/sec**
    - 44 instructions for 8 random number generation
    - 136 vector units (2 units/core) with 1.4GHz in Intel Xeon Phi Processor 7250
  - 54 % better performance when the array size can fit entirely into L1 cach
- NVIDIA P100 GPU (MRG8-GPU)
  - Memory-bandwidth is 570.5GB/sec => **p_m = 76.6 billion RNG/sec**
  - Compute-bound: **p_c = 49.7 billion RNG/sec**
    - 101 instructions for 1 random number generation
    - 3584 CUDA cores with 1.4 GHz in NVIDIA P100 GPU
  - **MRG8-GPU is a compute-bound kernel in all cases**

Performance Evaluation

Evaluation Environment

- **Cori Phase 2 @NERSC**
  - Intel Xeon Phi 7250
    - Knights Landing (KNL)
    - 96GB DDR4 and 16GB MCDRAM
    - Quadrant/Cache mode
    - 68 cores, 1.4GHz
  - Compiler
    - Intel C++ Compiler ver18.0.0
  - OS
    - SuSE Linux Enterprise Server

- **TSUBAME-3.0 @TokyoTech**
  - NVIDIA Tesla P100
    - #SM: 56
    - Memory: 16GB
  - Compiler
    - NVCC ver.8.0.61
  - OS
    - SUSE Linux Enterprise Server 12 SP2

Evaluation Methodology

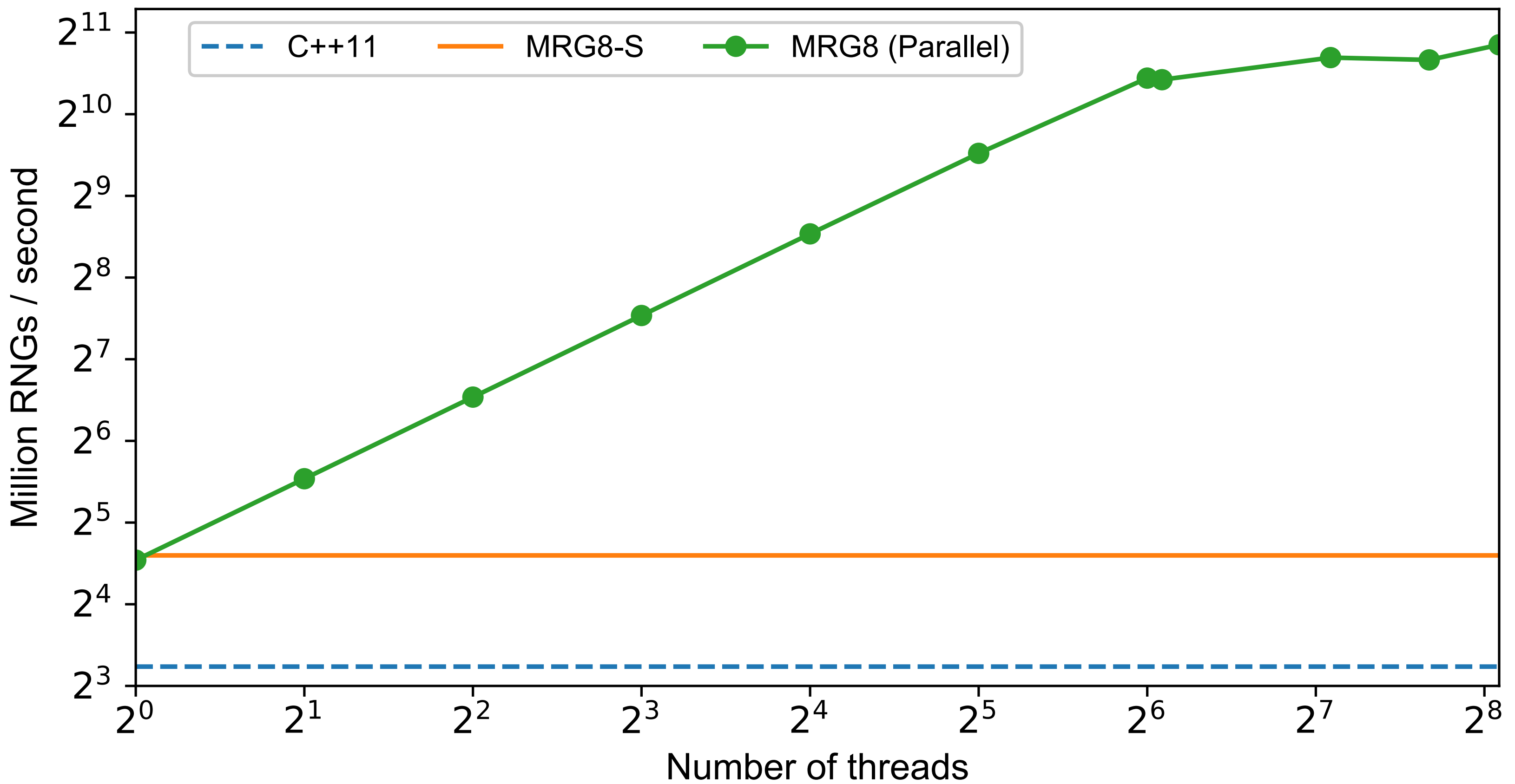
- Generate 64-bit floating random number
- Generating size
  - Single generation
    - 2^24 random numbers
  - Array generation
    - Large: 2^x (x=24~30)
      - Fit into MCDRAM and global memory of GPU, but not cache
    - Small: 32, 64, 128 (only for Intel KNL)
      - More practical case
      - Repeat 1000 times by each thread on KNL
      - Fit into L1 cache

