CUDA Toolkit 4.2
CURAND Guide

Published by
    NVIDIA Corporation
    2701 San Tomas Expressway
    Santa Clara, CA 95050

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES,
DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND
SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS". NVIDIA MAKES
NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH
RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED
WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS
FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA
Corporation assumes no responsibility for the consequences of use of such information or
for any infringement of patents or other rights of third parties that may result from its use.
No license is granted by implication or otherwise under any patent or patent rights of
NVIDIA Corporation. Specifications mentioned in this publication are subject to change
without notice. This publication supersedes and replaces all information previously
supplied. NVIDIA Corporation products are not authorized for use as critical components
in life support devices or systems without express written approval of NVIDIA
Corporation.

Trademarks

NVIDIA, CUDA, and the NVIDIA logo are trademarks or registered trademarks of
NVIDIA Corporation in the United States and other countries. Other company and
product names may be trademarks of the respective companies with which they are
associated.

Copyright

Copyright ©2005-2012 by NVIDIA Corporation. All rights reserved.

Portions of the MTGP32 (Mersenne Twister for GPU) library routines are subject to the following copyright:

CURAND Library

The CURAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers. A *pseudorandom* sequence of numbers satisfies most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm. A *quasirandom* sequence of n-dimensional points is generated by a deterministic algorithm designed to fill an n-dimensional space evenly.

CURAND consists of two pieces: a library on the host (CPU) side and a device (GPU) header file. The host-side library is treated like any other CPU library: users include the header file, `/include/curand.h`, to get function declarations and then link against the library. Random numbers can be generated on the device or on the host CPU. For device generation, calls to the library happen on the host, but the actual work of random number generation occurs on the device. The resulting random numbers are stored in global memory on the device. Users can then call their own kernels to use the random numbers, or they can copy the random numbers back to the host for further processing. For host CPU generation, all of the work is done on the host, and the random numbers are stored in host memory.

The second piece of CURAND is the device header file, `/include/curand_kernel.h`. This file defines device functions for setting up random number generator states and generating sequences of random numbers. User code may include this header file, and user-written kernels may then call the device functions defined in the header file. This allows random numbers to be generated and immediately consumed by user kernels without requiring the random numbers to be written to and then read from global memory.

Compatibility and Versioning

The host API of CURAND is intended to be backward compatible at the source level with future releases (unless stated otherwise in the release notes of a specific future release). In other words, if a program uses CURAND, it should continue to compile and work correctly with newer versions of CURAND without source code changes.

CURAND is not guaranteed to be backward compatible at the binary level. Using a different version of the `curand.h` header file and the shared library is not supported. Using

different versions of CURAND and the CUDA runtime is not supported.

The device API should be backward compatible at the source level for public functions in most cases.

Host API Overview

To use the host API, user code should include the library header file `curand.h` and dynamically link against the CURAND library. The library uses the CUDA runtime, so user code must also use the runtime. The CUDA driver API is not supported by CURAND.

Random numbers are produced by generators. A generator in CURAND encapsulates all the internal state necessary to produce a sequence of pseudorandom or quasirandom numbers. The normal sequence of operations is as follows:

1. Create a new generator of the desired type (see Generator Types) with `curandCreateGenerator()`.

2. Set the generator options (see Generator Options); for example, use `curandSetPseudoRandomGeneratorSeed()` to set the seed.

3. Allocate memory on the device with `cudaMalloc()`.

4. Generate random numbers with `curandGenerate()` or another generation function.

5. Use the results.

6. If desired, generate more random numbers with more calls to `curandGenerate()`.

7. Clean up with `curandDestroyGenerator()`.

To generate random numbers on the host CPU, in step one above call `curandCreateGeneratorHost()`, and in step three, allocate a host memory buffer to receive the results. All other calls work identically whether you are generating random numbers on the device or on the host CPU.

It is legal to create several generators at the same time. Each generator encapsulates a separate state and is independent of all other generators. The sequence of numbers produced by each generator is deterministic. Given the same set-up parameters, the same sequence will be generated with every run of the program. Generating random numbers on the device will result in the same sequence as generating them on the host CPU.

Note that `curandGenerate()` in step 4 above launches a kernel and returns asynchronously. If you launch another kernel in a different stream, and that kernel needs to use the results of curandGenerate(), you must either call `cudaThreadSynchronize()` or use the stream management/event management routines, to ensure that the random generation kernel has finished execution before the new kernel is launched.

Note that it is not valid to pass a host memory pointer to a generator that is running on

the device, and it is not valid to pass a device memory pointer to a generator that is running on the CPU. Behavior in these cases is undefined.

Generator Types

Random number generators are created by passing a type to `curandCreateGenerator()`. There are five types of random number generators in CURAND, that fall into two categories. `CURAND_RNG_PSEUDO_XORWOW`, `CURAND_RNG_PSEUDO_MRG32K3A`, and `CURAND_RNG_PSEUDO_MTGP32` are pseudorandom number generators. `CURAND_RNG_PSEUDO_XORWOW` is implemented using the XORWOW algorithm, a member of the xor-shift family of pseudorandom number generators. `CURAND_RNG_PSEUDO_MRG32K3A` is a member of the Combined Multiple Recursive family of pseudorandom number generators. `CURAND_RNG_PSEUDO_MTGP32` is a member of the Mersenne Twister family of pseudorandom number generators, with parameters customized for operation on the GPU. There are 4 variants of the basic SOBOL' quasi random number generator. All of the variants generate sequences in up to 20,000 dimensions. `CURAND_RNG_QUASI_SOBOL32`, `CURAND_RNG_QUASI_SCRAMBLED_SOBOL32`, `CURAND_RNG_QUASI_SOBOL64`, and `CURAND_RNG_QUASI_SCRAMBLED_SOBOL64` are quasirandom number generator types. `CURAND_RNG_QUASI_SOBOL32` is a Sobol' generator of 32-bit sequences. `CURAND_RNG_QUASI_SCRAMBLED_SOBOL32` is a scrambled Sobol' generator of 32-bit sequences. `CURAND_RNG_QUASI_SOBOL64` is a Sobol' generator of 64-bit sequences. `CURAND_RNG_QUASI_SCRAMBLED_SOBOL64` is a scrambled Sobol' generator of 64-bit sequences.

Generator Options

Once created, random number generators can be defined using the general options seed, offset, and order.

Seed

The seed parameter is a 64-bit integer that initializes the starting state of a pseudorandom number generator. The same seed always produces the same sequence of results.

Offset

The offset parameter is used to skip ahead in the sequence. If offset = 100, the first random number generated will be the 100th in the sequence. This allows multiple runs of the same program to continue generating results from the same sequence without overlap. Note that the skip ahead function is not available for the `CURAND_RNG_PSEUDO_MTGP32` generator.

