ThundeRiNG:
Generating Multiple Independent Random Number Sequences on FPGAs

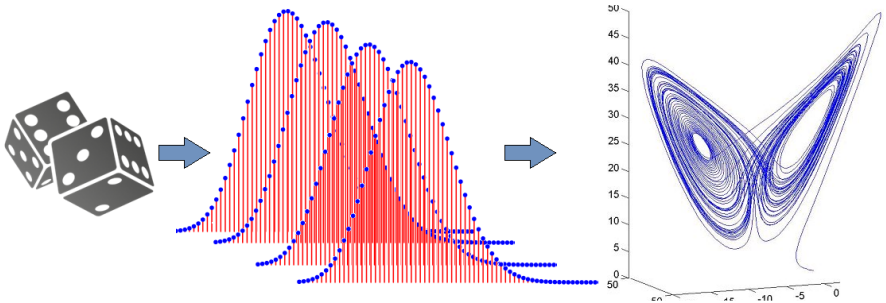
Hongshi Tan[1], Xinyu Chen[1], Yao Chen[2], Bingsheng He[1], Weng-Fai Wong[1]

[1] National University of Singapore

[2] Advanced Digital Sciences Center, Singapore

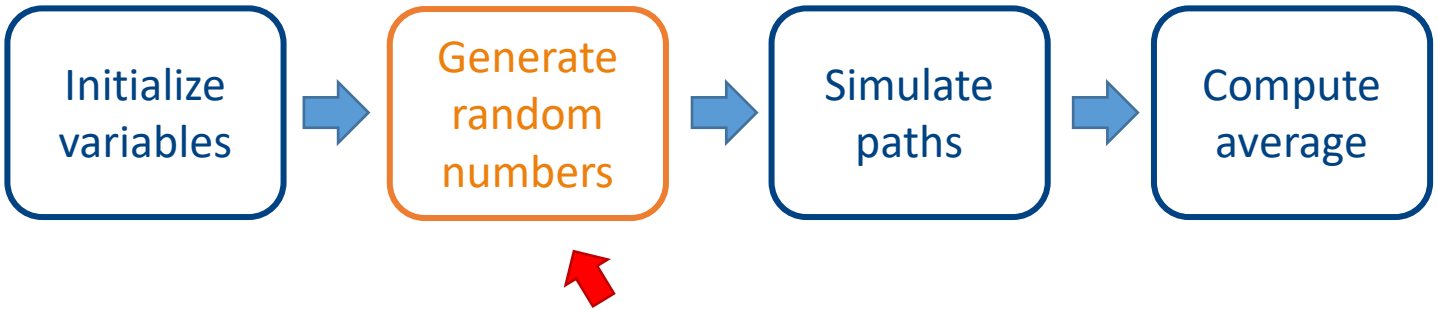
Pseudo Random Number Generators

- Pseudo random number generators (PRNGs) are major performance component in many applications
  - Monte Carlo simulation (MC) in scientific computing
  - Sampling-based data mining and data analysis
  - Emerging computing architecture



Example: Monte Carlo Option Pricing

- Simulation over millions of paths



Account for 54% of the total execution time

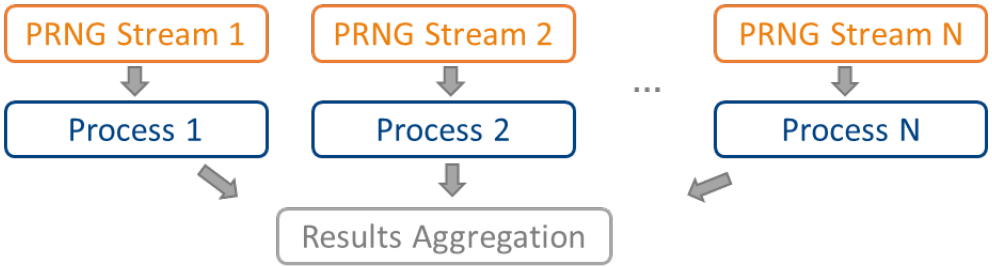
FPGA for PRNGs

- FPGAs have been increasing more applications in data centers.

- Advantages of FPGAs
  - Fine-grained parallelism
  - Energy efficiency
  - Efficient bitwise operation (e.g. truncation, permutation, etc.)
  - Arbitrary-precision integer arithmetic

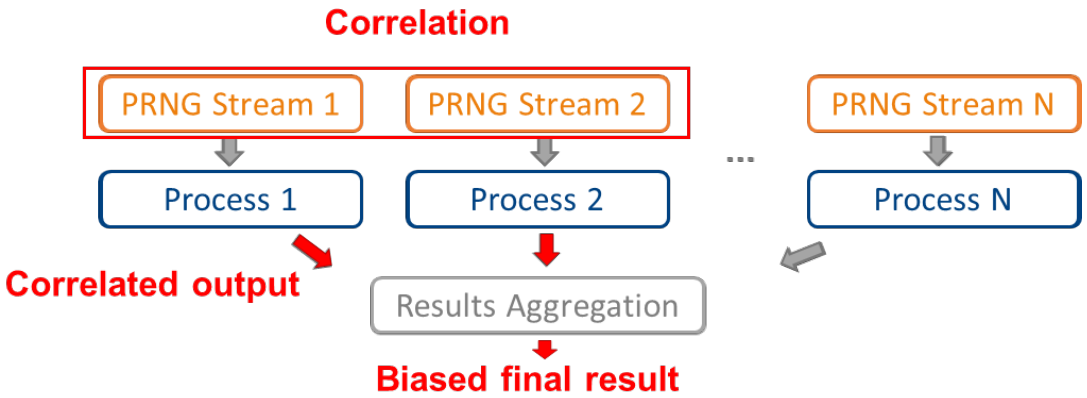
Requirement of MISRN on FPGAs

- Accelerating these applications requires **Multiple Independent Sequences of Random Numbers (MISRN)**.



