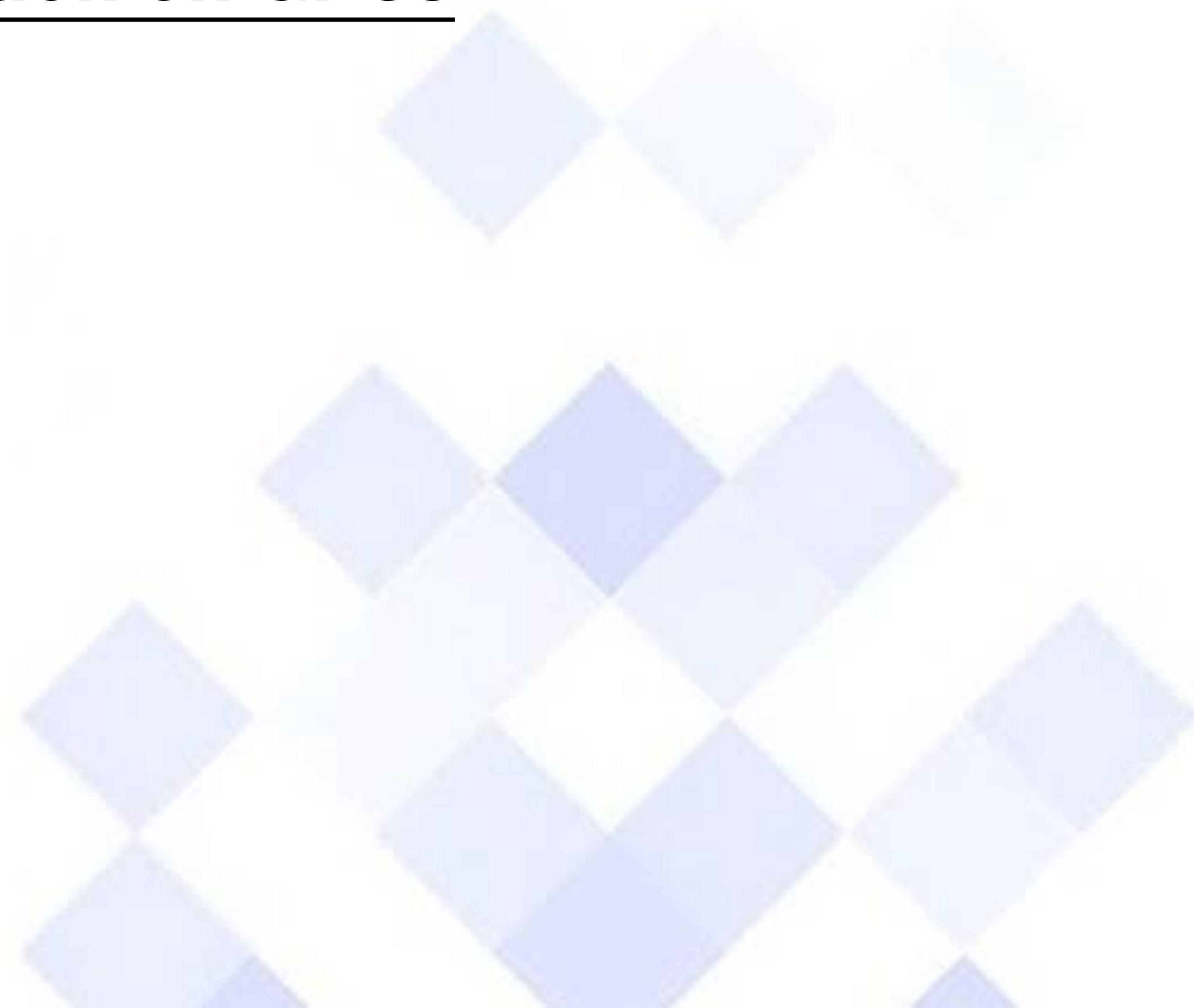
Random Number Generation on GPUs

István Kiss · STREAM HPC



STREAM
High Performance Computing

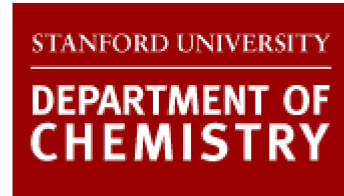
Company

GPU-software development company, working for companies in mostly N-America and W-Europe.

Partners and memberships:

Gromacs
fast
flexible
free

The University of Manchester



Few of the current and previous customers:

We do the **development** part of R&D.
We make **fast** software.