Order

The order parameter is used to choose how the results are ordered in global memory. There are three ordering choices for pseudorandom sequences: `CURAND_ORDERING_PSEUDO_DEFAULT`, `CURAND_ORDERING_PSEUDO_BEST`, and `CURAND_ORDERING_PSEUDO_SEEDED`. There is one ordering choice for quasirandom numbers, `CURAND_ORDERING_QUASI_DEFAULT`. The default ordering for pseudorandom number generators is `CURAND_ORDERING_PSEUDO_DEFAULT`, while the default ordering for quasirandom number generators is `CURAND_ORDERING_QUASI_DEFAULT`.

Currently, the two pseudorandom orderings `CURAND_ORDERING_PSEUDO_DEFAULT` and `CURAND_ORDERING_PSEUDO_BEST` produce the same output ordering for all pseudo-random generators. However, future releases of CURAND may change the ordering associated with `CURAND_ORDERING_PSEUDO_BEST` to improve either performance or the quality of the results. It will always be the case that the ordering obtained with `CURAND_ORDERING_PSEUDO_BEST` is deterministic and is the same for each run of the program. The ordering returned by `CURAND_ORDERING_PSEUDO_DEFAULT` is guaranteed to remain the same for all CURAND releases. In the current release, only the XORWOW generator has more than one ordering.

The behavior of the ordering parameters for each generator type is outlined below:

▶ XORWOW pseudorandom generator

- `CURAND_ORDERING_PSEUDO_BEST`

  The output ordering of `CURAND_ORDERING_PSEUDO_BEST` is the same as `CURAND_ORDERING_PSEUDO_DEFAULT` in the current release.

- `CURAND_ORDERING_PSEUDO_DEFAULT`

  The result at offset n in global memory is from position

  $$(n \bmod 4096) \cdot 2^{67} + \lfloor n/4096 \rfloor$$

  in the original XORWOW sequence.

- `CURAND_ORDERING_PSEUDO_SEEDED`

  The result at offset n in global memory is from position $\lfloor n/4096 \rfloor$ in the XORWOW sequence seeded with a combination of the user seed and the number $n \bmod 4096$. In other words, each of 4096 threads uses a different seed. This seeding method reduces state setup time but may result in statistical weaknesses of the pseudorandom output for some user seed values.

▶ MRG32k3a pseudorandom generator

- `CURAND_ORDERING_PSEUDO_BEST`

  The output ordering of `CURAND_ORDERING_PSEUDO_BEST` is the same as `CURAND_ORDERING_PSEUDO_DEFAULT` in the current release.

- **CURAND_ORDERING_PSEUDO_DEFAULT**

  The result at offset n in global memory is from position

  $$(n \bmod 4096) \cdot 2^{76} + \lfloor n/4096 \rfloor$$

  in the original MRG32k3a sequence. (Note that the stride between subsequent samples for MRG32k3a is not the same as for XORWOW)

▶ MTGP32 pseudorandom generator

- **CURAND_ORDERING_PSEUDO_BEST**

  The output ordering of **CURAND_ORDERING_PSEUDO_BEST** is the same as **CURAND_ORDERING_PSEUDO_DEFAULT** in the current release.

- **CURAND_ORDERING_PSEUDO_DEFAULT**

  The MTGP32 generator actually generates 64 distinct sequences based on different parameter sets for the basic algorithm. Let $S(p)$ be the sequence for parmeter set p.

  The result at offset n in global memory is from position $(n \bmod 256)$ from the sequence
  $$S(\lfloor n/256 \rfloor \bmod 64)$$

  In other words 256 samples from $S(0)$ are followed by 256 samples from $S(1)$ and so-on, up to $S(63)$. This pattern repeats, so the subsequent 256 samples are from $S(0)$, followed by 256 samples from $S(1)$, ands so on.

▶ 32 and 64 bit SOBOL and Scrambled SOBOL quasirandom generators

- **CURAND_ORDERING_QUASI_DEFAULT**

  When generating n results in d dimensions, the output will consist of n/d results from dimension 1, followed by n/d results from dimension 2, and so on up to dimension d. Only exact multiples of the dimension size may be generated. The dimension parameter d is set with **curandSetQuasiRandomGeneratorDimensions()** and defaults to 1.

Return Values

All CURAND host library calls have a return value of **curandStatus_t**. Calls that succeed without errors return **CURAND_STATUS_SUCCESS**. If errors occur, other values are returned depending on the error. Because CUDA allows kernels to execute asynchronously from CPU code, it is possible that errors in a non-CURAND kernel will be detected during a call to a library function. In this case, **CURAND_STATUS_PREEXISTING_ERROR** is returned.

Generation Functions

```
curandStatus_t
curandGenerate(
    curandGenerator_t generator,
    unsigned int *outputPtr, size_t num)
```

The `curandGenerate()` function is used to generate pseudo- or quasirandom bits of output. For XORWOW, MRG32k3a, MTGP32, and SOBOL32 generators, each output element is a 32-bit unsigned int where all bits are random. For SOBOL64 generators, each output element is a 64-bit unsigned long long where all bits are random.

```
curandStatus_t
curandGenerateUniform(
    curandGenerator_t generator,
    float *outputPtr, size_t num)
```

The `curandGenerateUniform()` function is used to generate uniformly distributed floating point values between 0.0 and 1.0, where 0.0 is excluded and 1.0 is included.

```
curandStatus_t
curandGenerateNormal(
    curandGenerator_t generator,
    float *outputPtr, size_t n,
    float mean, float stddev)
```

The `curandGenerateNormal()` function is used to generate normally distributed floating point values with the given mean and standard deviation.

```
curandStatus_t
curandGenerateLogNormal(
    curandGenerator_t generator,
    float *outputPtr, size_t n,
    float mean, float stddev)
```

The `curandGenerateLogNormal()` function is used to generate log-normally distributed floating point values based on a normal distribution with the given mean and standard deviation.

```
curandStatus_t
curandGenerateUniformDouble(
    curandGenerator_t generator,
    double *outputPtr, size_t num)
```

The function `curandGenerateUniformDouble()` generates uniformly distributed random numbers in double precision. The function

```
curandStatus_t
curandGenerateNormalDouble(
    curandGenerator_t generator,
    double *outputPtr, size_t n,
    double mean, double stddev)
```

`curandGenerateNormalDouble()` generates normally distributed results in double precision with the given mean and standard deviation. Double precision results can only be generated on devices of compute capability 1.3 or above, and the host.

```
curandStatus_t
curandGenerateLogNormalDouble(
    curandGenerator_t generator,
    double *outputPtr, size_t n,
    double mean, double stddev)
```

`curandGenerateLogNormalDouble()` generates log-normally distributed results in double precision, based on a normal distribution with the given mean and standard deviation.

For quasirandom generation, the number of results returned must be a multiple of the dimension of the generator.