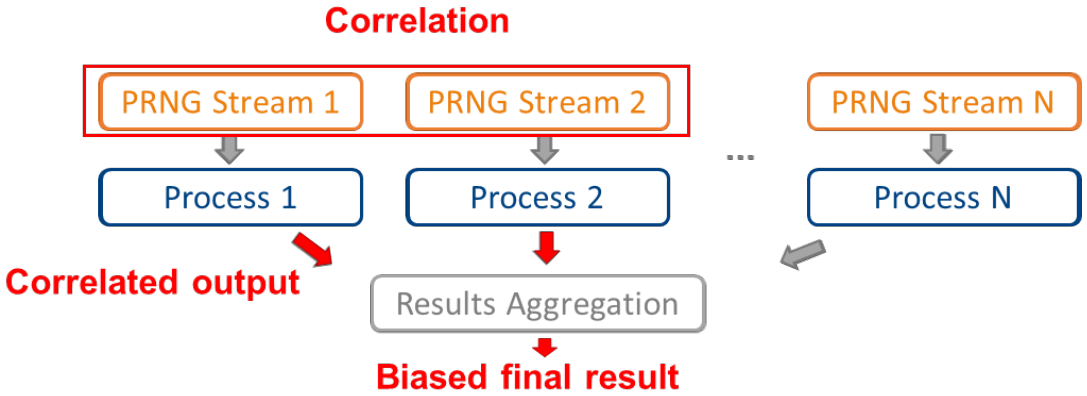
Requirement of MISRN on FPGAs

- Accelerating these applications requires **Multiple Independent Sequences of Random Numbers (MISRN)**.



Requirement of MISRN on FPGAs

- Accelerating these applications requires **Multiple Independent Sequences of Random Numbers (MISRN)**.



High Throughput Generation

High Quality Sequences

Problem of Existing Works

- Using linear recurrence modulo 2 ($\mathbb{F}_2$) algorithms [1,2]

- Requiring on-chip (BRAMs) intermediate variable storage

[1] Li, et al. "Software/hardware parallel long-period random number generation framework based on the well method." VLSI, 2013.
[2] Thomas, et al. "The LUT-SR family of uniform random number generators for FPGA architectures." VLSI 2012.

Problem of Existing Works

- Using linear recurrence modulo 2 ($\mathbb{F}_2$) algorithms [1,2]
  - Lack of **linear complexity**

- Requiring on-chip (BRAMs) intermediate variable storage

[1] Li, et al. "Software/hardware parallel long-period random number generation framework based on the well method." VLSI, 2013.
[2] Thomas, et al. "The LUT-SR family of uniform random number generators for FPGA architectures." VLSI 2012.

Problem of Existing Works

- Using linear recurrence modulo 2 ($\mathbb{F}_2$) algorithms [1,2]
  - Lack of **linear complexity**

→ **Poor Quality** ✕

- Requiring on-chip (BRAMs) intermediate variable storage

[1] Li, et al. "Software/hardware parallel long-period random number generation framework based on the well method." VLSI, 2013.
[2] Thomas, et al. "The LUT-SR family of uniform random number generators for FPGA architectures." VLSI 2012.

Problem of Existing Works

- Using linear recurrence modulo 2 ($\mathbb{F}_2$) algorithms [1,2]
  - Lack of **linear complexity**

→ **Poor Quality** ✕

- Requiring on-chip (BRAMs) intermediate variable storage
  - Limited **number of instances** because of the capacity of hardware resources

[1] Li, et al. "Software/hardware parallel long-period random number generation framework based on the well method." VLSI, 2013.
[2] Thomas, et al. "The LUT-SR family of uniform random number generators for FPGA architectures." VLSI 2012.

Problem of Existing Works

- Using linear recurrence modulo 2 ($\mathbb{F}_2$) algorithms [1,2]
  - Lack of **linear complexity**

→ **Poor Quality** ✕

- Requiring on-chip (BRAMs) intermediate variable storage
  - Limited **number of instances** because of the capacity of hardware resources

→ **Poor Throughput** ✕

[1] Li, et al. "Software/hardware parallel long-period random number generation framework based on the well method." VLSI, 2013.
[2] Thomas, et al. "The LUT-SR family of uniform random number generators for FPGA architectures." VLSI 2012.

Challenges of MISRN on FPGAs

- **Challenge 1**: guarantee independency of multiple random sequences
  - Inter-stream correlation exists among the sequences.

- **Challenge 2**: generate MISRN with high throughput
  - Hardware resources limit throughput and degree of parallelism.

Our Solution — ThundeRiNG

- **Challenge 1**: guarantee independency of multiple random sequences
  - Inter-stream correlation exists among the sequences.
- Solution: a novel **decorrelation mechanism** for multiple sequences of random numbers

- **Challenge 2**: generate MISRN with high throughput
  - Hardware resources limit throughput and degree of parallelism.
- Solution: a **sharing mechanism** to reuse the intermediate resource-hungry stage among multiple instances

Core Algorithms

- Linear Congruential Generator
- Towards High Quality: Decorrelation
- Towards High Throughput: State Sharing

Linear Congruential Generator (LCG)

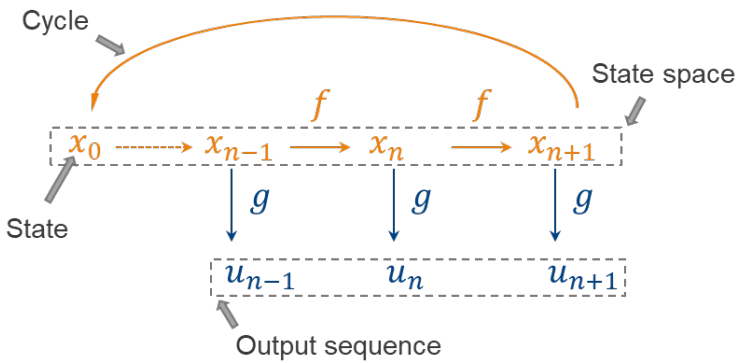
Two Stages Computation

- State transition function

  $f\colon x_n = (\boldsymbol{a} \cdot x_{n-1} + \boldsymbol{c}) \, mod \, \boldsymbol{m} \quad n \in \mathbb{Z}^+$

- Output function

  $g\colon u_n = truncation(x_n)$



Different $\boldsymbol{c}$ can generate distinct but **highly correlated sequences**.

Towards High Quality: Decorrelation

- Inspired by Yao's XOR Lemma
  - Weakly correlated sequences as decorrelators

LCG's variant → low intra stream correlation
Decorrelator → low inter stream correlation

Towards High Quality: Decorrelation

- Inspired by Yao's XOR Lemma
  - Weakly correlated sequences as decorrelators
- Constraints
  - K^1, K^2 are generated from the algorithm that **completely different** from LCG.
  - K^1 is **weakly correlated** with K^2, and there is no overlapped subsequences.

