# Massively Parallel Random Number Generators

Holger Dammertz | September 15, 2010

# Overview

## Applications for Pseudo-Random Numbers

- Monte Carlo Simulation, Integration
- Test and Content Generation
- These applications are often easily parallelizable
  - MRIP paradigm: multiple replications in parallel

## Generating Pseudo-Random Numbers

- Generate i.i.d. uniform random numbers (for example 32bit)
- Transform into $(0, 1)$
- Additional transformation to target distribution (for example, normal distributed)

# Structure of a RNG

## Formal Definition [L'E06]

$$(S, \mu, f, U, g)$$

- $S$ state space
- $\mu$ prob. distr. on $S$ to select initial state (seed) $s_0 \in S$
- $f : S \to S$ transition function
- $g : S \to U$ output function
- $U = (0,1)$ output space
- $s_i = f(s_{i-1}), i \geq 1$ and $u_i = g(s_i)$

# Structure of a RNG

## In a parallel Implementation

$$(S, \mu, f, U, g)$$

- $S$ needed per generating stream (usually per thread)
- $S$ if possible hold in fast memory
- $S$ store in global memory after finishing for multiple calls
- $f$ and $g$ are device functions (usually in a single function)
- output space often `unsigned int` $\rightarrow$ transform needed

## Example: Linear Congruence Generator LCG [Knu81]

- $s_i \in S$ is an integer (for example 32bit), $U = S, g = id$
- $f : s_i = (as_{i-1} + c) \bmod m$
- Needs well chosen $a$, $c$ and $m$

# Structure of a RNG

## Required properties of a RNG

- Speed
- Repeatability
- Minimal statistical bias

## Additional properties

- Random access on $u_i$
- Independent number streams
- Long period

# Structure of a RNG

## Why is a long period important?

From [SPM05]: for a cycle length of $n$ a single simulation should use at most

$$16\sqrt[3]{n}$$

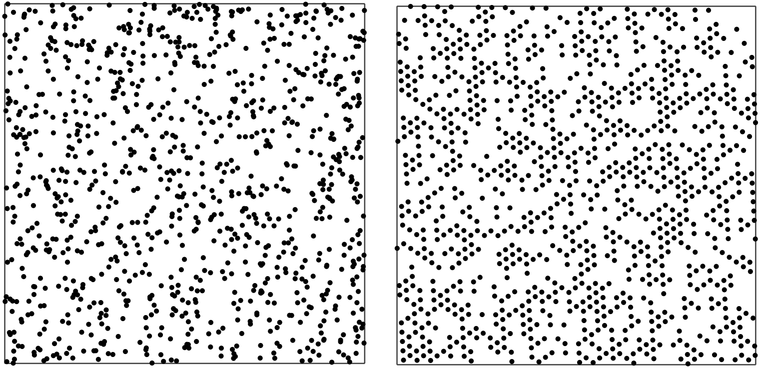
random numbers (to trust the results of statistical simulation).
Assume period of $2^{48}$ and simple parallel cuda app with 4096 threads:

$$\frac{16\sqrt[3]{2^{48}}}{4096} \approx 256$$

random numbers per thread.

# Choice of RNG parameters are important

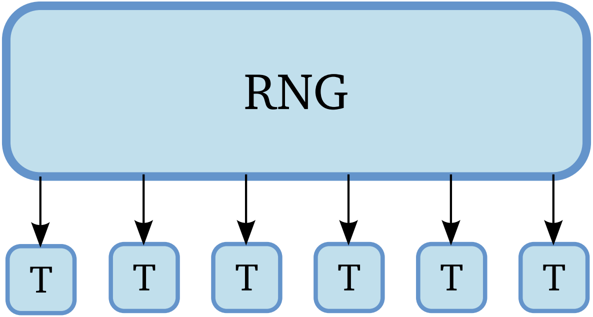
Simple LCG, $2^{10} - 3$ points



# Two different views on parallel random numbers

## 1) Single Stream for all Threads

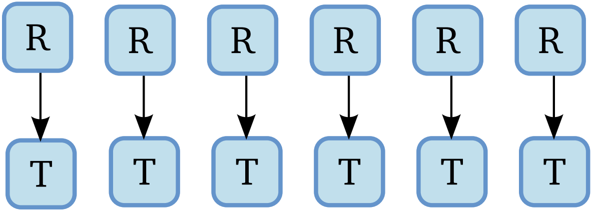
- Each thread computes parts of one problem
- Result should not depend on number of threads and should be repeatable
- Ideally RNG update is also parallel ($\rightarrow$ speed)