Evaluation Methodology
PRNG Libraries

- Single generation
  - C++11 standard library
    - MT19937
- Array generation
  - Intel MKL
    - MT19937, MT2203, SFMT19937, MRG32K3A, PHILOX
  - NVIDIA cuRAND
    - MT19937, SFMT19937, XORWOW, MRG32K3A, PHILOX

Performance on KNL
Single generation

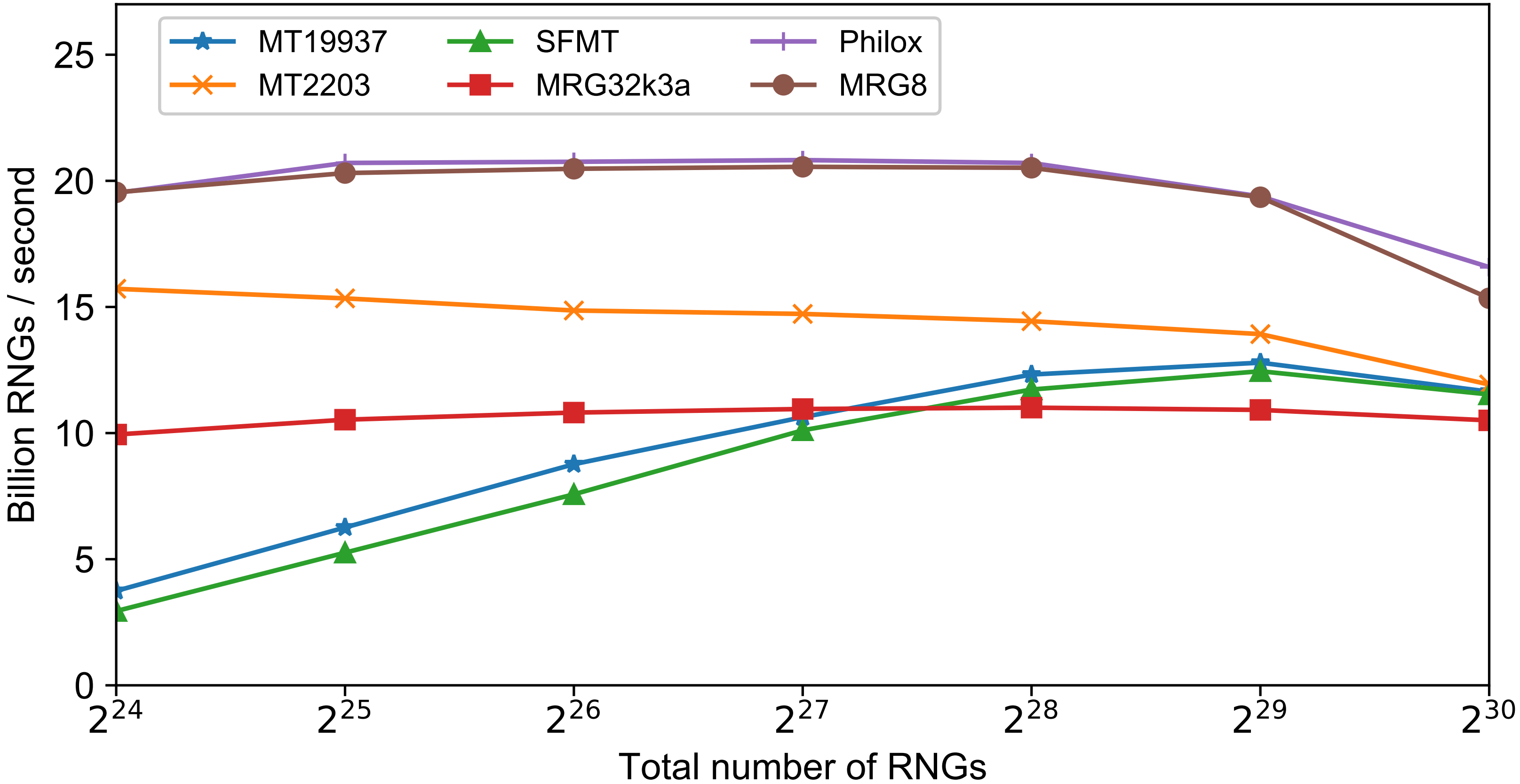
- MRG8 shows **good performance and scalability**
  - C++11 does not support jump-ahead