Towards High Quality: Decorrelation

- Inspired by Yao's XOR Lemma
  - Weakly correlated sequences as decorrelators
- Constraints
  - K^1, K^2 are generated from the algorithm that **completely different** from LCG.
  - K^1 is **weakly correlated** with K^2, and there is no overlapped subsequences.
- Algorithm Flow

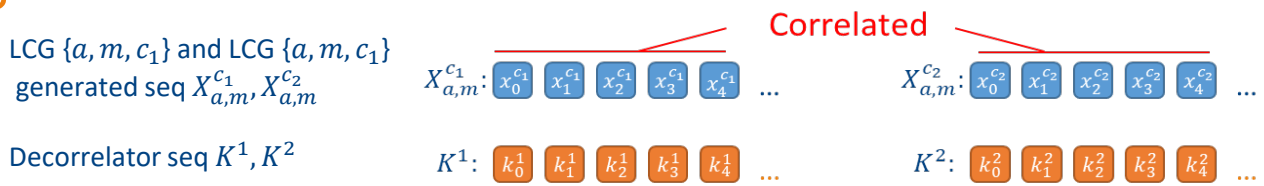
Towards High Quality: Decorrelation

- Inspired by Yao's XOR Lemma
  - Weakly correlated sequences as decorrelators
- Constraints
  - K^1, K^2 are generated from the algorithm that **completely different** from LCG.
  - K^1 is **weakly correlated** with K^2, and there is no overlapped subsequences.
- Algorithm Flow

LCG $\{a, m, c_1\}$ and LCG $\{a, m, c_1\}$ generated seq $X_{a,m}^{c_1}, X_{a,m}^{c_2}$

Towards High Quality: Decorrelation

- Inspired by Yao's XOR Lemma
  - Weakly correlated sequences as decorrelators
- Constraints
  - K^1, K^2 are generated from the algorithm that **completely different** from LCG.
  - K^1 is **weakly correlated** with K^2, and there is no overlapped subsequences.
- Algorithm Flow

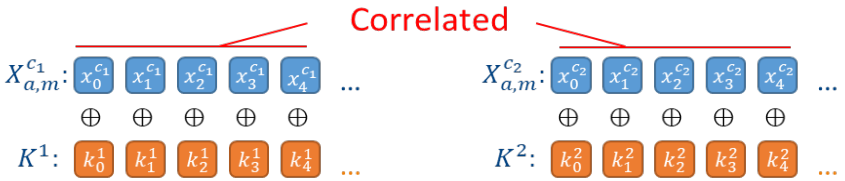


Towards High Quality: Decorrelation

- Inspired by Yao's XOR Lemma
  - Weakly correlated sequences as decorrelators
- Constraints
  - K^1, K^2 are generated from the algorithm that **completely different** from LCG.
  - K^1 is **weakly correlated** with K^2, and there is no overlapped subsequences.
- Algorithm Flow

LCG $\{a, m, c_1\}$ and LCG $\{a, m, c_1\}$ generated seq $X_{a,m}^{c_1}, X_{a,m}^{c_2}$

Decorrelator seq K^1, K^2



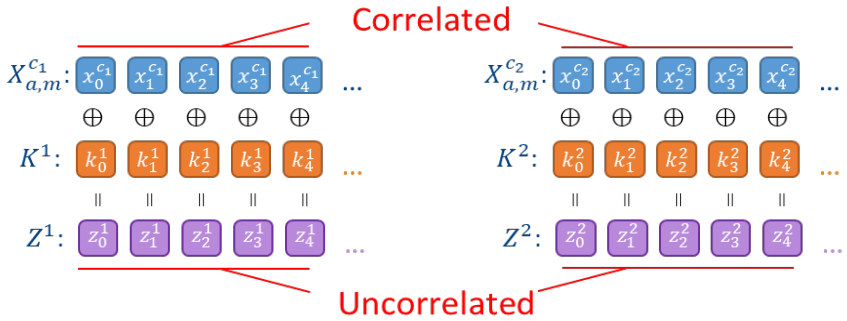
Towards High Quality: Decorrelation

- Inspired by Yao's XOR Lemma
  - Weakly correlated sequences as decorrelators
- Constraints
  - K^1, K^2 are generated from the algorithm that **completely different** from LCG.
  - K^1 is **weakly correlated** with K^2, and there is no overlapped subsequences.
- Algorithm Flow

LCG $\{a, m, c_1\}$ and LCG $\{a, m, c_1\}$ generated seq $X_{a,m}^{c_1}, X_{a,m}^{c_2}$

Decorrelator seq K^1, K^2

Final generated seq Z^1, Z^2



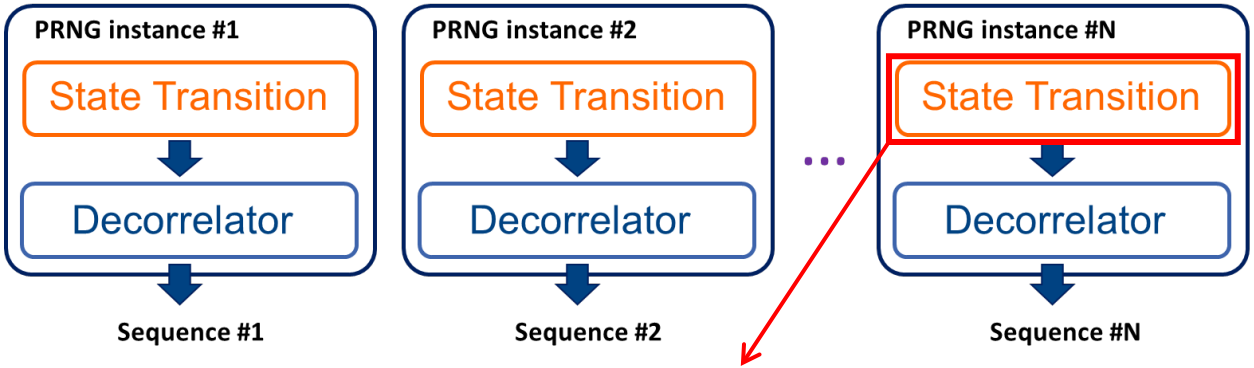
Towards High Quality: Decorrelation

- Proof of Two Sequences Decorrelation
  - Transform the XOR operator to multiplication
  - Construct the representation of correlation coefficient
- Proof of Multiple Sequences Decorrelation
  - Induction proof using the conclusion of two sequences decorrelation
- Validation
  - Empirical tests for inter-stream correlation

→ Solve the Challenge 1

Towards High Throughput

- The number of instances is limited by available DSPs

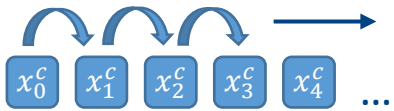


Towards High Throughput: State Sharing

- Revision of LCG

LCG $\{a, m, c\}$ generated seq

$X_{a,m}^{c}$:

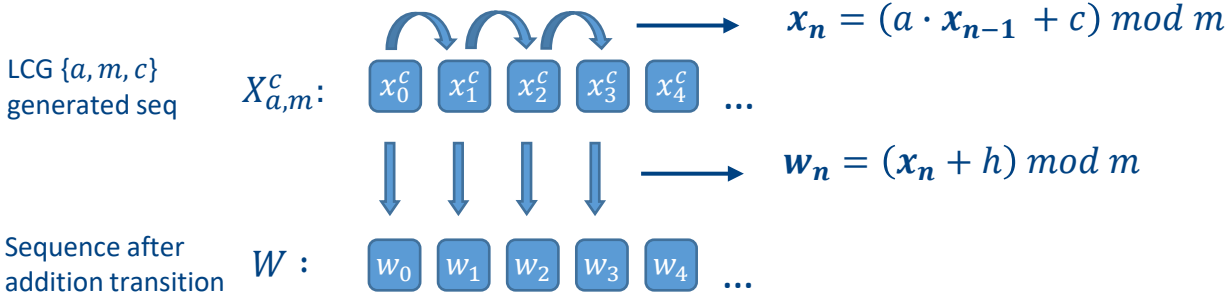


$$\boldsymbol{x_n} = (a \cdot \boldsymbol{x_{n-1}} + c) \; mod \; m$$

Sequence after addition transition

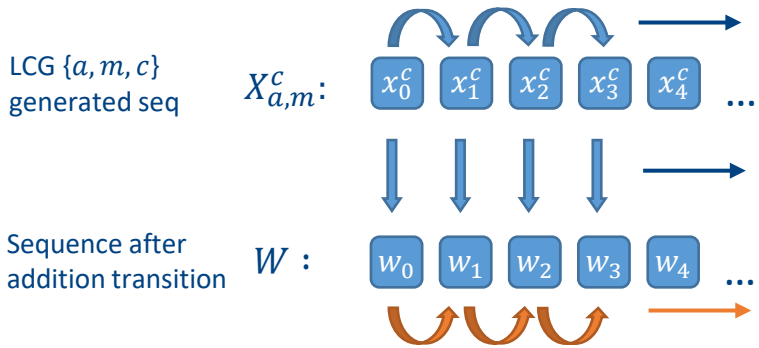
Towards High Throughput: State Sharing

- Revision of LCG
  - Adding an addition transition after LCG transition



Towards High Throughput: State Sharing

- Revision of LCG
  - Adding an addition transition after LCG transition
  - The combined transition is with the same form as the LCG transition



$$\boldsymbol{x_n} = (a \cdot \boldsymbol{x_{n-1}} + c)\ mod\ m$$

$$\boldsymbol{w_n} = (\boldsymbol{x_n} + h)\ mod\ m$$

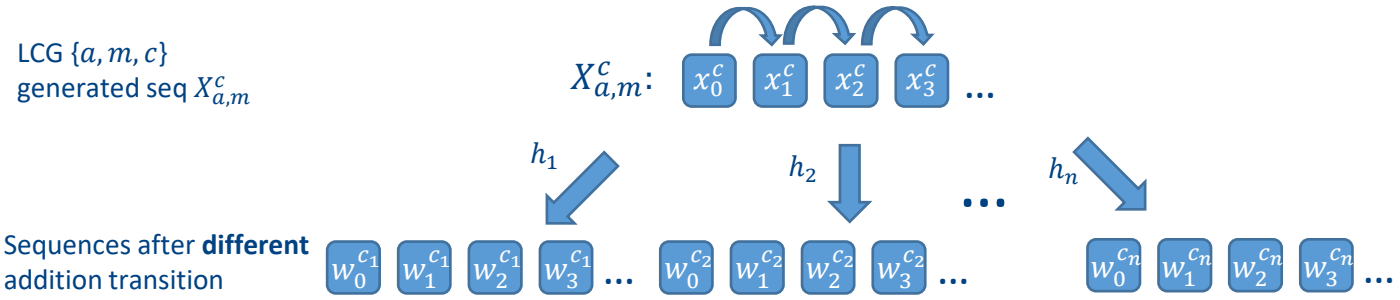
$$\boldsymbol{w_n} = [a \cdot \boldsymbol{w_{n-1}} + (k \cdot m + c - a \cdot h)]\ mod\ m$$

LCG transition with $\{a, m, (k \cdot m + c - a \cdot h)\}$

* h should be an even number (by Hull-Dobell Theorem).

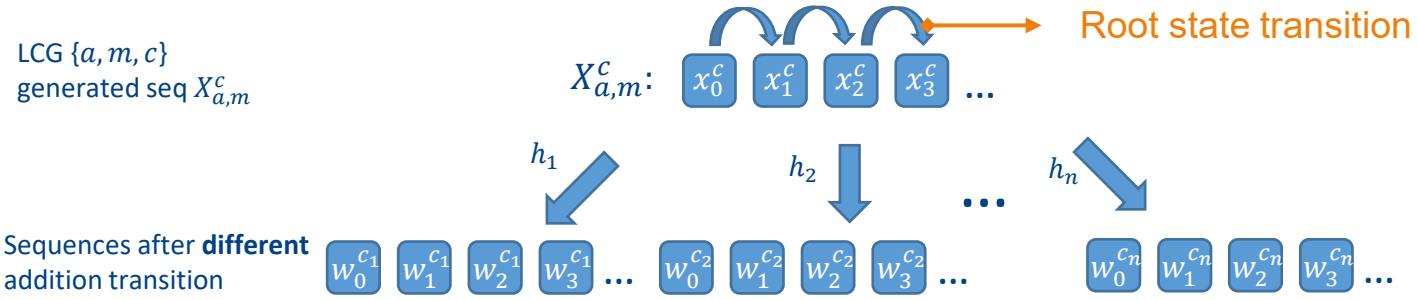
Towards High Throughput: State Sharing

- Revision of LCG
  - Adding an addition transition after LCG transition
  - The combined transition is with the same form as the LCG transition