Generation functions can be called multiple times on the same generator to generate successive blocks of results. For pseudorandom generators, multiple calls to generation functions will yield the same result as a single call with a large size. For quasirandom generators, because of the ordering of dimensions in memory, many shorter calls will not produce the same results in memory as one larger call; however the generated n-dimensional vectors will be the same.

Double precision results can only be generated on devices of compute capability 1.3 or above, and the host.

Host API Example

```
/*
 * This program uses the host CURAND API to generate 100
 * pseudorandom floats.
 */
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand.h>

#define CUDA_CALL(x) do { if((x)!=cudaSuccess) { \
```

```
    printf("Error at %s:%d\n",__FILE__,__LINE__);\
    return EXIT_FAILURE;}} while(0)
#define CURAND_CALL(x) do { if((x)!=CURAND_STATUS_SUCCESS) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__);\
    return EXIT_FAILURE;}} while(0)

int main(int argc, char *argv[])
{
    size_t n = 100;
    size_t i;
    curandGenerator_t gen;
    float *devData, *hostData;

    /* Allocate n floats on host */
    hostData = (float *)calloc(n, sizeof(float));

    /* Allocate n floats on device */
    CUDA_CALL(cudaMalloc((void **)&devData, n*sizeof(float)));

    /* Create pseudo-random number generator */
    CURAND_CALL(curandCreateGenerator(&gen,
                CURAND_RNG_PSEUDO_DEFAULT));

    /* Set seed */
    CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen,
                1234ULL));

    /* Generate n floats on device */
    CURAND_CALL(curandGenerateUniform(gen, devData, n));

    /* Copy device memory to host */
    CUDA_CALL(cudaMemcpy(hostData, devData, n * sizeof(float),
        cudaMemcpyDeviceToHost));

    /* Show result */
    for(i = 0; i < n; i++) {
        printf("%1.4f ", hostData[i]);
    }
    printf("\n");

    /* Cleanup */
    CURAND_CALL(curandDestroyGenerator(gen));
    CUDA_CALL(cudaFree(devData));
    free(hostData);
```

```
    return EXIT_SUCCESS;
}
```

Performance Notes

In general you will get the best performance from the CURAND library by generating blocks of random numbers that are as large as possible. Fewer calls to generate many random numbers is more efficient than many calls generating only a few random numbers. The default pseudorandom generator, XORWOW, with the default ordering takes some time to setup the first time it is called. Subsequent generation calls do not require this setup. To avoid this setup time, use the `CURAND_ORDERING_PSEUDO_SEEDED` ordering.

The MTGP32 Mersenne Twister algorithm is closely tied to the thread and block count. The state structure for MTGP32 actually contains the state for 256 consecutive samples from a given sequence, as determined by a specific parameter set. Each of 64 blocks uses a different parameter set and each of 256 threads generates one sample from the state, and updates the state. Hence the most efficient use of MTGP32 is to generate a multiple of 16384 samples.

Device API Overview

To use the device API, include the file `curand_kernel.h` in files that define kernels that use CURAND device functions. The device API includes functions for pseudorandom generation and quasirandom generation.

Pseudorandom Sequences

The functions for pseudorandom sequences support bit generation and generation from distributions.

Bit Generation with XORWOW and MRG32k3a generators

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

Following a call to `curand_init()`, `curand()` returns a sequence of pseudorandom numbers with a period greater than 2^{190}. If `curand()` is called with the same initial state each time, and the state is not modified between the calls to `curand()`, the same sequence is always generated.

```
__device__ void
curand_init (
    unsigned long long seed, unsigned long long sequence,
    unsigned long long offset, curandState_t *state)
```

The `curand_init()` function sets up an initial state allocated by the caller using the given seed, sequence number, and offset within the sequence. Different seeds are guaranteed to produce different starting states and different sequences. The same seed always produces the same state and the same sequence. The state set up will be the state after $2^{67} \cdot$ `sequence` $+$ `offset` calls to `curand()` from the seed state.

Sequences generated with different seeds usually do not have statistically correlated values, but some choices of seeds may give statistically correlated sequences. Sequences generated with the same seed and different sequence numbers will not have statistically correlated values.

For the highest quality parallel pseudorandom number generation, each experiment should be assigned a unique seed. Within an experiment, each thread of computation should be assigned a unique sequence number. If an experiment spans multiple kernel launches, it is recommended that threads between kernel launches be given the same seed, and sequence numbers be assigned in a monotonically increasing way. If the same configuration of threads is launched, random state can be preserved in global memory between launches to avoid state setup time.

Bit Generation with the MTGP32 generator

The MTGP32 generator is an adaptation of code developed at Hiroshima University (see [1]). In this algorithm, samples are generated for multiple sequences, each sequence based on a set of computed parameters. CURAND uses the 200 parameter sets that have been pre-generated for the 32-bit generator with period 2^{11213}. It would be possible to generate other parameter sets, as described in [1], and use those instead. There is one state structure for each parameter set (sequence), and the algorithm allows thread-safe generation and state update for up to 256 concurrent threads (within a single block) for each of the 200 sequences.

Note that two different blocks can not operate on the same state safely. Also note that, within a block, at most 256 threads may operate on a given state.

For the MTGP32 generator, two host functions are provided to help set up parameters for the different sequences in device memory, and to set up the initial state.

```
__host__ curandStatus_t
curandMakeMTGP32Constants(mtgp32_params_fast_t params[], mtgp32_kernel_params_t * p)
```

This function re-organizes the paramter set data from the pre-generated format (`mtgp32_params_fast_t`) into the format used by the kernel functions (`mtgp32_kernel_params_t`), and copies them to device memory.

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

This function initializes `n` states, based on the specified parameter set and seed, and copies them to device memory indicated by `s`. Note that if you are using the pre-generated states, the maximum value of `n` is 200.

The CURAND MTGP32 generator provides two kernel functions to generate random bits.

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

This function computes a thread index, and for that index generates a result and updates state. The thread index `t` is computed as:

```
t = (blockDim.z * blockDim.y * threadIdx.z) + (blockDim.x * threadIdx.y) +
threadIdx.x
```

This function may be called repeatedly from a single kernel launch, with the following constraints:

It may only be called safely from a block that has 256 or fewer threads.

A given state may not be used by more than one block.

A given block may generate randoms using multiple states.

```
__device__ unsigned int
curand_mtgp32_specific(curandStateMtgp32_t *state, unsigned char index, unsigned char n)
```

This function generates a result and updates state for the position specified by a thread-specific `index`, and advances the offset in the state by `n` positions. `curand_mtgp32_specific` may be called multiple times within a kernel launch, with the following constraints:

At most 256 threads may call this function for a given state.

Within a block, for a given state, if `n` threads are calling the function, the indices must run from `0...n-1`. The indices do not have to match the thread numbers, and may be distributed among the threads as required by the calling program.