Abstract

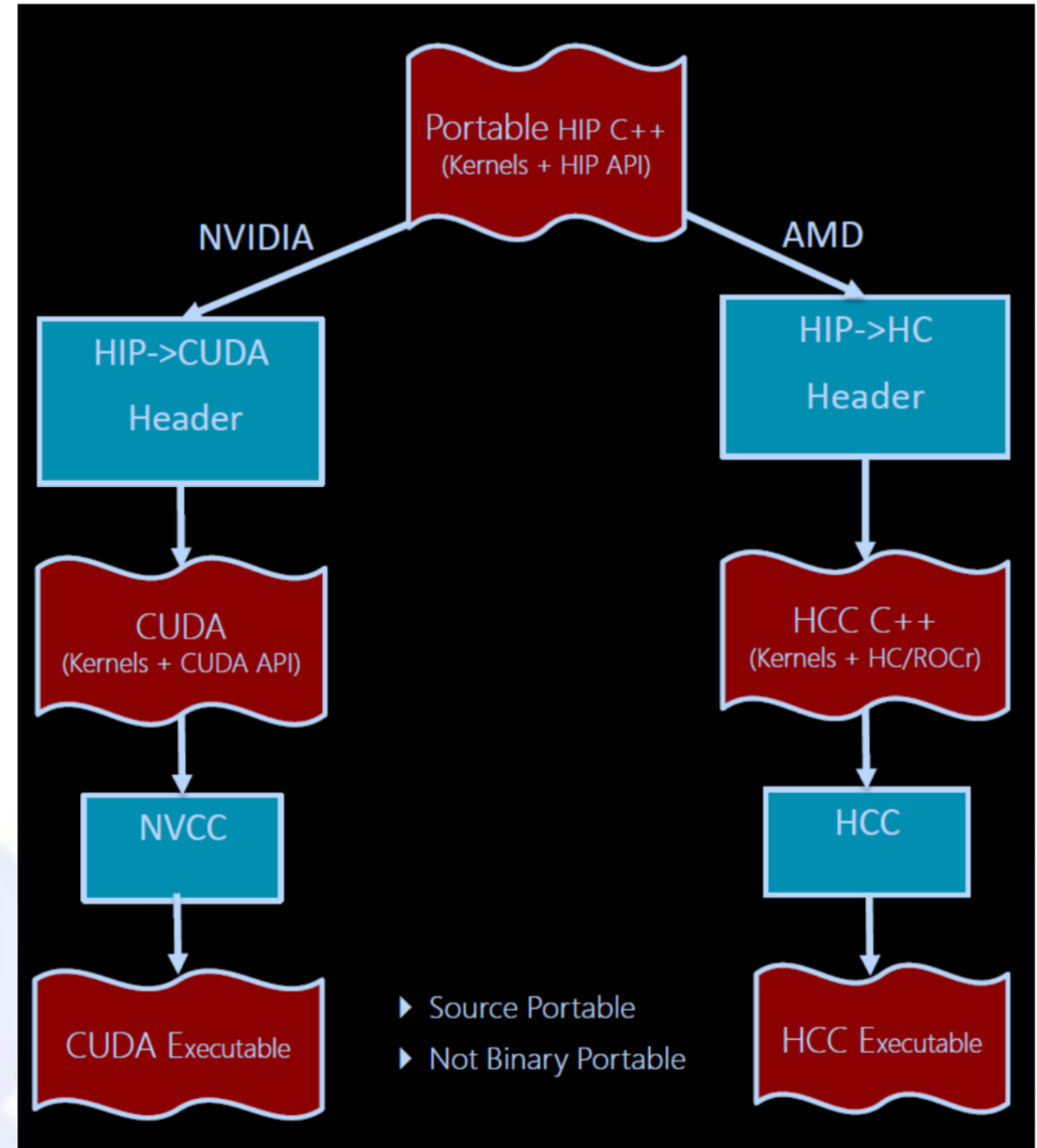
When a developer needs a lot of random numbers, he/she looks for support in the used language and uses it with default settings - after "seeding" it of course, which was mentioned somewhere. Ask any developer the difference between a real, pseudo and quasi random number and you get a blank face with a few exceptions.

In this talk we discuss what random numbers are, what are the differences between popular generators like XORWOW, Mersenne Twister, Philox and Sobol, and how we created and optimized AMD's RNG GPU library. As we worked with both Nvidia and AMD GPUs, we'll also explain some key differences we encountered while developing the library.

The goal of this talk is that you never look the same at a random number again and get insights in how modern AMD GPUs compare to Nvidia GPUs.

The ROCm platform - development tools and libraries

- ROCm Development Tools:
    - HCC compiler
    - HIP
    - ROCm Device Libraries
    - ROCm OpenCL, which is created from the following components:
        - ROCm OpenCL Runtime
        - ROCm OpenCL Driver
    - ...
    - Example Applications:
        - HCC Examples
        - HIP Examples
- ROCm libraries:
    - rocRAND
    - rocPRIM
    - rocThrust
    - hipCUB
    - ...



HIP compilation hierarchy
http://gpgpu10.athoura.com/ROCMGPGPUKeynote.pdf

STREAM
High Performance Computing

The ROCm platform - supported devices

ROCm officially supports AMD GPUs that use following chips:

- GFX8 GPUs
  - "Fiji" chips, such as on the AMD Radeon R9 Fury X and Radeon Instinct MI8
  - "Polaris 10" chips, such as on the AMD Radeon RX 580 and Radeon Instinct MI6
  - "Polaris 11" chips, such as on the AMD Radeon RX 570 and Radeon Pro WX 4100
  - "Polaris 12" chips, such as on the AMD Radeon RX 550 and Radeon RX 540
- GFX9 GPUs
  - "Vega 10" chips, such as on the AMD Radeon RX Vega 64 and Radeon Instinct MI25
  - "Vega 7nm" chips, such as on the Radeon Instinct MI50, Radeon Instinct MI60 or AMD Radeon VII

Supported CPUs for these GPUs

Current CPUs which support PCIe Gen3 + PCIe Atomics are:

- AMD Ryzen CPUs;
- The CPUs in AMD Ryzen APUs;
- AMD Ryzen Threadripper CPUs
- AMD EPYC CPUs;
- Intel Xeon E7 v3 or newer CPUs;
- Intel Xeon E5 v3 or newer CPUs;
- Intel Xeon E3 v3 or newer CPUs;
- Intel Core i7 v4, Core i5 v4, Core i3 v4 or newer CPUs (i.e. Haswell family or newer).
- Some Ivy Bridge-E systems

The ROCm platform - hipify tool

CUDA

```
/*
* Square each element in the array A and write to array C.
*/
template <typename T>
__global__ void
vector_square(T *C_d, const T *A_d, size_t N)
{
    size_t offset = (blockIdx_x * blockDim_x + threadIdx_x);
    size_t stride = blockDim_x * gridDim_x ;

    for (size_t i=offset; i<N; i+=stride) {
        C_d[i] = A_d[i] * A_d[i];
    }

  }
```

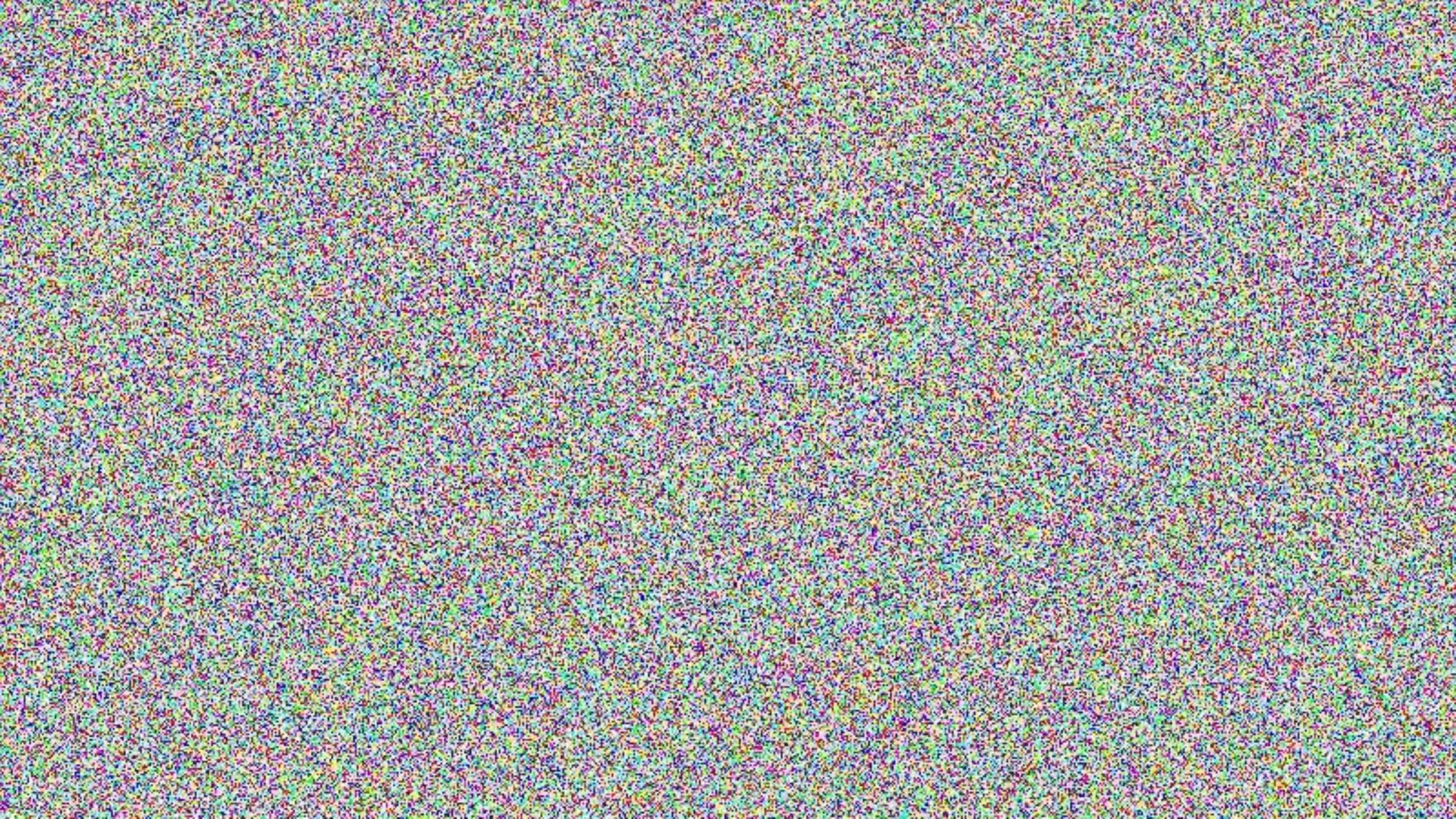
hipify tool + one manual edit

HIP

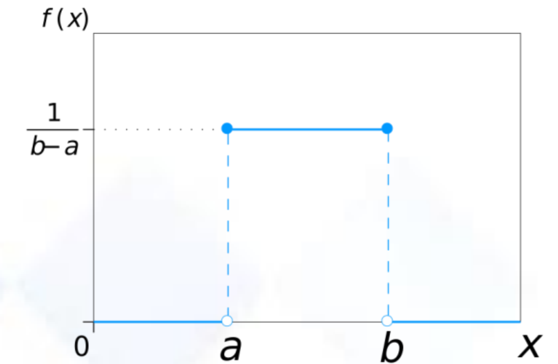
```
/*
* Square each element in the array A and write to array C.
*/
template <typename T>
__global__ void
vector_square(hipLaunchParm lp, T *C_d, const T *A_d, size_t N)
{
    size_t offset = (hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x);
    size_t stride = hipBlockDim_x * hipGridDim_x ;

    for (size_t i=offset; i<N; i+=stride) {
        C_d[i] = A_d[i] * A_d[i];
    }

  }
```