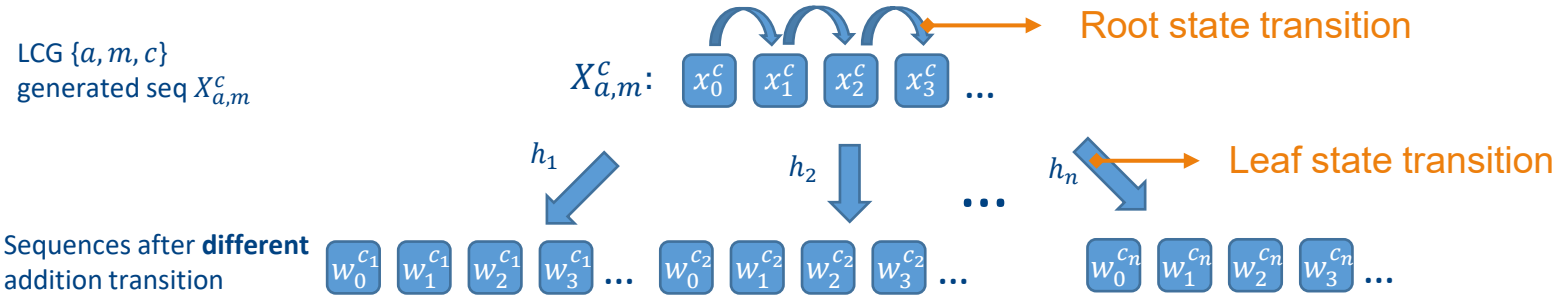
Towards High Throughput: State Sharing

- Revision of LCG
  - Adding an addition transition after LCG transition
  - The combined transition is with the same form as the LCG transition



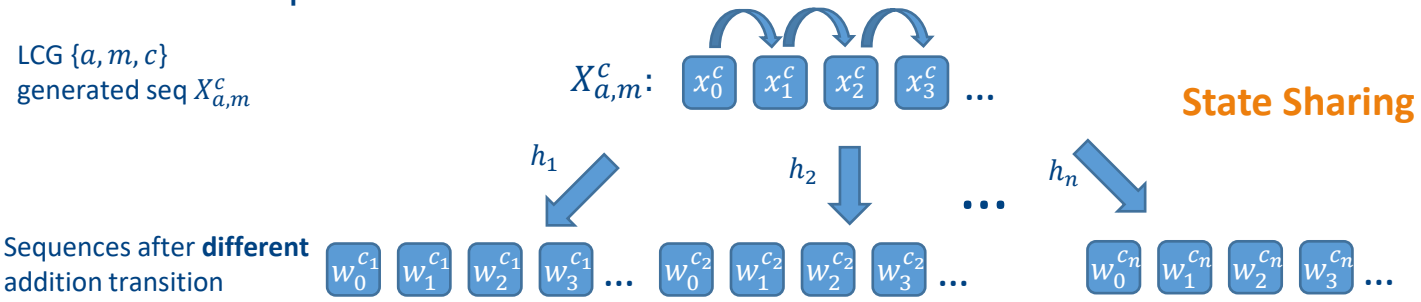
Towards High Throughput: State Sharing

- Revision of LCG
  - Adding an addition transition after LCG transition
  - The combined transition is with the same form as the LCG transition



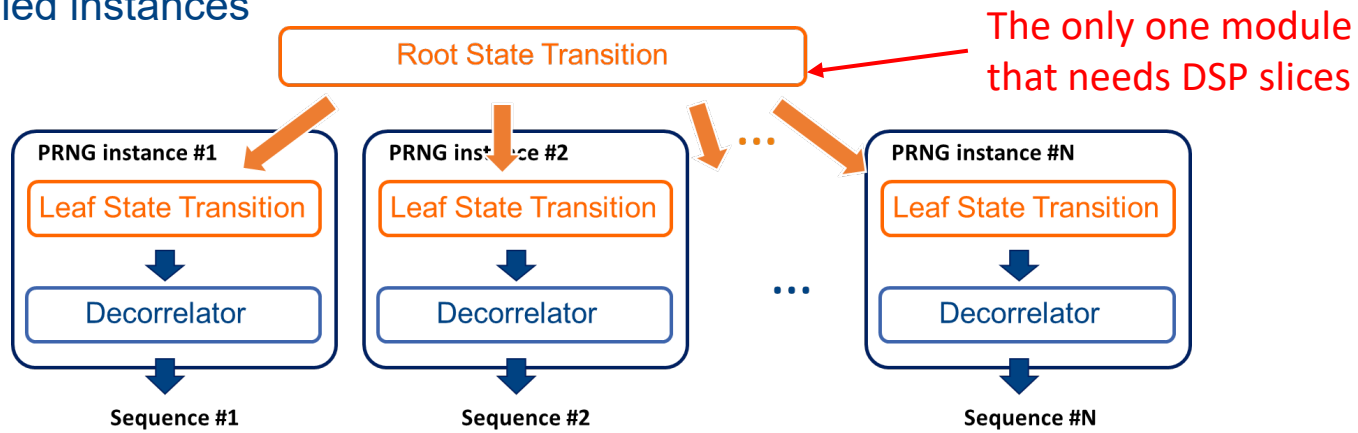
Towards High Throughput: State Sharing

- Share the root state among multiple instances
  - Without state sharing, ***n*** instances need ***n*** times multiplications and ***n*** times additions in each step.
  - With state sharing, ***n*** instances only need ***1*** multiplication and ***(n+1)*** times additions in each step.



Towards High Throughput: State Sharing

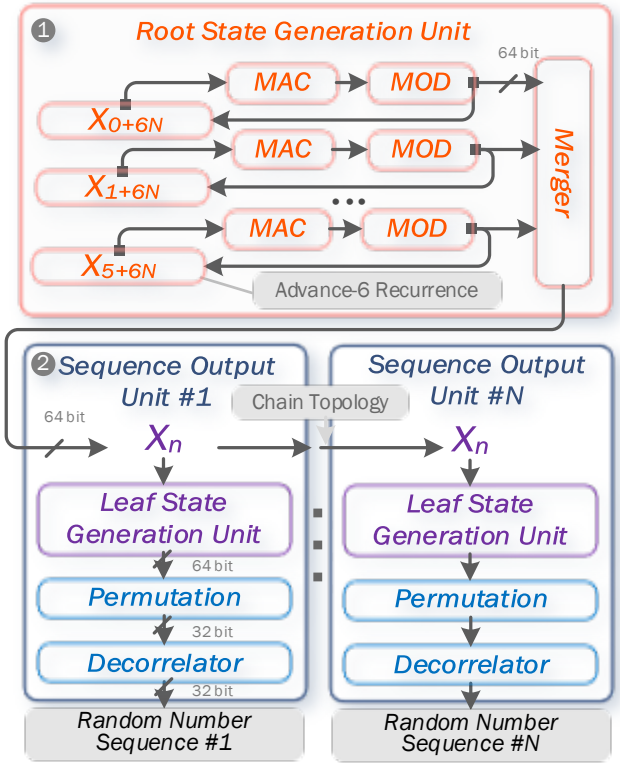
- Share the root state among multiple instances
  - The number of available DSP slices is no longer the limitation of increasing the paralleled instances



→ Solve the Challenge 2

Architecture Overview

- Root State Generation Unit (RSGU)
  - Perform the root state transition
- Sequence Output Unit (SOU)
  - Each SOU generates sequence of random numbers.