# Two different views on parallel random numbers

## 2) Each Thread uses it's own RNG

- Each thread computes individual solutions
- Need to guarantee independence of streams




- If you could have a true RNG both methods would behave exactly the same (but repeatability would be lost).
- Using Pseudo RNGs this needs to be explicitly designed in the program

# Parallelizing RNGs

## Random Seeding

- Easy to implement
- Generally very bad parallelization method
  - Need to seed valid states
  - No guarantee of independence
- Can work for generators with a long period
- Better alternative: well chosen seeds (per thread)
  - For example: Mersenne Twister dcmt library
  - New GPU Mersenne Twister (MTGP) provides seed tables

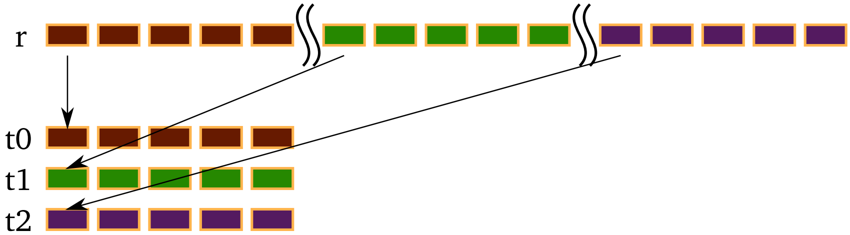
## Parametrization

- $n$ different RNG configurations for $n$ threads
- Needs to be especially developed and tested
- Often restricted to specific $n$

# Parallelizing RNGs

## Block Splitting

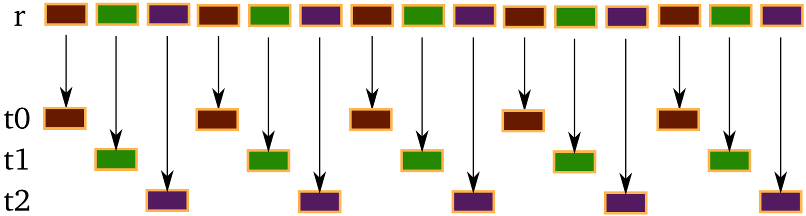
- Can guarantee non overlapping sequences of length $m$
- $m$ needs to be known in advance
- Starting states need to be known or computed



# Parallelizing RNGs

## Leap-Frogging

- Needs RNG to be able to skip *n* numbers (or else quite inefficient)



# Application Design Considerations

## On the fly Computation

Compute RN when needed in kernel

- State needs to be stored per stream
- RNG uses additional resources (registers and memory)
- Needs properly parallelized implementation

## Pre-computation

Store RN in main memory

- Only memory access per RN
- Easily parallelizable (same access as leap frog)
- Memory requirements can be huge
- Bandwidth need to be considered

# Upload vs. Computation Example

Intel(R) Xeon(R) E5420 @ 2.50GHz, GTX 480, 256MB random numbers, Mersenne Twister: MTGP/SFMT, $2^{14}$ threads, blocksize 256

## Precomputation

- Upload CPU-computed RN
  - Precompute on CPU (SFMT): 180ms
  - Upload: 44ms
- Precompute on GPU
  - Precompute on GPU (MTGP): 9.18ms
- Consume (Asian Options): 783.5ms

## Produce and consume (MTGP)

- Single Kernel: 1058.1ms

# Overview of some RNGs

## LCGs

$$s_i = (a s_{i-1} + c) \bmod m$$

- Combining multiple LCGs can give longer period
- Independent streams: Wichmann-Hill (273 threads)

## Multiple Recursive Generator

$$s_i = \left( \sum_{\xi=1}^{k} a_\xi s_{i-\xi} \right) \bmod m$$

- Larger period (for k=1 equal to LCG)
- Blocking: MRG32k3a [LSCK02]

# Overview of some RNGs

## RNGs based on Cryptographic functions

- Creates white noise from input
- MD5 [TW08]: hash function
- Tiny Encryption Algorithm (TEA) [ZOC10]
- Different configuration per thread (parametrization)
  - Transform counter and thread id
- Random Access in the sequence

## Mersenne Twister

- New GPU version [Sai10]
- Long period: 32bit version provides $2^{11213}-1, 2^{23209}-1, 2^{44497}-1$
- Good seeding strategies (see MTGPDC)

# Testing RNGs

## Why use tests

- Implementation of RNGs is very sensitive
- Before using any RNG implementation it should be tested
- Failed tests: very likely bad sequence
- Passed tests: guarantees nothing

## Test Suits

- Use statistical tests to find flaws
- DIEHARD + NIST (both integrated in DIEHARDER [Bro09])
- TestU01 [LS07]