A given state may not be used by more than one block.

A given block may generate randoms using multiple states.

Figure 1 is an illustration of how blocks and threads in MTGP32 operate on the generator states. Each row represents a circular state array of 32-bit integers `s(n)`. Threads operating on the array are identified as `T(m)`. The specific case shown matches the internal implementation of the host API, which launches 64 blocks of 256 threads. Each block operates on a different sequence, determined by a unique set of paramters, `P(n)`. One complete state of an MTGP32 sequence is defined by 351 32-bit integers. Each thread `T(m)` operates on one of these integers, `s(n+m)` combining it with `s(n+m+1)` and a pickup element `s(n+m+p)`, where `p <= 95`. It stores the new state at position `s(n+m+351)` in the state array. After thread synchronization, the base index `n` is advanced by the number of threads that have updated the state. To avoid being overwritten, the array itself must be at least $256 + 351$ integers in length. In fact it is sized at 1024 integers for efficiency of indexing.

The limitation on the number of threads in a block, which can operate on a given state array, arises from the need to ensure that state `s(n+351)` has been updated before it is needed as a pickup state. If there were a thread `T(256)`, it could use `s(n+256+95)` i.e. `s(n+351)` before thread zero has updated `s(n+351)`. If an application requires that more than 256 threads in a block invoke an MTGP32 generator function, it must use multiple MTGP32 states, either by using multiple parameter sets, or by using multiple generators with different seeds. Also note that the generator functions synchronize threads at the end of each call, so it is most efficient for 256 threads in a block to invoke the generator.
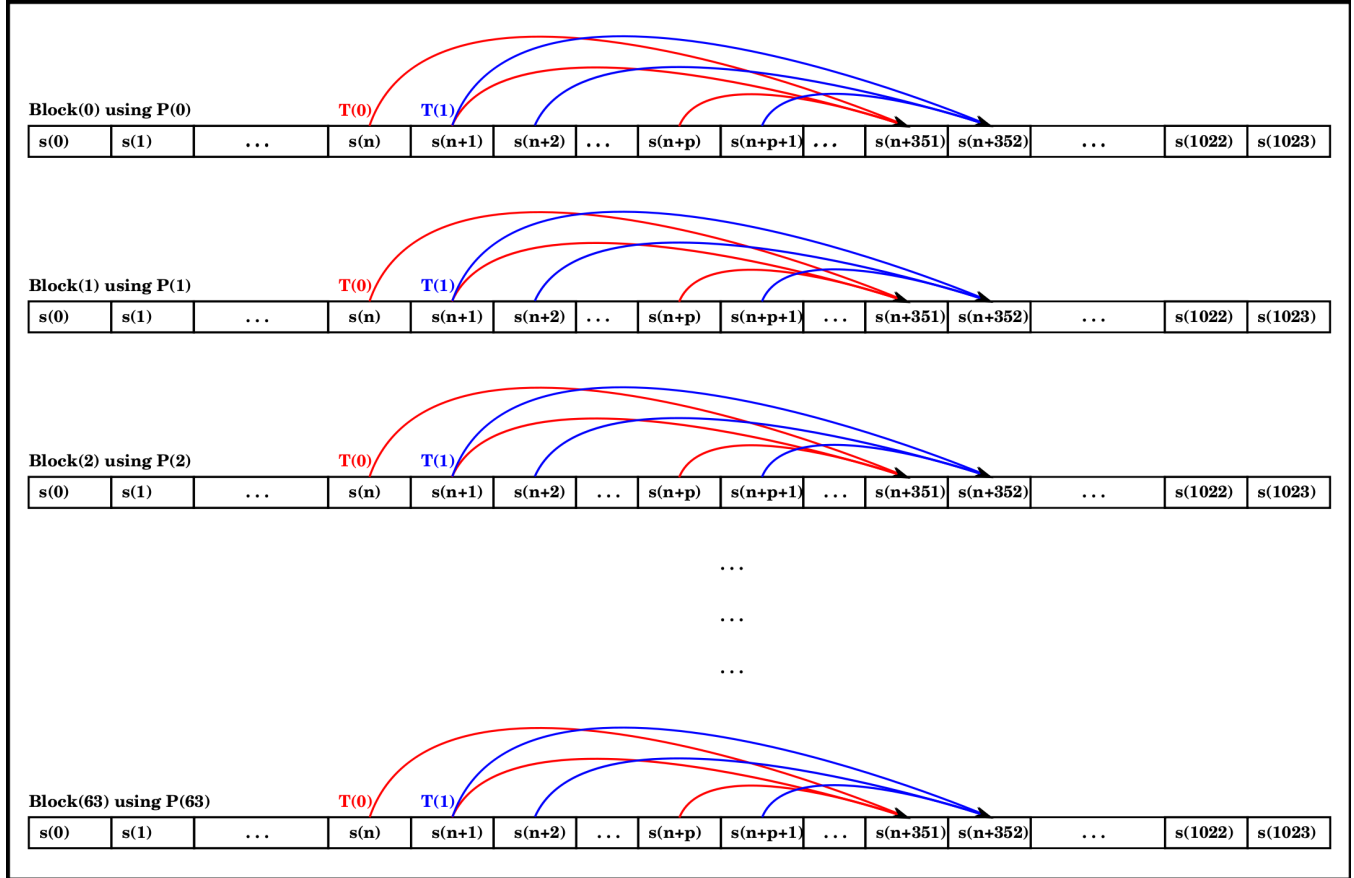
Figure 1: MTGP32 Block and Thread Operation

Distributions

```
__device__ float
curand_uniform (curandState_t *state)
```

This function returns a sequence of pseudorandom floats uniformly distributed between 0.0 and 1.0. It may return from 0.0 to 1.0, where 1.0 is included and 0.0 is excluded. Distribution functions may use any number of unsigned integer values from a basic generator. The number of values consumed is not guaranteed to be fixed.

```
__device__ float
curand_normal (curandState_t *state)
```

This function returns a single normally distributed float with mean 0.0 and standard deviation 1.0. This result can be scaled and shifted to produce normally distributed values with any mean and standard deviation.

```
__device__ float
curand_log_normal (curandState_t *state, float mean, float stddev)
```

This function returns a single log-normally distributed float based on a normal distribution with the given mean and standard deviation.

```
__device__ double
curand_uniform_double (curandState_t *state)
```

```
__device__ double
curand_normal_double (curandState_t *state)
```

```
__device__ double
curand_log_normal_double (curandState_t *state, double mean, double stddev)
```

The three functions above are the double precision versions of `curand_uniform()`, `curand_normal()`, and `curand_log_normal()`.