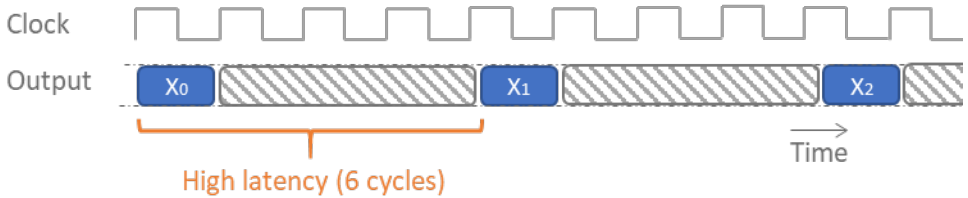
Overview of ThundeRiNG's architecture

Implementation Optimization: Throughput

- Dependency in Root State Generation

  $x_n = (a \cdot x_{n-1} + c)\ mod$

  - **6** cycles multiplication



Implementation Optimization: Throughput

- Dependency in Root State Generation

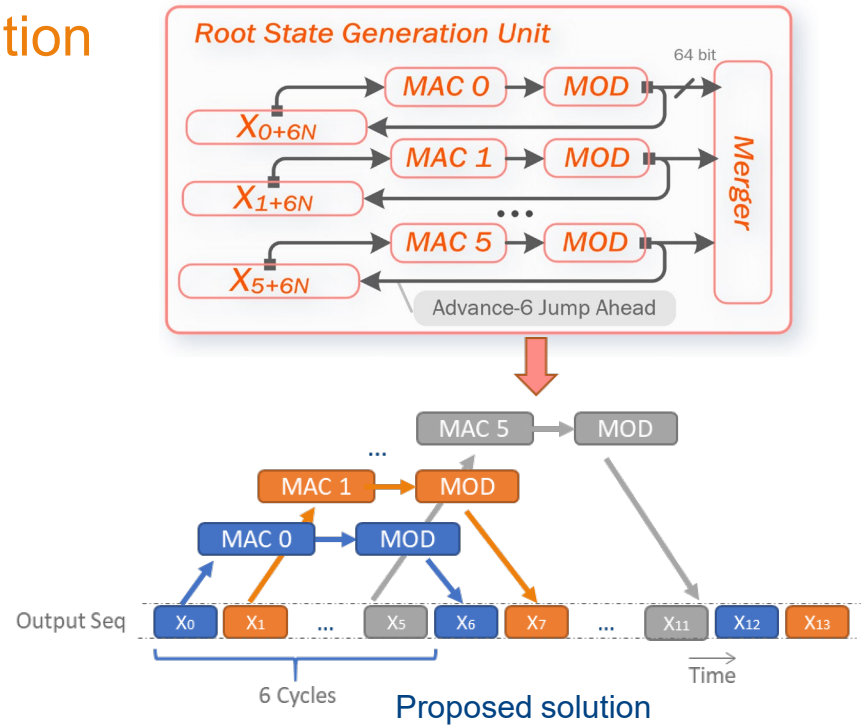
  $x_n = (a \cdot x_{n-1} + c)\ mod$

  - **6** cycles multiplication

- Proposed Solution
  - Recursion between $\boldsymbol{x_n}$ and $\boldsymbol{x_{n-6}}$

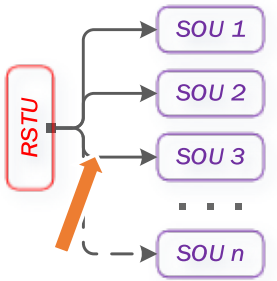
$$\boldsymbol{x_n} = \left(a^6 \cdot \boldsymbol{x_{n-6}} + \frac{c(a^6 - 1)}{a - 1}\right) mod\ m$$

  - Generate **one root state per cycle**



Implementation Optimization: Frequency

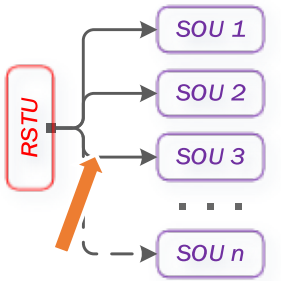
- Sharing root state for massive number of SOUs introduces high fanout problem



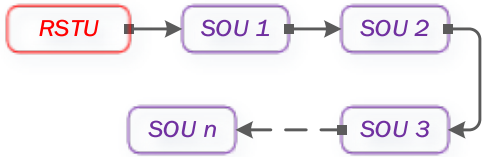
High fanout

Implementation Optimization: Frequency

- Sharing root state for massive number of SOUs introduces high fanout problem



High fanout

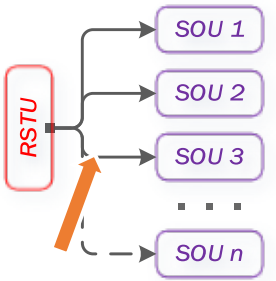


Daisy-chaining topology

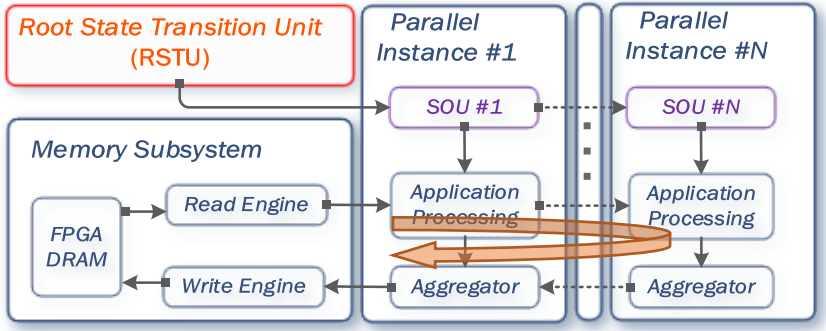
- Using chain topology to connect the SOUs