# Collection of Parallel Random Number Generators

## Selecting suitable RNGs

- Domain specific problem
- Depends on current compiler and hardware

## Our Collection

- CUDA implementation of several different RNGs
- Easy to use
- Currently tested on Linux
- Most of the code: MIT License
- Copy-paste ready code

## Available at

`http://mprng.sf.net`

# Collection of Parallel Random Number Generators

## Integrated Benchmark

- Benchmarks compiles and runs on your machine
- Configure threads and blocks according to your target application

## Integrated Test Suits

- DIEHARDER and TestU01
- Automatic report generation

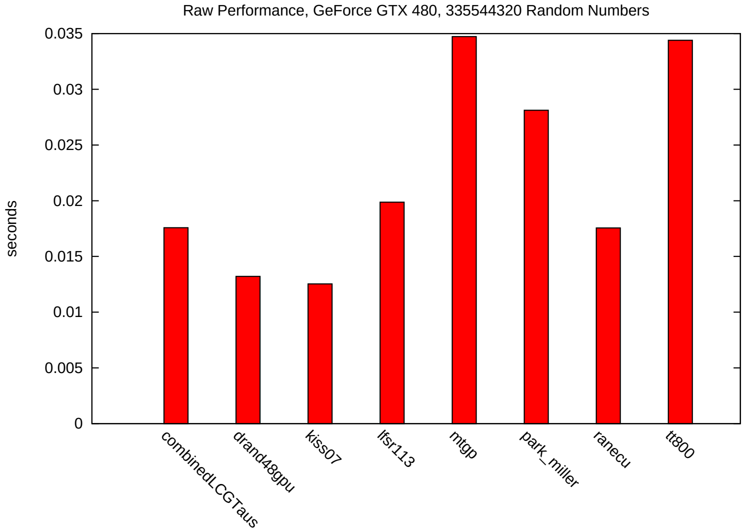
## Easy to extend

- Integrate your own RNG
- Integrate your own tests

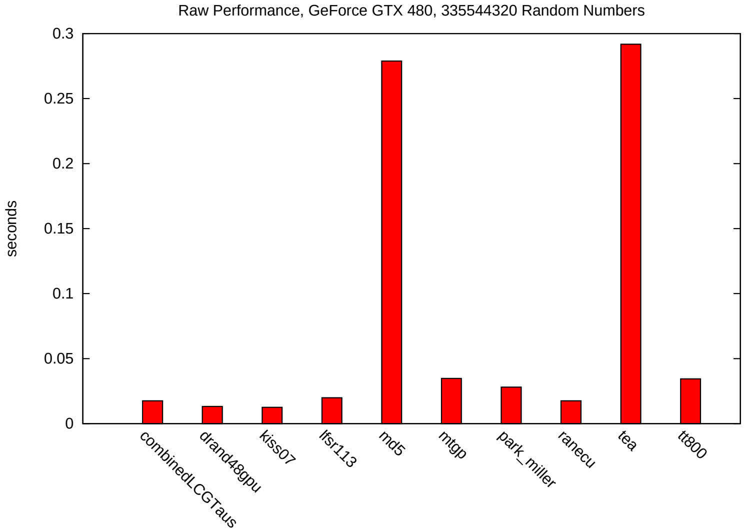
# Comparisons: TestU01 SmallCrush Battery

| RNG Name | BirthdaySpacings | Collision | Gap | SimpPoker | CouponCollector | MaxOft | MaxOft AD | WeightDistrib | MatrixRank | HammingIndep | RandomWalk1 H | RandomWalk1 M | RandomWalk1 J | RandomWalk1 R | RandomWalk1 C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| combinedLCGTaus | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |
| drand48gpu | P | P | P | P | P | F | F | P | P | P | F | F | F | F | P |
| kiss07 | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |
| lfsr113 | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |
| md5 | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |
| mtgp | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |
| park_miller | F | F | P | P | P | F | F | P | P | P | F | F | F | F | F |
| ranecu | P | F | P | P | P | F | F | P | P | P | F | F | F | F | F |
| tea | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |
| tt800 | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P |