Hipified host code

```
printf ("info: allocate device mem (%6.2f MB)\n",
2*Nbytes/1024.0/1024.0);
CHECK(hipMalloc(&A_d, Nbytes));
CHECK(hipMalloc(&C_d, Nbytes));
printf ("info: copy Host2Device\n");
CHECK ( hipMemcpy(A_d, A_h, Nbytes, hipMemcpyHostToDevice));
const unsigned blocks = 512;
const unsigned threadsPerBlock = 256;
printf ("info: launch 'vector_square' kernel\n");
hipLaunchKernel(HIP_KERNEL_NAME(vector_square), dim3(blocks),
dim3(threadsPerBlock), 0, 0, C_d, A_d, N);
printf ("info: copy Device2Host\n");

CHECK ( hipMemcpy(C_h, C_d, Nbytes,
```



Introduction



What is a random number (sequence)?

Values are uniformly distributed.
Future values are not dependent on present or past ones.

Usage of random numbers:

Lottery Cryptography Simulations / Numerical Analysis / Monte Carlo methods

Sources of random numbers:

Physical generators:

Non-deterministic, real random
Slow
Needs maintenance, reliability issues
Not reproducible
May introduce vulnerabilities into cryptography solutions

Tables:

Can be fast
Limited quantity
Storage requirements

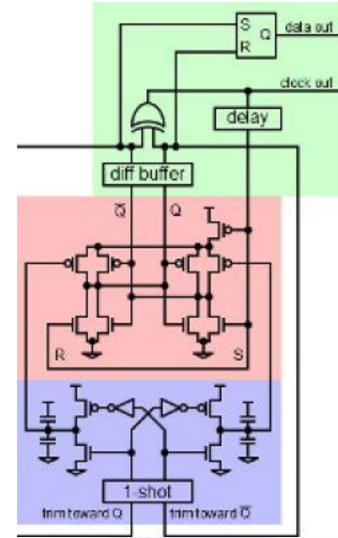
Algorithmic generators:

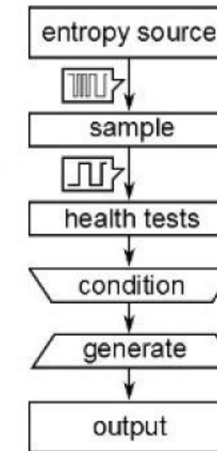
Deterministic: pseudo-randomness
Fast
Reproducible

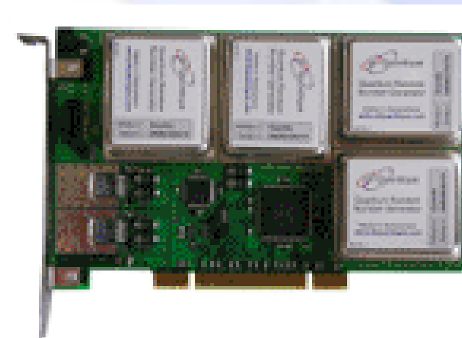
Physical RNGs

Many different possible “HW entropy sources”:

Thermal noise of diodes or resistors (8-bit Ataris, Ivy Bridge)
Diode shot noise
Radioactive decay (fourmilab.ch/hotbits)
Fluctuation of clock ticks
Photonic processes (ID Quantique)
Lava lamp
Clouds
Atmospheric noise (random.org)
Lottery result



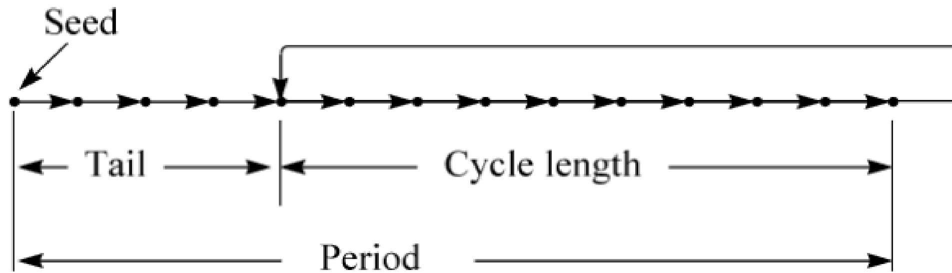




Pseudo-Random Number Generators

"Appears" to be random, but it isn't.

Statistical randomness: it has to be indistinguishable from true random by statistical checks.



What we aim for:

- Uniformity and other statistical properties
- Long cycle length
- Independency
- Low computational cost

Normal Numbers

Borel, 1909:

In normal numbers digits appear in uniform distribution in arbitrary number system.

Almost all real numbers are normal. (Lebesgue-measure of non-normal numbers is zero.)

Suspected normal numbers: π, e, sqrt(2), ln(2) (not proved)

Proved normal numbers:

Champernowne number: 0.12345678910111213…

Copeland-Erdős constant: 0.23571113171923 29… (primes)

Chaitin constant: 0.0078749969978 12384

Conjecture: all irrational algebraic numbers are normal. (Bailey and Crandall, 2001)

Some normal numbers can be used as pseudo-random sequences.