Implementation Optimization: Frequency

- Sharing root state for massive number of SOUs introduces high fanout problem



High fanout

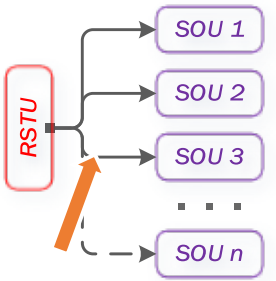


Data path loopback

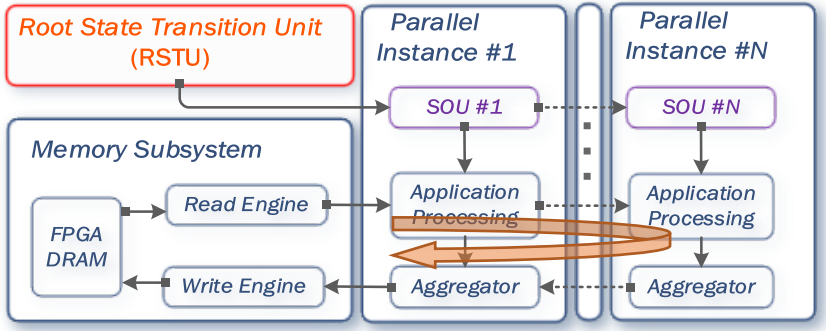
- Using chain topology to connect the SOUs

Implementation Optimization: Frequency

- Sharing root state for massive number of SOUs introduces high fanout problem



High fanout



Data path loopback

- Using chain topology to connect the SOUs
- Frequency is improved from 225 to **421** MHz (512 SOUs)

Experiment Setup

- We implement our design and conduct the comparison with the state-of-the-art works on the following hardware platforms.

	Hardware	#Core	Software Environment
FPGA	Xilinx Alveo U250 accelerator card	-	Vitis HLS Toolchain 2020.1
GPU	NVIDIA Tesla P100 GPU	1 (3584 cores)	CUDA Toolkit 10.1
CPU	Intel Xeon 6248R CPUs	2 (96 cores)	oneAPI MKL 2021.2

Quality Evaluation

- Intra-stream Correlation
  - BigCrush test suit (TestU01)
  - PractRand test suit
- Inter-stream Correlation
  - Testing on the interleaved sequence
  - Pairwise correlation
  - Hamming weight dependency analysis

Quality Tests	
BigCrush	Pass
PractRand	>8TB
BigCrush on Interleaved Seq	Pass
Pearson Correlation	3e-5
Spearman's Rank	3e-5
Kendall's Rank	2e-5
Hamming Weight Dependency	>1e+14

Quality Evaluation

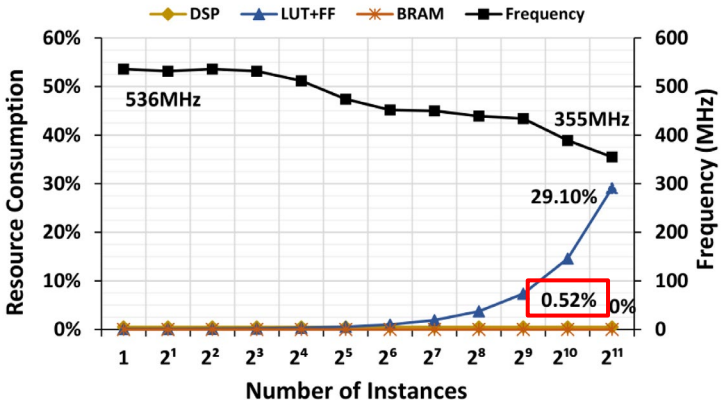
- Intra-stream Correlation
  - BigCrush test suit (TestU01)
  - PractRand test suit
- Inter-stream Correlation
  - Testing on the interleaved sequence
  - Pairwise correlation
  - Hamming weight dependency analysis

Quality Tests		
BigCrush	Pass	✓
PractRand	>8TB	✓
BigCrush on Interleaved Seq	Pass	✓
Pearson Correlation	3e-5	✓
Spearman's Rank	3e-5	✓
Kendall's Rank	2e-5	✓
Hamming Weight Dependency	>1e+14	✓

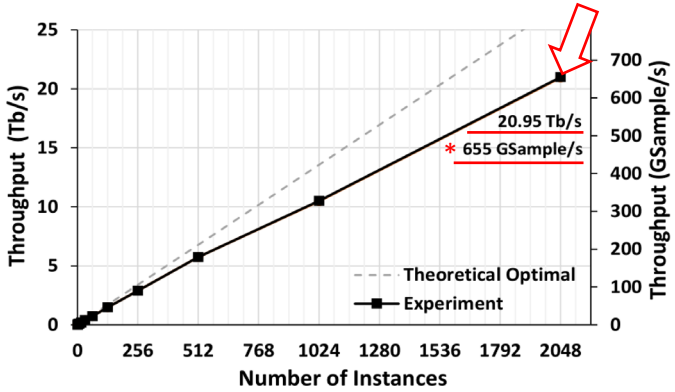
ThundeRiNG passes all the statistical tests.

Resource Utilization and Throughput

- ThundeRiNG uses a **constant** number of DSP slices regardless of the number of sequences to be generated
- The observed throughput is **nearly proportional** to the number of instances and can be up to **20.95 Tb/s** with 2048 instance



Resources consumption and clock frequency with varying number of instances



Throughput with varying number of SOU instances

* 1 sample = 32-bit random number

Comparison with FPGA-based PRNGs

- ThundeRiNG outperforms all current FPGA based PRNGs significantly in **performance** as well as **quality of the randomness**.

PRNGs	Quality	Freq. (MHz)	Max #ins.	BRAM (%)	DSP (%)	Thr. (Tb/s)	Sp.
Implementation Benchmarking:							
ThundeRiNG	**Crush-resistant**	**355**	**2048**	**0%**	**0.5%**	**20.95**	1×
Li et al. [1]	Crushable	475	16	1.6%	0%	0.24	87.08×
LUT-SR [2]	Crushable	600	1	0%	0%	0.37	55.9×
Optimistic Scaling:							
Philox4_32 [3]	Crush-resistant	500	442	0%	100%	2.83	7.39×
Xoroshiro128** [4]	Crush-resistant	500	1150	0%	100%	18.40	1.14×
Li et al. [1]	Crushable	500	1000	100%	0%	16.00	1.37×

Throughput, quality and resource utilization of the SOTA FPGA-based works and CPU-based designs

[1] Li, et al. "Software/hardware parallel long-period random number generation framework based on the well method." VLSI, 2013.
[2] Thomas, et al. "The LUT-SR family of uniform random number generators for FPGA architectures." VLSI 2012.
[3] Salmon, et al. "Parallel random numbers: as easy as 1, 2, 3." SC 2011.
[4] Blackman, et al. " Scrambled linear pseudorandom number generators." arXiv 2018.

Comparison with GPU-based PRNGs

- ThundeRiNG demonstrates a **10.62×** speedup over the state-of-the-art GPU-based implementation.