For pseudorandom generators, the double precision functions use multiple calls to `curand()` to generate 53 random bits.

```
__device__ float2
curand_normal2 (curandState_t *state)
```

```
__device__ float2
curand_log_normal2 (curandState_t *state)
```

```
__device__ double2
curand_normal2_double (curandState_t *state)
```

```
__device__ double2
curand_log_normal2_double (curandState_t *state)
```

The above functions generate two normally or log normally distributed pseudorandom results with each call. Because the underlying implementation uses the Box-Muller transform, this is generally more efficient than generating a single result with each call.

Quasirandom Sequences

Although the default generator type is pseudorandom numbers from XORWOW, Sobol' sequences based on Sobol' 32-bit integers can be generated using the following functions:

```
__device__ void
curand_init (
    unsigned int *direction_vectors,
    unsigned int offset,
    curandStateSobol32_t *state)
```

```
__device__ void
curand_init (
    unsigned int *direction_vectors,
    unsigned int scramble_c,
    unsigned int offset,
    curandStateScrambledSobol32_t *state)

__device__ unsigned int
curand (curandStateSobol32_t *state)

__device__ float
curand_uniform (curandStateSobol32_t *state)

__device__ float
curand_normal (curandStateSobol32_t *state)

__device__ float
curand_log_normal (
    curandStateSobol32_t *state,
    float mean,
    float stddev)

__device__ double
curand_uniform_double (curandStateSobol32_t *state)

__device__ double
curand_normal_double (curandStateSobol32_t *state)

__device__ double
curand_log_normal_double (
    curandStateSobol32_t *state,
    double mean,
    double stddev)
```

The `curand_init()` function initializes the quasirandom number generator state. There is no seed parameter, only direction vectors and offset. For scrambled Sobol' generators, there is an additional parameter `scramble_c`, which is the initial value of the scrambled sequence. For the `curandStateSobol32_t` type and the `curandStateScrambledSobol32_t` type the direction vectors are an array of 32 unsigned integer values. For the `curandStateSobol64_t` type and the `curandStateScrambledSobol64_t` type the direction vectors are an array of 64 unsigned long long values. Offsets and initial constants for the scrambled sequence are of type unsigned int for 32-bit Sobol' generators. These parameters are of type unsigned long long for 64-bit Soblol' generators. For the `curandStateSobol32_t` type and the `curandStateScrambledSobol32_t` type the sequence is exactly 2^{32} elements long where each element is 32 bits. For the `curandStateSobol64_t` type and the `curandStateScrambledSobol64_t` type the sequence is exactly 2^{64} elements long where each element is 64 bits. Each call to `curand()` returns the next quasirandom

element. Calls to `curand_uniform()` return quasirandom floats or doubles from 0.0 to 1.0, where 1.0 is included and 0.0 is excluded. Similarly, calls to `curand_normal()` return normally distributed floats or doubles with mean 0.0 and standard deviation 1.0. Calls to `curand_log_normal()` return log-normally distributed floats or doubles, derived from the normal distribution with the specified mean and standard deviation. All of the generation functions may be called with any type of Sobol' generator.

As an example, generating quasirandom coordinates that fill a unit cube requires keeping track of three quasirandom generators. All three would start at `offset` $= 0$ and would have dimensions 0, 1, and 2, respectively. A single call to `curand_uniform()` for each generator state would generate the x, y, and z coordinates. Tables of direction vectors are accessible on the host through the `curandGetDirectionVectors32()` and `curandGetDirectionVectors64()` functions. The direction vectors needed should be copied into device memory before use.

The normal distribution functions for quasirandom generation use the inverse cumulative density function to preserve the dimensionality of the quasirandom sequence. Therefore there are no functions that generate more than one result at a time as there are with the pseudorandom generators.

The double precision Sobol32 functions return results in double precision that use 32 bits of internal precision from the underlying generator.

The double precision Sobol64 functions return results in double precision that use 53 bits of internal precision from the underlying generator. These bits are taken from the high order 53 bits of the 64 bit samples.

Skip-Ahead

There are several functions to skip ahead from a generator state.

```
__device__ void
skipahead (unsigned long long n, curandState_t *state)
```

```
__device__ void
skipahead (unsigned int n, curandStateSobol32_t *state)
```

Using this function is equivalent to calling `curand()` n times without using the return value, but it is much faster.

```
__device__ void
skipahead_sequence (unsigned long long n, curandState_t *state)
```

This function is the equivalent of calling `curand()` $n \cdot 2^{67}$ times without using the return value and is much faster.

Performance Notes

Calls to `curand_init()` are slower than calls to `curand()` or `curand_uniform()`. Large offsets to `curand_init()` take more time than smaller offsets. It is much faster to save and restore random generator state than to recalculate the starting state repeatedly.

As shown below, generator state can be stored in global memory between kernel launches, used in local memory for fast generation, and then stored back into global memory.

```
__global__ void example(curandState *global_state)
{
    curandState local_state;
    local_state = global_state[threadIdx.x];
    for(int i = 0; i < 10000; i++) {
        unsigned int x = curand(&local_state);
        ...
    }
    global_state[threadIdx.x] = local_state;
}
```

Initialization of the random generator state generally requires more registers and local memory than random number generation. It may be beneficial to separate calls to `curand_init()` and `curand()` into separate kernels for maximum performance.

State setup can be an expensive operation. One way to speed up the setup is to use different seeds for each thread and a constant sequence number of 0. This can be especially helpful if many generators need to be created. While faster to set up, this method provides less guarantees about the mathematical properties of the generated sequences. If there happens to be a bad interaction between the hash function that initializes the generator state from the seed and the periodicity of the generators, there might be threads with highly correlated outputs for some seed values. We do not know of any problem values; if they do exist they are likely to be rare.