Performance on KNL

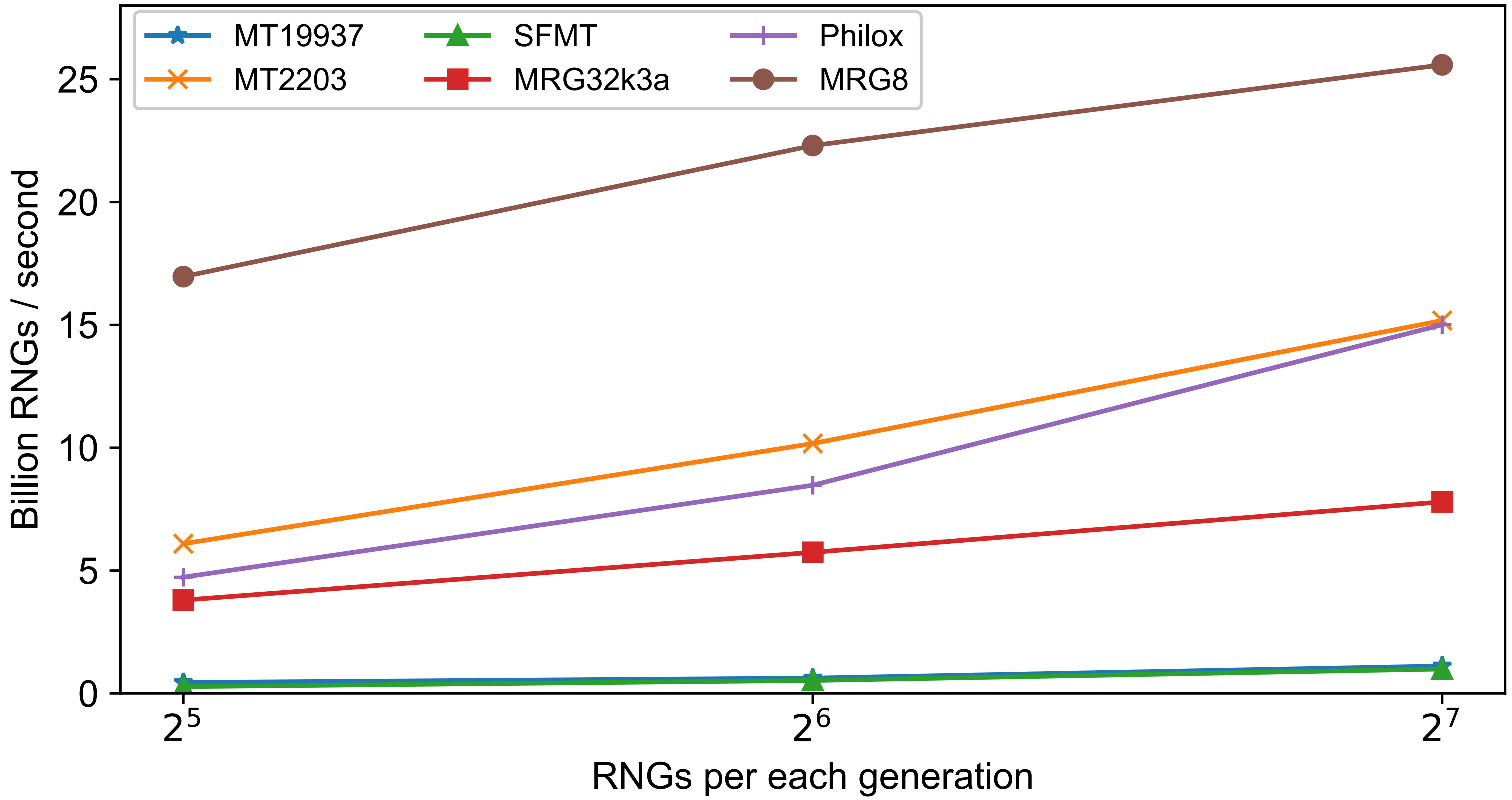
Array generation for large size

- MRG8 shows comparable performance to Philox
  - Both close to the **upper bound for memory bandwidth**