STREAM
High Performance Computing

Pseudo-Random Number Generators

Middle-Square, Neumann, 1946:

Used in the first Monte Carlo simulations on ENIAC (reading tables

took too much time)

Neumann's suggestion for RNG:

$$x_{n+1} = middle_k_digits_of\ (x_n{}^2)$$

+ fast
- limited period length
- gradually accumulates zeros

i	Z_i	Z_i^2
0	**7182**	51581124
1	5811	33767721
2	7677	58936329
3	9363	87665769
4	6657	44315649
5	3156	9960336
6	9603	92217609
7	2176	4734976
…	…	…

Linear Congruence (LCG), Lehmer, 1949:

$$x_{n+1} = (ax_n + c) \bmod m$$

The most commonly used method.
Due to modulo *m*, the number of different values is *m*.
Period length cannot be larger than *m.*

Theorem (Hull and Dobell 1962)
LCG has full period if and only if the following three conditions hold.

1. *m and c are relative primes.*
2. *If q is a prime number that divides m, then q divides a-1.*
3. *If 4 divides m, then 4 divides a-1.*

Pseudo-Random Number Generators

Linear Congruence (LCG), Lehmer, 1949:

$a = 5$
$c = 3$
$m = 16$
$x_0 = 7$

i	x_i	u_i
0	7	0.4375
1	6	0.375
2	1	0.0625
3	8	0.5
4	11	0.6875
5	10	0.625
6	5	0.3125
7	12	0.75
…	…	…

$x_1 = (5*x_0 + 3) \bmod 16 = 38 \bmod 16 = 6$
$x_2 = (5*x_1 + 3) \bmod 16 = 33 \bmod 16 = 1$
$x_3 = (5*x_2 + 3) \bmod 16 = 8 \bmod 16 = 8$
…

+ fast
+ abundant, well proven for many applications
- limited period length
- fail some of the statistical tests (n-dim random points align into hyperplanes)

Pseudo-Random Number Generators

Lagged Fibonacci:

$$x_{n+1} = (x_{n-k} + x_{n-l}) \bmod m$$

State is dependent of two previous elements.
Period length can be longer than *m.*
Mitchell and Moore (1958): k=24, l=55, m even, arbitrary first 54 number.
Period length is at least 2^{55}.

Blum Blum Shub, 1986:

$$x_{n+1} = x_n^2 \bmod m$$

Where m = pq, p and q are primes.
p mod 4 = 3, q mod 4 = 3.

Multiplicative Recursive:

$$x_{n+1} = (a_1 x_n + a_2 x_{n-1} + a_3 x_{n-2}) \bmod m$$

Mersenne Twister:

State consists of 624 numbers.
A “twist” operator generates the new state.
Very long period (2^{219937}).
Passes most tests.
Not particularly fast, memory usage can be significant.

Tausworthe:

A multiplicative generator that produces bits:

$$x_{n+1} = (a_1 x_n + a_2 x_{n-1} + \cdots + a_3 x_{n-2}) \bmod 2$$

STREAM
High Performance Computing

Pseudo-Random Number Generators

add-with-carry, subtract-with-borrow, multiply-with-carry, Marsaglia:

AWC: $x_{n+1} = (x_{n-k} + x_{n-l} + carry) \bmod m$

SWB: $x_{n+1} = (x_{n-k} - x_{n-l} - carry) \bmod m$

MWC: $x_{n+1} = (ax_{n-1} + carry) \bmod m$

+ large period length (10^{18}, easily)
- two seed numbers

XORshift, XORWOW, Marsaglia:

```
x ^= (x << 21);
x ^= (x >> 35);
x ^= (x << 4);
```

```
t=(x^(x>>2)); x=y; y=z; z=w; w=v;
v=(v^(v<<4))^(t^(t<<1));
return (d+=362437)+v;
```

+ large period length (XORshift: 2^{64}, XORWOW: 2^{192})

Cryptographically safe RNGs:

Should pass the next-bit test. Given the first *k* bits of a random sequence, there is no polynomial-time algorithm that can predict the (*k*+1)th bit with probability of success non-negligibly better than 50%