Device API Example

This example uses the device API to calculate the proportion of pseudorandom integers that have the low bit set.

```
/*
 * This program uses the device CURAND API to calculate what
 * proportion of pseudo-random ints have low bit set.
```

```
 */
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>

#define CUDA_CALL(x) do { if((x) != cudaSuccess) { \
    printf("Error at %s:%d\n",__FILE__,__LINE__); \
    return EXIT_FAILURE;}} while(0)

__global__ void setup_kernel(curandState *state)
{
    int id = threadIdx.x + blockIdx.x * 64;
    /* Each thread gets same seed, a different sequence
       number, no offset */
    curand_init(1234, id, 0, &state[id]);
}

__global__ void generate_kernel(curandState *state,
                                int *result)
{
    int id = threadIdx.x + blockIdx.x * 64;
    int count = 0;
    unsigned int x;
    /* Copy state to local memory for efficiency */
    curandState localState = state[id];
    /* Generate pseudo-random unsigned ints */
    for(int n = 0; n < 100000; n++) {
        x = curand(&localState);
        /* Check if low bit set */
        if(x & 1) {
            count++;
        }
    }
    /* Copy state back to global memory */
    state[id] = localState;
    /* Store results */
    result[id] += count;
}

int main(int argc, char *argv[])
{
    int i, total;
    curandState *devStates;
```

```c
    int *devResults, *hostResults;

    /* Allocate space for results on host */
    hostResults = (int *)calloc(64 * 64, sizeof(int));

    /* Allocate space for results on device */
    CUDA_CALL(cudaMalloc((void **)&devResults, 64 * 64 *
              sizeof(int)));

    /* Set results to 0 */
    CUDA_CALL(cudaMemset(devResults, 0, 64 * 64 *
              sizeof(int)));

    /* Allocate space for prng states on device */
    CUDA_CALL(cudaMalloc((void **)&devStates, 64 * 64 *
              sizeof(curandState)));

    /* Setup prng states */
    setup_kernel<<<64, 64>>>(devStates);

    /* Generate and use pseudo-random  */
    for(i = 0; i < 10; i++) {
        generate_kernel<<<64, 64>>>(devStates, devResults);
    }

    /* Copy device memory to host */
    CUDA_CALL(cudaMemcpy(hostResults, devResults, 64 * 64 *
        sizeof(int), cudaMemcpyDeviceToHost));

    /* Show result */
    total = 0;
    for(i = 0; i < 64 * 64; i++) {
        total += hostResults[i];
    }
    printf("Fraction with low bit set was %10.13f\n",
        (float)total / (64.0f * 64.0f * 100000.0f * 10.0f));

    /* Cleanup */
    CUDA_CALL(cudaFree(devStates));
    CUDA_CALL(cudaFree(devResults));
    free(hostResults);
    return EXIT_SUCCESS;
}
```

Thrust and CURAND Example

The following example demonstrates mixing CURAND and Thrust. It is a minimally modified version of `monte_carlo.cu`, one of the standard Thrust examples. The example estimates π by randomly picking points in the unit square and calculating the distance to the origin to see if the points are in the quarter unit circle.

```
#include <thrust/iterator/counting_iterator.h>
#include <thrust/functional.h>
#include <thrust/transform_reduce.h>
#include <curand_kernel.h>

#include <iostream>
#include <iomanip>

// we could vary M & N to find the perf sweet spot

struct estimate_pi :
    public thrust::unary_function<unsigned int, float>
{
  __device__
  float operator()(unsigned int thread_id)
  {
    float sum = 0;
    unsigned int N = 10000; // samples per thread

    unsigned int seed = thread_id;

    curandState s;

    // seed a random number generator
    curand_init(seed, 0, 0, &s);

    // take N samples in a quarter circle
    for(unsigned int i = 0; i < N; ++i)
    {
      // draw a sample from the unit square
      float x = curand_uniform(&s);
      float y = curand_uniform(&s);

      // measure distance from the origin
      float dist = sqrtf(x*x + y*y);
```

```
      // add 1.0f if (u0,u1) is inside the quarter circle
      if(dist <= 1.0f)
        sum += 1.0f;
    }

    // multiply by 4 to get the area of the whole circle
    sum *= 4.0f;

    // divide by N
    return sum / N;
  }
};

int main(void)
{
  // use 30K independent seeds
  int M = 30000;

  float estimate = thrust::transform_reduce(
        thrust::counting_iterator<int>(0),
        thrust::counting_iterator<int>(M),
        estimate_pi(),
        0.0f,
        thrust::plus<float>());
  estimate /= M;

  std::cout << std::setprecision(3);
  std::cout << "pi is approximately ";
  std::cout << estimate << std::endl;
  return 0;
}
```

Testing

Sobol' sequences are generated using the direction vectors recommended by Joe and Kuo [2]. The scrambled Sobol' method is described in [3] and [4].

The XORWOW generator was proposed by Marsaglia [5] and has been tested using the TestU01 "Crush" framework of tests [6]. The full suite of NIST pseudorandomness tests [7] has also been run, though the focus has been on TestU01. The most rigorous the the TestU01 batteries is "BigCrush", which executes 106 statistical tests over the course of approximately 5 hours on a high-end CPU/GPU. The XORWOW generator passes all of the tests on most runs, but does produce occasional suspect statistics. Below is an examples of the summary output from a run that did not pass all tests, with the detail of the specific failure.

```
========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        curandXORWOW
 Number of statistics:   160
 Total CPU time:    05:17:59.63
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 ------------------------------------------------
 81  LinearComp, r = 29            1 - 7.1e-11
 ------------------------------------------------
 All other tests were passed

Detail from test 81:

scomp_LinearComp test:
------------------------------------------------
   N =  1,  n = 400020,  r = 29,    s = 1



------------------------------------------------
Number of degrees of freedom        :    12
Chi2 statistic for size of jumps     :     7.11
p-value of test                      :     0.85
```

```
--------------------------------------------------
Normal statistic for number of jumps   :   -6.41
p-value of test                         : 1 - 7.1e-11    *****
```

To put this into perspective, there is a table in [6] that gives the results of running various levels of the "Crush" tests on a broad selection of generators. Only a small number of generators pass all of the BigCrush tests. For example the widely-respected Mersenne twister [8] consistently fails two of the linear complexity tests.

The MRG32k3a generator was proposed in [9], with a specific implementation suggested in [10]. This generator passes all "BigCrush" tests frequently, with occasional marginal results similar to those shown below.

```
========= Summary results of BigCrush =========

 Version:           TestU01 1.2.3
 Generator:         curandMRG32k3a
 Number of statistics:  160
 Total CPU time:    07:14:55.41
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                        p-value
 --------------------------------------------------
 59  WeightDistrib, r = 0          5.2e-4
 --------------------------------------------------
 All other tests were passed

Detail from test 59:

svaria_WeightDistrib test:
--------------------------------------------------
   N =  1,  n = 20000000,  r =  0,  k = 256,  Alpha =     0,  Beta =  0.25


--------------------------------------------------
Number of degrees of freedom      :   67
Chi-square statistic              :   111.55
p-value of test                   :   5.2e-4    *****

--------------------------------------------------
CPU time used                     :  00:02:56.25
```

The MTGP32 generator is an adaptation of the work outlined in [1]. The MTGP32

generator exhibits some marginal results on "BigCrush". Below is an example.

```
========= Summary results of BigCrush =========

 Version:          TestU01 1.2.3
 Generator:        curandMtgp32Int
 Number of statistics:  160
 Total CPU time:   05:45:29.49
 The following tests gave p-values outside [0.001, 0.9990]:
 (eps  means a value < 1.0e-300):
 (eps1 means a value < 1.0e-15):

       Test                          p-value
 --------------------------------------------------
 12  CollisionOver, t = 21          0.9993
 --------------------------------------------------
 All other tests were passed

Detail from test 12:

smultin_MultinomialOver test:
--------------------------------------------------
   N = 30,  n = 20000000,  r = 28,   d =   4,   t = 21,
       Sparse =   TRUE

       GenerCell = smultin_GenerCellSerial
       Number of cells = d^t =      4398046511104
       Expected number per cell =  1 /  219902.33
       EColl = n^2 / (2k) =  45.47473509
       Hashing =   TRUE

       Collision test

       CollisionOver:   density = n / k =  1 /  219902.33
       Expected number of collisions = Mu =      45.47

--------------------------------------------------
Results of CollisionOver test:

POISSON approximation                    :
Expected number of collisions = N*Mu  :     1364.24
Observed number of collisions         :     1248
p-value of test                       :     0.9993     *****
```

```
-----------------------------
Total number of cells containing j balls

  j =  0                              :  131940795334368
  j =  1                              :         599997504
  j =  2                              :              1248
  j =  3                              :                 0
  j =  4                              :                 0
  j =  5                              :                 0


------------------------------------------------
CPU time used                     :  00:04:32.52
```