Performance on KNL
Array generation for small size

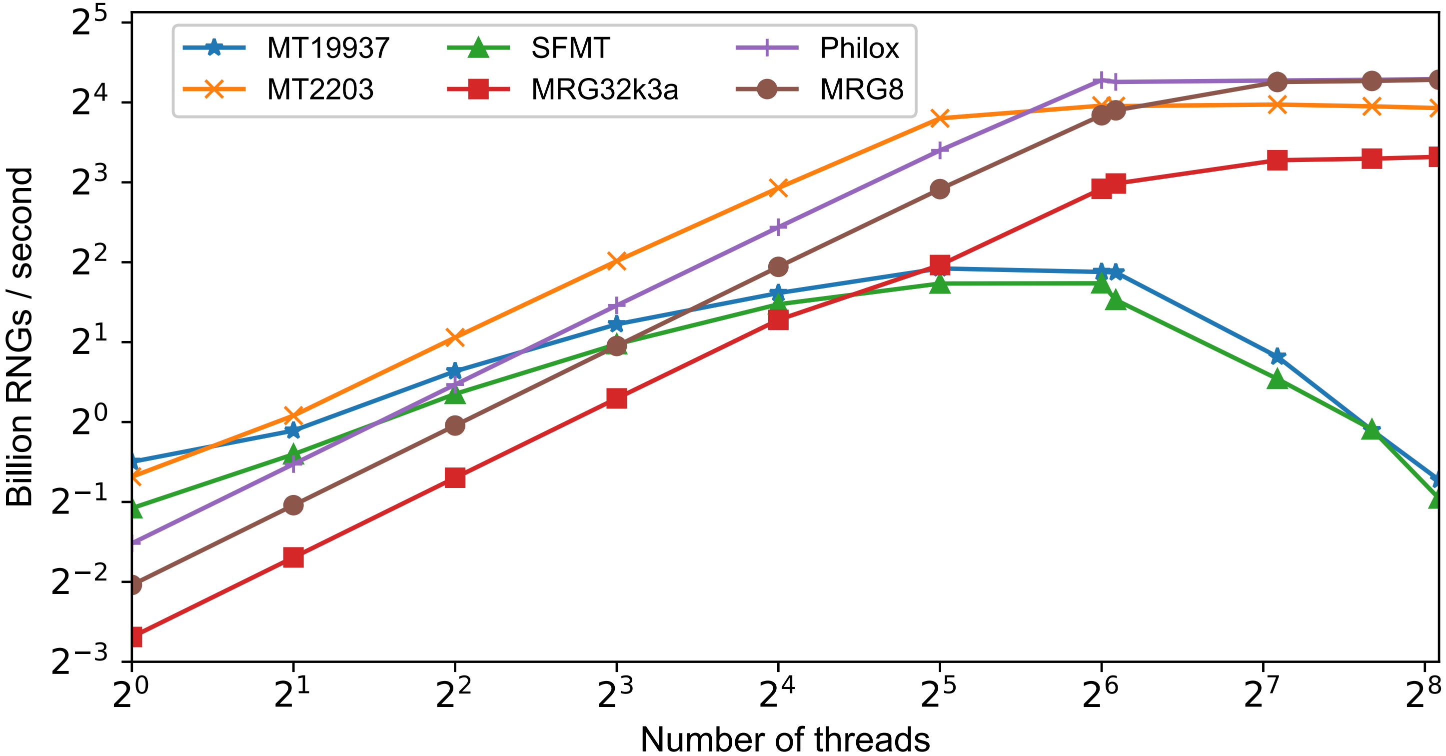
- MRG8 overcomes the upper bound of memory bandwidth
  - **x1.69 faster** than the other random number generations