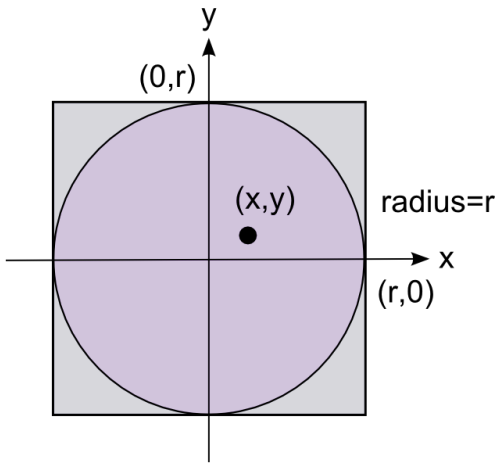
Algorithms (cuRAND)	BigCrush	Throughput GSample/s	Throughput Tb/s	ThundeRiNG's Speedup
Philox-4×32 [1]	Pass	61.6234	1.9719	**10.62×**
MT19937 [2]	Pass	51.7373	1.6556	12.65×
MRG32k3a [3]	1 failure	26.2662	0.8405	**24.92×**
xorwow [4]	1 failure	56.6053	1.8114	11.56×
MTGP32 [5]	1 failure	29.1273	0.9321	22.47×

Throughput of various GPU PRNG schemes running on Nvidia Tesla P100 compared to ThundeRiNG's throughput

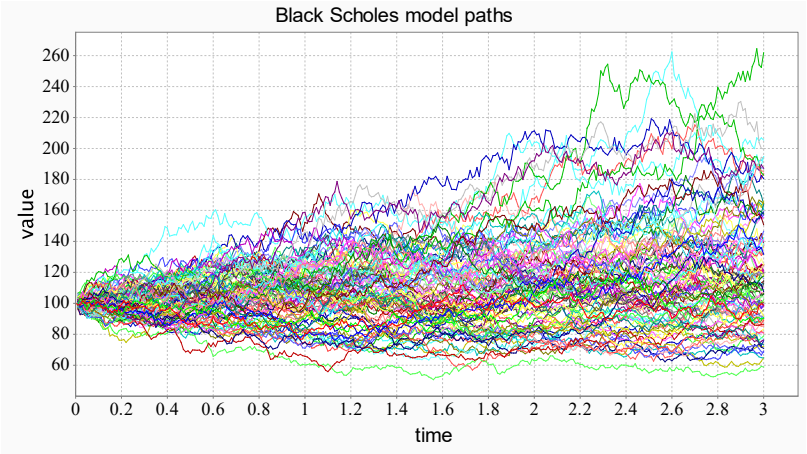
[1] Salmon, et al. "Parallel random numbers: as easy as 1, 2, 3." SC 2011.
[2] Matsumoto, et al. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." TOMACS, 1998.
[2] Pierre L'Ecuyer. "Combined multiple recursive random number generators." Operations research 1996.
[4] George Marsaglia. " Xorshift RNGs." Journal of Statistical Software 2003.
[5] Mutsuo Saito. "A variant of mersenne twister suitable for graphic processors." CoRR 2010.

Case Studies

- Estimation of π
  - Commonly used benchmark

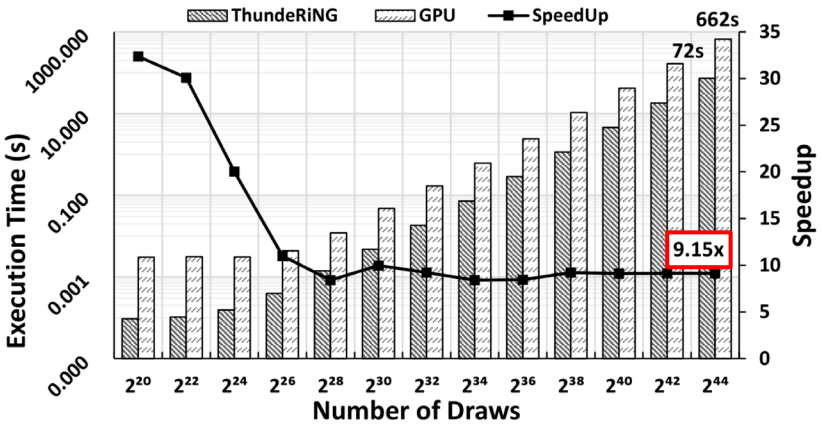


- Monte Carlo option pricing
  - Application in mathematical finance domain



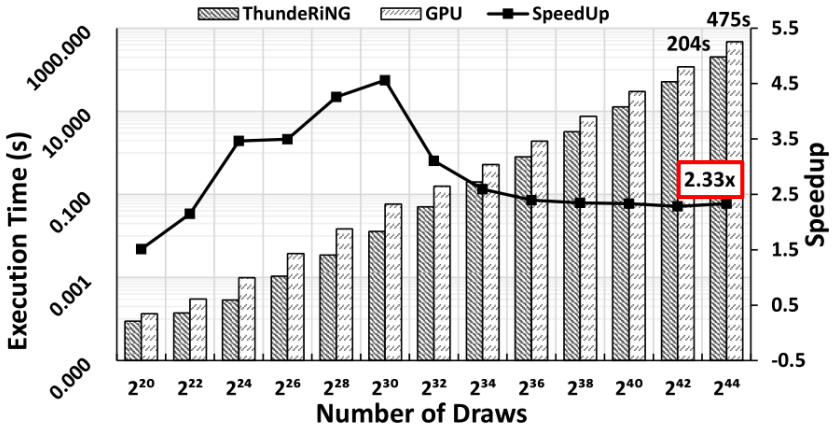
Case Studies

- Estimation of π



Execution time of ThundeRiNG on estimation of π and GPU-based solution with varying number of draws.

- Monte Carlo option pricing



Execution time of ThundeRiNG on Monte Carlo option pricing and GPU-based solution with varying number of draws.

Conclusion

- ThundeRiNG is an FPGA-based PRNG that able to generate multiple independent sequences of random number with high **quality** and high **throughput**.
- ThundeRiNG outperforms all current FPGA based PRNGs significantly in performance as well as quality of the randomness.
- Comparing with the state-of-the-art GPU-based implementation, ThundeRiNG delivers up to $10.62\times$ performance improvement.
- FPGA is a promising platform for PRNG involved applications.
- Source code is available at https://github.com/Xtra-Computing/ThundeRiNG

Acknowledgement

- We thank the anonymous reviewers for their valuable feedback on this work.
- We thank the Xilinx Adaptive Compute Cluster (XACC) Program for the generous donation.
- This work is supported by MoE AcRF Tier 1 grant (T1 251RES1824), Tier 2 grant (MOE2017-T2-1-122) in Singapore, and also partially supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.