Testing of the normal distribution, with the each of the generators, has been done using the Pearson chi-squared test [11], [12], the Jarque-Bera test [13], the Kolmogorov-Smirnov test [14], [15], and the Anderson-Darling test [16].

Tests are run over the range +/- 6 standard deviations. Three Pearson tests are run, with cell counts 1000, 100, and 25. The test output has columns labeled PK for Pearson with 1000 cells, PC for Pearson with 100 cells, P25 for Pearson with 25 cells, JB for Jarque-Bera, KS for Kolmogorov-Smirnov, and AD for Anderson-Darling. The rejection criterion for each test is printed below the label.

The following tables are representative of the test output for statistical testing of the normal distribution for XORWOW, MRG32k3a, MTGP32, Sobol' 32-bit, and scrambled Sobol' 32-bit generators. The rows of each table represent the statistical results computed over successive sequences of 10000 samples.

XORWOW Generator:

```
      PK          PC          P25          JB          KS          AD
     <1058       <118         <33         <4.6       <0.0122      <.632

     ----------------------------------------------------------------------
     684.48120   58.97784    20.44693     2.84152     0.00540     0.32829
     686.37925   54.84938     7.79583     0.55109     0.00900     0.25832
     673.21437   69.15825    15.46540     0.30335     0.00872     0.26772
     568.26999   49.99519     8.85046     0.66624     0.00870     0.22939
     639.10690   84.23040    10.19753     0.19844     0.00542     0.27939
```

MRg32k3a Generator:

```
      PK          PC          P25          JB          KS          AD
     <1058       <118         <33         <4.6       <0.0122      <.632

     ----------------------------------------------------------------------
```

```
764.38500      74.48157      19.32716       1.50118       0.01103      0.60351
795.31006      74.15086      11.78414       1.15159       0.00821      0.35343
741.85426      91.88692      20.67103       2.34232       0.00900      0.61787
644.62093      70.68369      17.18277       0.32870       0.01243*     0.34630
806.02693      93.50691      23.10548       2.67340       0.00978      0.51466
```

MTGP32 Generator:

```
   PK            PC            P25           JB            KS           AD
  <1058         <118          <33           <4.6          <0.0122      <.632

-------------------------------------------------------------------------

  924.62604     110.19868     23.45811      0.86919       0.00519      0.33411
  708.76047     79.42919      20.67913      1.13427       0.01142      0.54632
  674.17713     65.80415      13.09834      1.07799       0.01040      0.23860
  733.35915     57.13829      17.66337      3.17017       0.01188      0.30864
  620.17297     50.39043      14.75682      0.57970       0.00845      0.28916
```

Sobol' 32-bit generator:

```
   PK            PC            P25           JB            KS           AD
  <1058         <118          <33           <4.6          <0.0122      <.632

-------------------------------------------------------------------------

  157.04578      6.47398       1.45802      0.19007       0.00024      0.00188
  243.82767     11.98164       1.34982      0.00668       0.00030      0.00086
  229.87234     10.40206       2.73912      0.04165       0.00036      0.00137
  290.29451     17.09013       3.25717      0.02583       0.00042      0.00172
  327.32072     19.22832       5.09510      0.00335       0.00036      0.00127
```

Scrambled Sobol' 32-bit generator:

```
   PK            PC            P25           JB            KS           AD
  <1058         <118          <33           <4.6          <0.0122      <.632

-------------------------------------------------------------------------

  255.80606     10.93180       1.33766      0.01226       0.00036      0.00112
  258.84244      8.45589       1.56766      0.04164       0.00036      0.00170
  585.34346     49.33610       5.32037      0.04069       0.00043      0.00208
  337.50312     27.64720       3.38925      0.01953       0.00041      0.00211
  729.56687     56.89682      32.89772      0.00911       0.00040      0.00204
```

Even though the log-normal distribution is closely derived from the normal distribution, it has also been tested using the Pearson chi-squared test and the Kolmogorov-Smirnov test.

The following tables are representative of the test output for statistical testing of the log normal distribution for XORWOW, MRG32k3a, MTGP32, Sobol' 32-bit, and scrambled Sobol' 32-bit generators.

XORWOW generator:

```
     PK            PC            P25           KS
    <1058         <118          <33           <0.0122
  ------------------------------------------------------

   1019.57936   105.63667     13.15820       0.00540
    991.93663    91.95369     20.46549       0.00900
    983.09678   115.34978     20.50434       0.00872
    966.45604   113.30013     24.54060       0.00870
    996.35262   111.50026     21.01332       0.00542
```

MRG32k3a generator:

```
     PK            PC            P25           KS
    <1058         <118          <33           <0.0122
  ------------------------------------------------------

   1000.00359    90.12428     22.82709       0.00826
    942.17843    81.16259     16.13670       0.00739
   1005.62148   102.29924     23.62705       0.00697
   1053.68391    98.75565     28.65422       0.01107
    998.38936   103.43649     19.26568       0.00803
```

MTGP32 generator:

```
     PK            PC            P25           KS
    <1058         <118          <33           <0.0122
  ------------------------------------------------------

   1010.18903    94.51850     17.98126       0.00771
    993.78319    76.86543     12.48859       0.00831
   1010.22068    63.76027     11.65743       0.00677
    963.33103    89.44369     17.96636       0.01200
    927.15616    75.85515     13.64221       0.00566
```

Sobol' 32-bit generator:

```
     PK            PC            P25           KS
    <1058         <118          <33           <0.0122
```

```
   --------------------------------------------------------

   289.42589       5.03327        0.48858        0.00024
   386.79860       6.57783        0.76902        0.00030
   355.04631       8.54472        1.12228        0.00036
   434.19211       9.54021        2.07006        0.00042
   343.57507      10.71571        0.42503        0.00036
```

Scrambled Sobol- 32-bit generator:

```
     PK              PC             P25             KS
    <1058           <118            <33           <0.0122

   --------------------------------------------------------

   354.55037        8.20727        0.24592        0.00036
   506.45280       12.93848        0.73323        0.00036
   451.96949       18.18903        0.69465        0.00043
   593.25666       16.55782        0.54769        0.00041
   423.05263       12.06600        0.53472        0.00040
```

Bibliography

[1] Mutsuo Saito. A variant of mersenne twister suitable for graphic processors. *arXiv:1005.4973v2 [cs.MS]*, Jun 2010.