Performance on KNL

Scalability

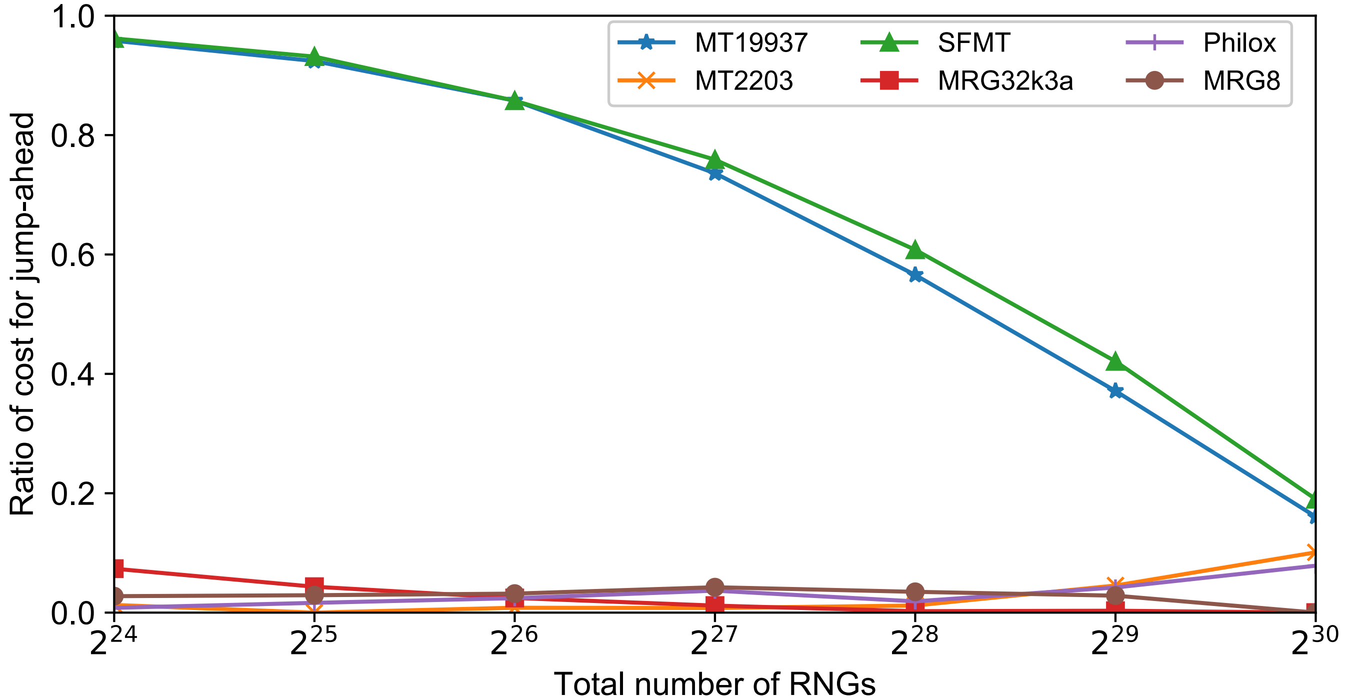
- Performance goes down after 64 threads in MT19937 and SFMT
  - **Large jump-ahead cost**
- MRG8 shows good scalability