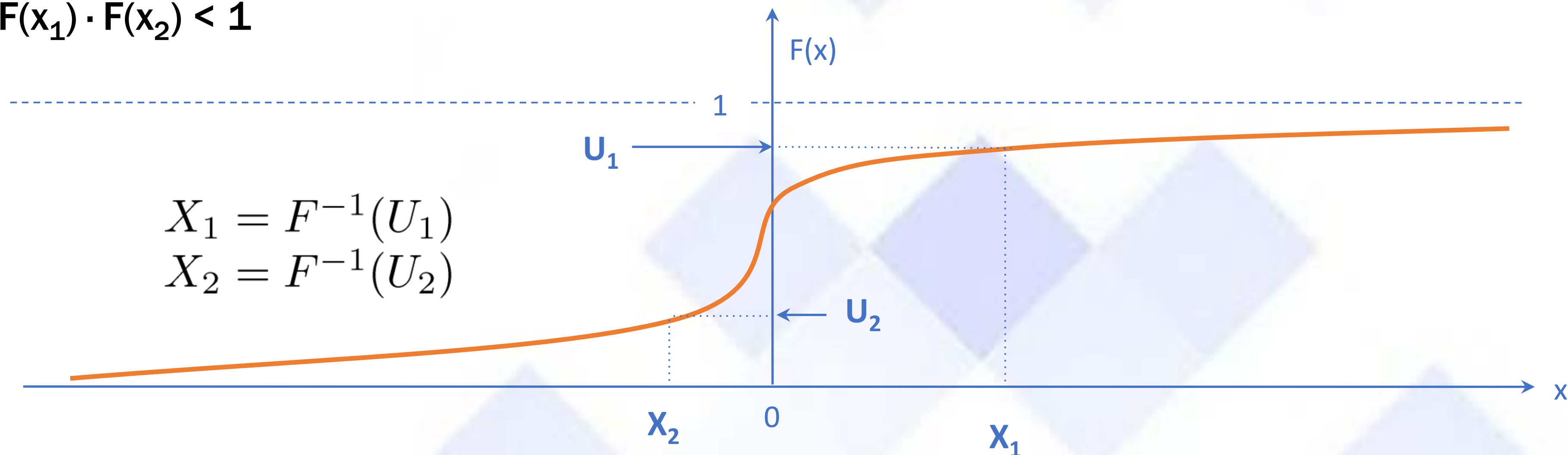
If part of its state has been revealed, it should be impossible to reconstruct the stream of random numbers prior to the revelation.
E.g.: Block Ciphers, Philox.

Non-Uniform Distributions

Random number: U(0,1) → Random variate: arbitrary distribution

Inverse Transformation

- generate continuous random variate X
  - distribution function F (continuous, strictly increasing)
    - $0 < F(x) < 1$
    - if $x_1 < x_2$ then $0 < F(x_1) \cdot F(x_2) < 1$
  - inverse of F: F^{-1}
- algorithm
  - generate U ~ U(0,1)
  - return $X = F^{-1}(U)$
  - returned value X has desired distribution F



Other method: Rejection

Pseudo-Random Number Generators

Parallel RNGs

Avoid correlation: results of individual threads have to be statistically independent.
How can we ensure parallel threads won't overlap?

Choosing different seeds:

There is a non-zero probability for correlation.
It can be feasible if either:

- the period length is very high compared to number of randoms used.
- randoms are used for many different purposes.

Parametrization:

Different parameters for each thread. Only feasible with a low number of threads.

Skip ahead (leapfrogging):

Some RNGs make possible to calculate x_n without calculating each previous element.
For LCG:

$$x_{n+k} = \left(a^k x_n + \frac{a^{k-1}}{a-1} c\right) \bmod m$$

Counter-based generators:

Creating a pseudorandom number generator using only an integer counter as the internal state of the generator.

Statistical Tests

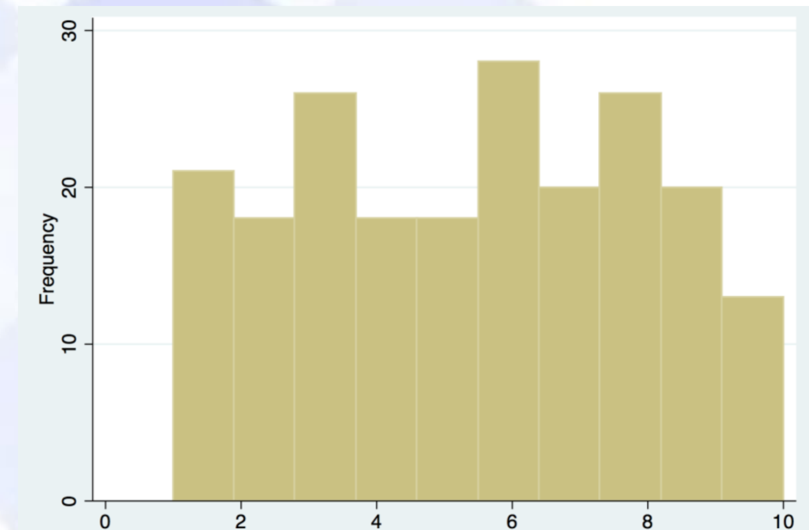
+ visual n-D pattern check, collisions, coupon, equidistribution, gap, maximum of t, permutations, poker, runs up, serial, sum of distributions
+ DIEHARD package (Marsaglia): Birthday spacings, Overlapping Permutations, Ranks of matrices, Monkey tests, Count the 1s, Parking lot test, Minimum distance test, Random spheres test, The squeeze test, Overlapping sums test, Runs test, The craps test
+ TestU01 (L'Ecuyer)
+ Binary Matrix Rank Test, Discrete Fourier Transform (Spectral Test), Non-Overlapping Template Matching Test
+ Overlapping Template Matching Test, Maurer's Universal Statistical Test, Linear Complexity Test, Serial Test
+ Approximate Entropy Test, Cumulative Sums Test, Random Excursions Test, Random Excursions Variant Test
+ Parallel RNG tests (Exponential sums, parallel spectral test, interleaved, Fourier transform, blocking)

Frequency Test

The most fundamental test. In a true random sequence, the number of all possible values should be about the same.

This test checks whether this is correct or not.

The result of a statistical test is itself statistical in its nature: χ^2-test can be used to judge the likeliness of the outcome.