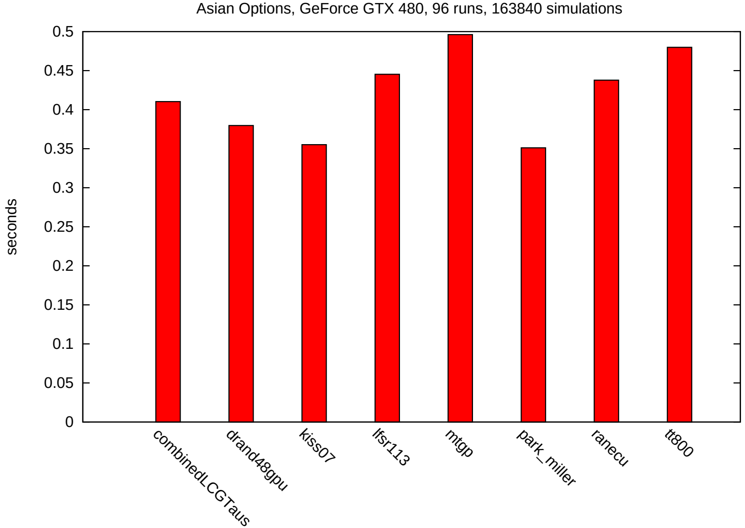
# Raw Performance Test



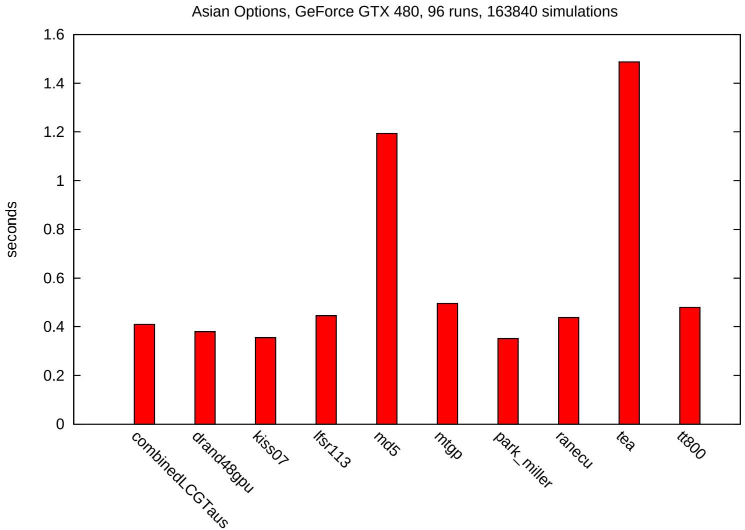
# Raw Performance Test



# Performance Test: Asian Option Example from [HT07]



# Performance Test: Asian Option Example from [HT07]

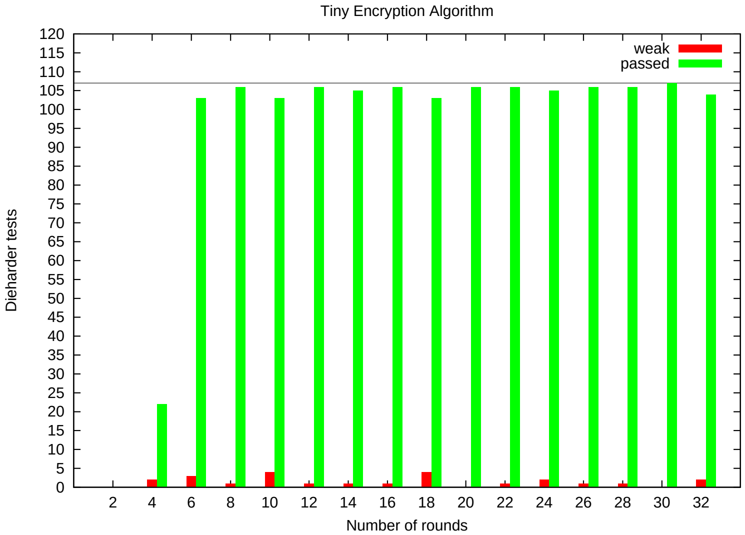


# Quality vs. Speed

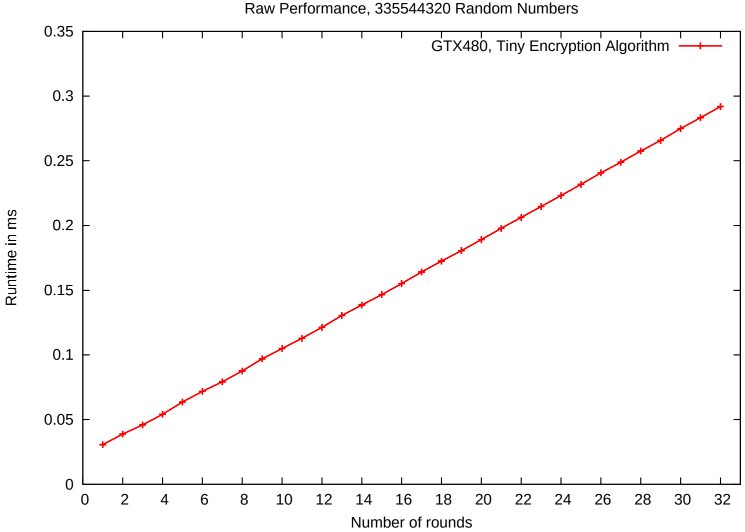
## Reduced rounds of MD5/TEA RNG

- For some applications speed more important than quality
- MD5 and TEA allow to reduce number of iterations
  - Quality of random numbers may degrade
  - Avalanche effect
- MD5: passes most of the DIEHARDER tests after 16 rounds