Performance on KNL

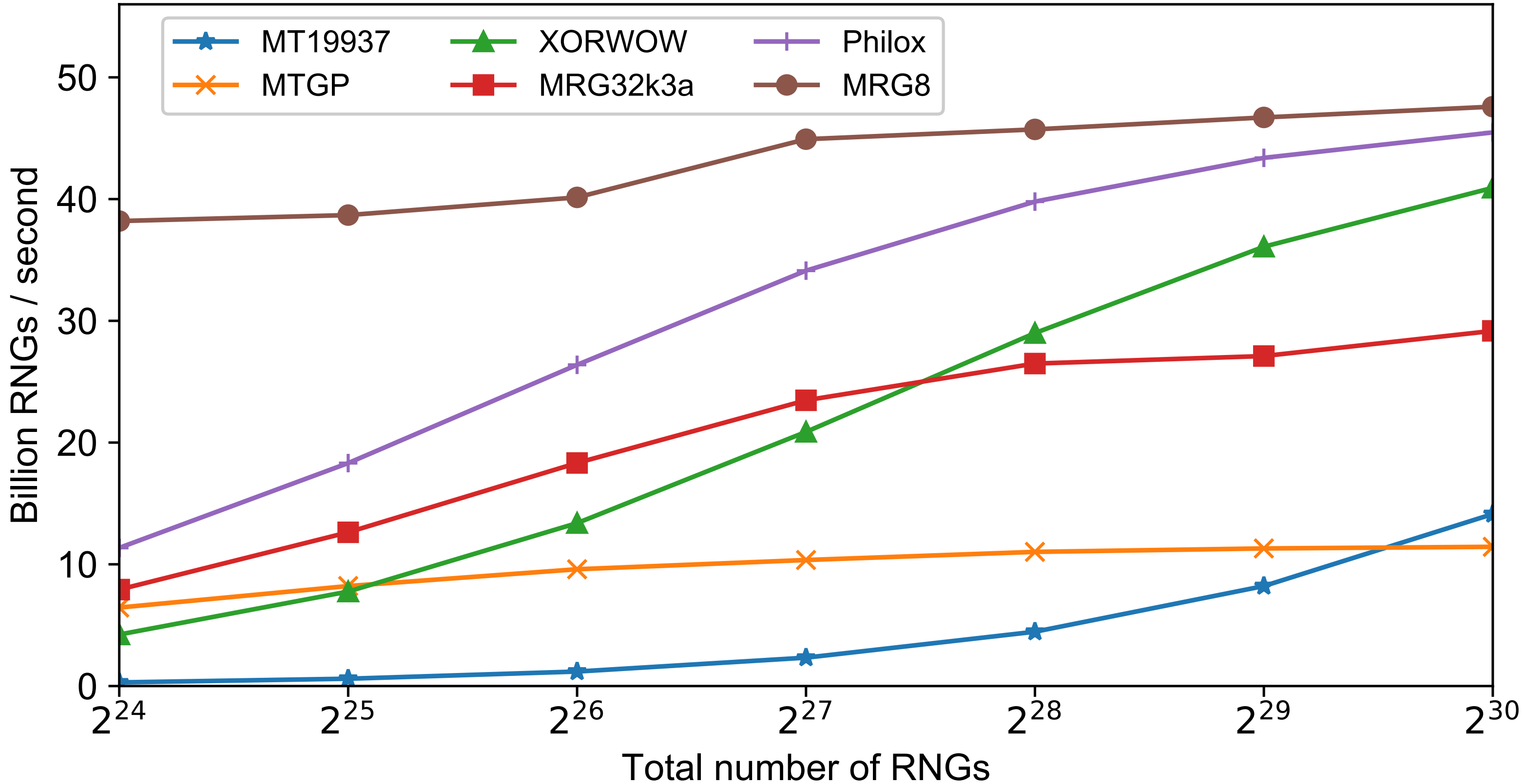
Cost of jump-ahead

- Jump-ahead becomes serious bottleneck on MT19937 and SFMT
  - **Limit scalability**
- **Little cost for jump-ahead in MT2203, MRG32k3a, Philox and MRG8**



Performance on GPU
Array generation

- MRG8 achieves high throughput for any random number sequence length
  - **Up to x3.36 speedup**



Memory Usage of MRG8

- Small memory of many-core processors require less memory usage of random number generator
  - Memory usage of MRG8 is small and does not affect the applications
- MRG8-AVX512
  - 8-by-8 matrix: A^8 matrix and A^i for jump-ahead
  - Thread private state vector
  - **235 bytes / thread** on 272 threads
- MRG8-CUDA
  - 32-by-8 matrix and 8-by-8 matrix for jump-ahead
  - State vector
  - No more than **5 bytes** per thread on 2^{17} threads

Quality of Random Numbers

- Test of statistical quality on TestU01
  - **Secured statistical quality of our MRG8 reimplementation**

	Period (MKL)	Period (cuRAND)	Test
MT19937	$2^{19937} - 1$	$2^{19937} - 1$	
MT2203	$2^{2203} - 1$	$2^{2203} - 1$	
SFMT19937	$2^{19937} - 1$	-	
MTGP	-	$2^{19937} - 1$	
XORWOW	-	$(2^{160} – 1)\ 2^{32}$	