Libraries & Implementations (CPU)

+ std
+ Boost
+ Intel MKL
+ NumPy

Numerical Recipes

$m = 2^{32}$ a = 1664525 c = 1013904223

GCC

$m = 2^{32}$ a = 1103515245 c = 12345

MMIX

$m = 2^{64}$ a = 6364136223846793005 c = 1442695040888963407

Libraries & Implementations (GPU)

CUDA Implementations

CURAND:XORWOW, MRG32k3a, MTGP32
Thrust: RANDO, RANLUX, RANLUX48

TAUS88
NAG: MRG32k3a, MT19937
GASPRNG: LFG, MLFG, PCMLCG, LCG48, LCG64
Random123: AES, Threefish, Philox
PRAND: MRG32k3a, MT19937, LFSR1123
MPRNG: MTGP, RANECU, TT800, PM, TAUS88,
LFSR113, KISS07, DRAND48
GPU-rand: LCG, LFG, WHG
ShoveRand: MRG32k3a, MTGP32, TinyMT
MTGP: Different MT variants

Variations of Mersenne Twister, Combined Multiple Recursive and Tausworthe generators are the most common.

OpenCL Implementations:

Random123: ->CUDA
MTGP: ->CUDA
OpenCLRNG: MT variant
RANLUXCL: RANLUX
MWC64X: MWC64x

cuRAND and rocRAND

How to use these random generators:

- Create the the generator itself and/or the the generator constans.
- Call the host side random generations(If it’s has only host side interface or you want generate and store your number generators in memory during your kernel)
- Call the device function to generate random number on the fly where you need.

- Simple Example:

```
__host__ curandStatus_t
curandMakeMTGP32KernelState(curandStateMtgp32_t *s,
                                                     mtgp32_params_fast_t
params[],
                            mtgp32_kernel_params_t *k,
                            int n,
                            unsigned long long seed)
```

```
__device__ unsigned int
curand (curandStateMtgp32_t *state)
```

STREAM
High Performance Computing

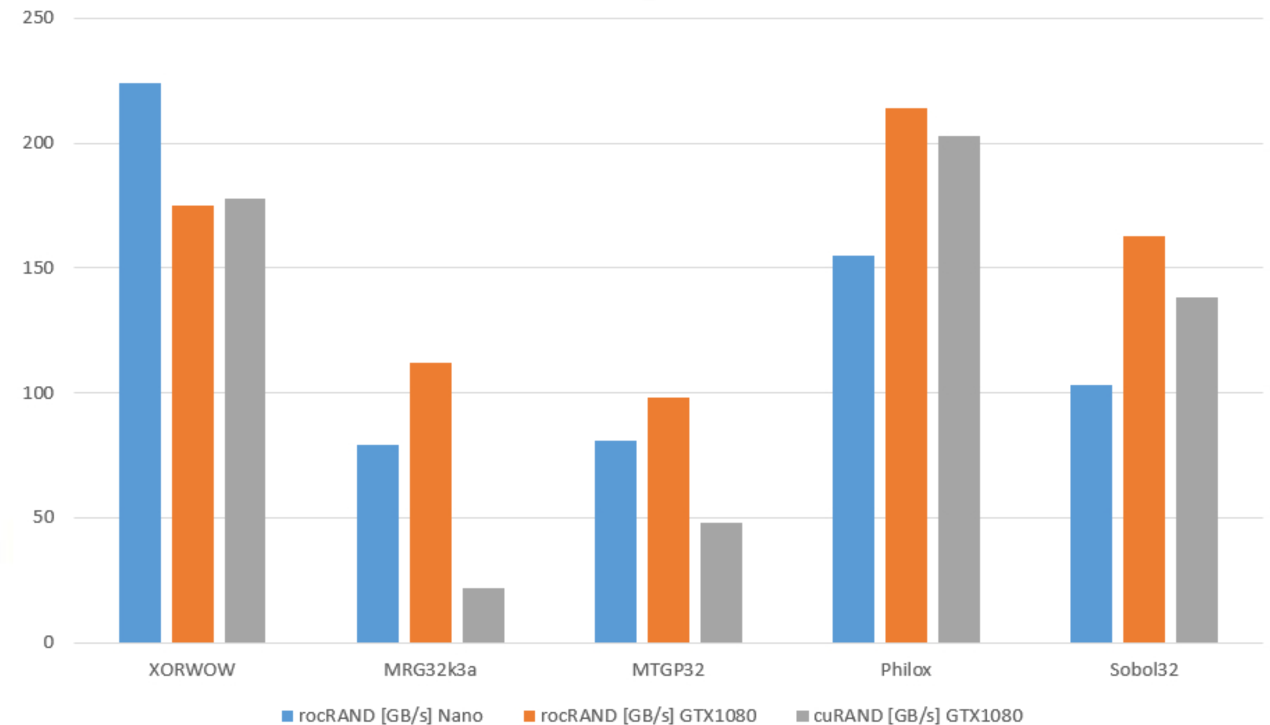
RocRAND

The library provides the most used PRNGs and QRNG (Quasi RNG) based on what we found on Github.