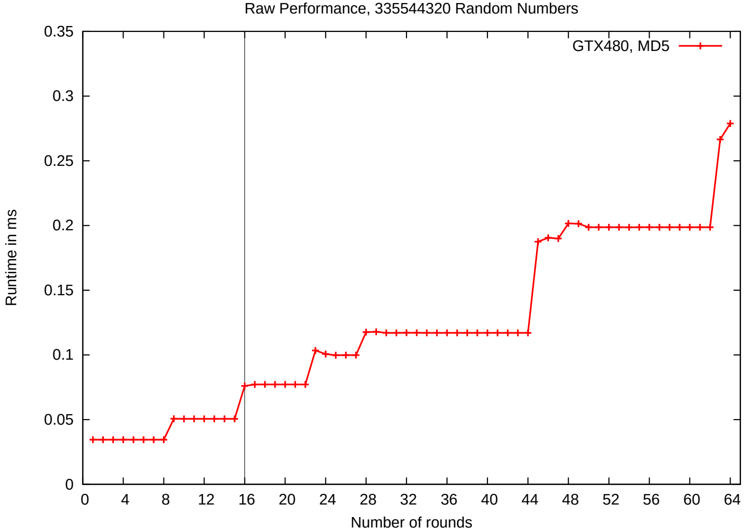
# Quality vs. Speed, TEA, DIEHARDER tests



# Quality vs. Speed, TEA performance



# Quality vs. Speed, MD5 performance



# Conclusion

- Many good (parallel) RNGs exist
- Several different properties
- Choice of fitting RNG application dependent

## Some Picks

- KISS
  - \+ Simple code and state management
  - − Random seeding: may be ok for non-critical applications
- MTGPU
  - \+ Very good quality and sophisticated seeding
  - \+ Long period
  - − Relatively complex code
  - − Fixed block/thread layout
- MD5, TEA
  - \+ Random access
  - \+ No Seeding
  - − Slow

# Conclusion

## Precomputing on GPU

- May be an alternative to in kernel computation

## RNG Collection

- Always evaluate your RNG choice and implementation
- Our framework provides an easy platform for testing
  `http://mprng.sf.net`

# Conclusion

## Precomputing on GPU

- May be an alternative to in kernel computation

## RNG Collection

- Always evaluate your RNG choice and implementation
- Our framework provides an easy platform for testing
  `http://mprng.sf.net`

# Questions?


## Acknowledgements

- Framework: *Christoph Schied*
- This work was supported by a NVIDIA Professor Partnership Award with Hendrik P.A. Lensch

Robert G. Brown.
Dieharder: A random number test suite.
http://www.phy.duke.edu/~rgb/General/dieharder.php, 2009.

Lee Howes and David Thomas.
Efficient Random Number Generation and Application Using CUDA.
In *GPU Gems 3*, pages 805–830, 2007.

D.E. Knuth.
The art of computer programming: Seminumerical algorithms, volume 2, 1981.

P. L'Ecuyer.
Uniform random number generation.
*Handbooks in Operations Research and Management Science*, 13:55–81, 2006.

P. L'Ecuyer and R. Simard.
TestU01: AC library for empirical testing of random number generators.
*ACM Transactions on Mathematical Software (TOMS)*, 33(4):22, 2007.

Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton.
An object-oriented random-number package with many long streams and substreams.

*Oper. Res.*, 50(6):1073–1075, 2002.

Mutsuo Saito.
A variant of mersenne twister suitable for graphic processors.
*CoRR*, abs/1005.4973, 2010.

Marcus Schoo, Krzysztof Pawlikowski, and Donald C. Mcnickle.
A survey and empirical comparison of modern pseudo-random number generators for distributed stochastic simulations, 2005.

S. Tzeng and L.Y. Wei.
Parallel white noise generation on a GPU via cryptographic hash.
In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 79–87. ACM, 2008.

F. Zafar, M. Olano, and A. Curtis.
GPU Random Numbers via the Tiny Encryption Algorithm.
2010.