MRG32k3a	2^{191}	$>2^{190}$	✓
Philox	2^{130}	2^{128}	✓
MRG8	$(2^{31} – 1)^{8} - 1$	$(2^{31} – 1)^{8} - 1$	✓

Conclusion

- MRG8 is a high quality PRNG
  - Key qualities of statistical uniformity
  - Efficient parallelism
  - Long recurrence length
- We reformulate the MRG8 for Intel KNL and NVIDIA P100 GPU
  - Huge performance benefit from existing libraries
    - MRG8-AVX512 achieves a substantial **69% improvement**
    - MRG8-GPU shows a maximum **x3.36 speedup**
- Follow-up work
  - Demonstrate the value in real applications

Code is available at https://github.com/kenmiura/mrg8

Acknowledgement

- This work was partially supported by JST CREST Grant Number JP-MJCR1303 and JPMJCR1687, and performed under the collaboration with DENSO IT Laboratory, inc., and performed under the auspices of Real-World Big-Data Computation Open Innovation Laboratory, Japan.
- The Lawrence Berkeley National Laboratory portion of this research is supported by the DoE Office of Advanced Scientific Computing Research under contract DE-AC02-05CH11231.
- One of the authors (KM) would like to thank Prof. Pierre L'Ecuyer of Montreal University for providing the 8th order primitive polynomial for this study.