Several you can find in cuRAND:

- XORWOW
- MRG32k3a
- Mersenne Twister for Graphic Processors (MTGP32)
- Philox (4x32, 10 rounds)
- Sobol32

Before optimization:



STREAM
High Performance Computing

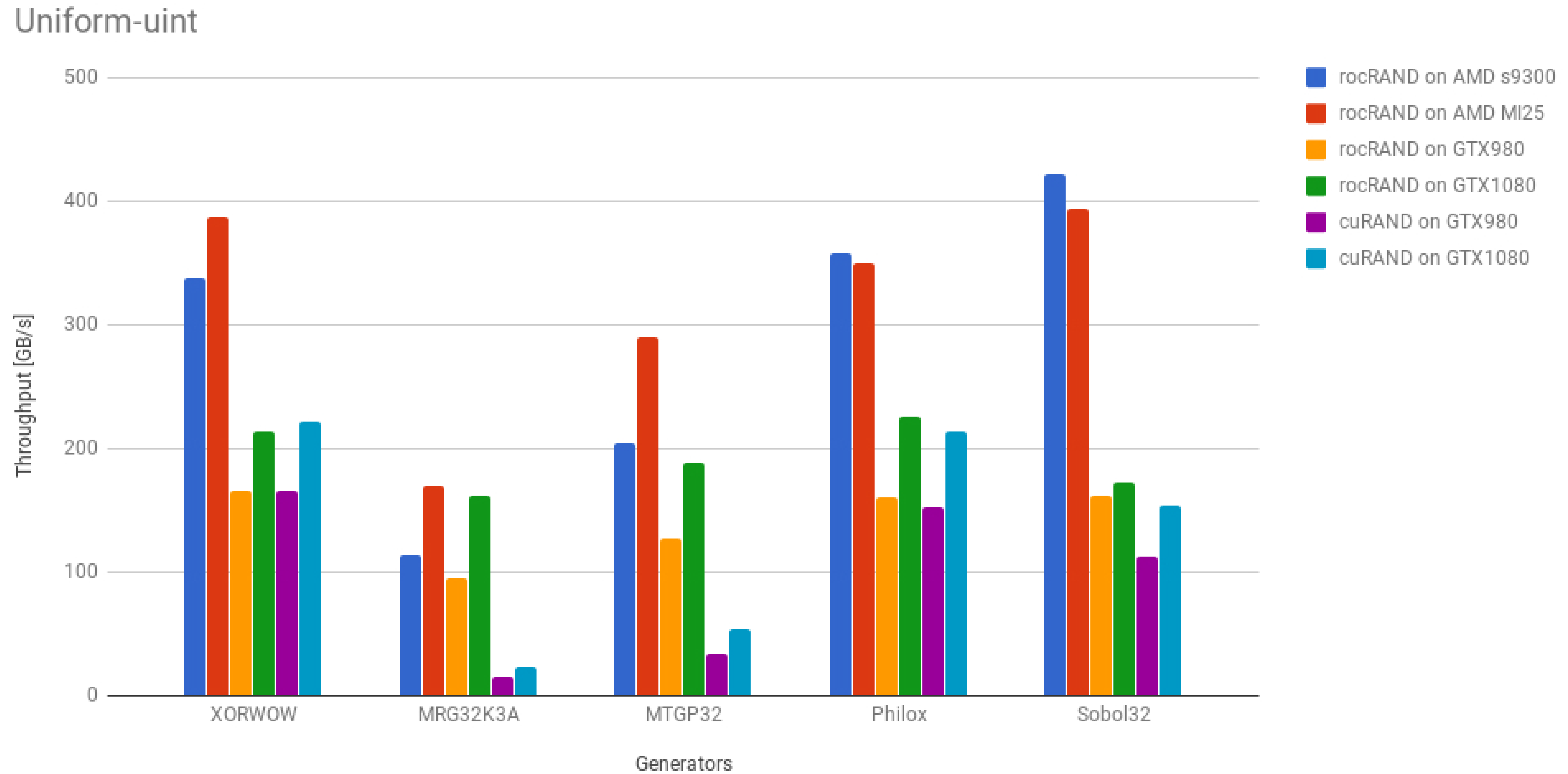
Used low level instruction (asm instructions)


```
unsigned long long r;
unsigned long long c; // carry bits, SGPR, unused
// x has "r" constraint. This allows to use both VGPR and SGPR
// (to save VGPR) as input.
// y and z have "v" constraints, because only one SGPR or literal
// can be read by the instruction.
asm volatile("v_mad_u64_u32 %0, %1, %2, %3, %4"
    : "=v"(r), "=s"(c) : "r"(x), "v"(y), "v"(z)
);
return r;

#elif defined(__HIP_PLATFORM_NVCC__) && defined(__HIP_DEVICE_COMPILE__) \
    && defined(ROCRAND_ENABLE_INLINE_ASM)

unsigned long long r;
asm("mad.wide.u32 %0, %1, %2, %3;"
   : "=l"(r) : "r"(x), "r"(y), "l"(z)
);
return r;
```

STREAM
High Performance Computing

RocRAND

After optimization:



STREAM
High Performance Computing

More Information

Numerous articles by G. Marsaglia, P. L'Ecuyer, M. Matsumoto and many others

Donald Knuth: The Art of Computer Programming – Vol 2.

Volodymyr Kindratenko: Numerical Computations with GPU

James E. Gentle: Random Number Generation and Monte Carlo Methods

https://github.com/ROCmSoftwarePlatform/rocRAND

ISO 28640:2010 Random variate generation methods

NIST SP 800-90A Recommendation for Random Number Generation Using Deterministic Random Bit Generators

Thanks!