[2] S. Joe and F. Y. Kuo. Remark on algorithm 659: Implementing sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software*, 29:49–57, March 2003.

[3] Jiri Matousek. Journal of complexity. *ACM Transactions on Mathematical Software*, 14(4):527–556, December 1998.

[4] Art B. Owen. Local antithetic sampling with scrambled nets. *The Annals of Statistics*, 36(5):2319–2343, 2008.

[5] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14), 2003. Available at http://www.jstatsoft.org/v08/i14/paper.

[6] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), August 2007. Available at http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf.

[7] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for the validation of random number generators and pseudorandom number generators for cryptographic applications. Special Publication 800-22 Revision 1a, National Institute of Standards and Technology, April 2010. Available at http://csrc.nist.gov/groups/ST/toolkit/rng/index.html.

[8] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[9] Pierre L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1), Jan-Feb 1999.

[10] Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6), Nov-Dec 2002.

[11] Karl Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine*, 50(302):157–175, July 1900.

[12] R. L. Placket. Karl Pearson and the chi-squared test. *International Statistics Review*, 51:59–72, 1983.

[13] Carlos M. Jarque and Anil K. Bera. Efficient tests for normality, homoscedasticity and serial independence of regression residuals. *Economics Letters*, 6(3):255–259, 1980.

[14] A. Kolmogorov. Sulla determinazione empirica di una legge di distribuzione. *G. Inst. Ital. Attuari*, 4(83), 1933.

[15] Frank J. Massey. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[16] T. W. Anderson and D. A. Darling. Asymptotic theory of certain "goodness-of-fit" criteria based on stochastic processes. *Annals of Mathematical Statistics*, 23(2):193–212, 1952.

CURAND Reference

Host API

Functions

▶ curandStatus_t curandCreateGenerator (curandGenerator_t ∗generator, curandRngType_t rng_type)

    Create new random number generator.

▶ curandStatus_t curandCreateGeneratorHost (curandGenerator_t ∗generator, curandRngType_t rng_type)

    Create new host CPU random number generator.

▶ curandStatus_t curandDestroyGenerator (curandGenerator_t generator)

    Destroy an existing generator.

▶ curandStatus_t curandGenerate (curandGenerator_t generator, unsigned int ∗outputPtr, size_t num)

    Generate 32-bit pseudo or quasirandom numbers.

▶ curandStatus_t curandGenerateLogNormal (curandGenerator_t generator, float ∗outputPtr, size_t n, float mean, float stddev)

    Generate log-normally distributed floats.

▶ curandStatus_t curandGenerateLogNormalDouble (curandGenerator_t generator, double ∗outputPtr, size_t n, double mean, double stddev)

    Generate log-normally distributed doubles.

▶ curandStatus_t curandGenerateLongLong (curandGenerator_t generator, unsigned long long ∗outputPtr, size_t num)

Generate 64-bit quasirandom numbers.

▶ curandStatus_t curandGenerateNormal (curandGenerator_t generator, float ∗outputPtr, size_t n, float mean, float stddev)

　　Generate normally distributed floats.

▶ curandStatus_t curandGenerateNormalDouble (curandGenerator_t generator, double ∗outputPtr, size_t n, double mean, double stddev)

　　Generate normally distributed doubles.

▶ curandStatus_t curandGenerateSeeds (curandGenerator_t generator)

　　Setup starting states.

▶ curandStatus_t curandGenerateUniform (curandGenerator_t generator, float ∗outputPtr, size_t num)

　　Generate uniformly distributed floats.

▶ curandStatus_t curandGenerateUniformDouble (curandGenerator_t generator, double ∗outputPtr, size_t num)

　　Generate uniformly distributed doubles.

▶ curandStatus_t curandGetDirectionVectors32 (curandDirectionVectors32_t ∗vectors[], curandDirectionVectorSet_t set)

　　Get direction vectors for 32-bit quasirandom number generation.

▶ curandStatus_t curandGetDirectionVectors64 (curandDirectionVectors64_t ∗vectors[], curandDirectionVectorSet_t set)

　　Get direction vectors for 64-bit quasirandom number generation.

▶ curandStatus_t curandGetScrambleConstants32 (unsigned int ∗∗constants)

　　Get scramble constants for 32-bit scrambled Sobol' .

▶ curandStatus_t curandGetScrambleConstants64 (unsigned long long ∗∗constants)

　　Get scramble constants for 64-bit scrambled Sobol' .

▶ curandStatus_t curandGetVersion (int ∗version)

　　Return the version number of the library.

▶ curandStatus_t curandSetGeneratorOffset (curandGenerator_t generator, unsigned long long offset)

Set the absolute offset of the pseudo or quasirandom number generator.

▶ curandStatus_t curandSetGeneratorOrdering (curandGenerator_t generator, curandOrdering_t order)

    Set the ordering of results of the pseudo or quasirandom number generator.

▶ curandStatus_t curandSetPseudoRandomGeneratorSeed (curandGenerator_t generator, unsigned long long seed)

    Set the seed value of the pseudo-random number generator.

▶ curandStatus_t curandSetQuasiRandomGeneratorDimensions (curandGenerator_t generator, unsigned int num_dimensions)

    Set the number of dimensions.

▶ curandStatus_t curandSetStream (curandGenerator_t generator, cudaStream_t stream)

    Set the current stream for CURAND kernel launches.

▶ enum curandDirectionVectorSet {

  CURAND_DIRECTION_VECTORS_32_JOEKUO6 = 101,

  CURAND_SCRAMBLED_DIRECTION_VECTORS_32_JOEKUO6 = 102,

  CURAND_DIRECTION_VECTORS_64_JOEKUO6 = 103,

  CURAND_SCRAMBLED_DIRECTION_VECTORS_64_JOEKUO6 = 104 }

▶ enum curandOrdering {

  CURAND_ORDERING_PSEUDO_BEST = 100,

  CURAND_ORDERING_PSEUDO_DEFAULT = 101,

  CURAND_ORDERING_PSEUDO_SEEDED = 102,

  CURAND_ORDERING_QUASI_DEFAULT = 201 }

▶ enum curandRngType { ,

  CURAND_RNG_PSEUDO_DEFAULT = 100,

  CURAND_RNG_PSEUDO_XORWOW = 101,

  CURAND_RNG_PSEUDO_MRG32K3A = 121,

  CURAND_RNG_PSEUDO_MTGP32 = 141,

  CURAND_RNG_QUASI_DEFAULT = 200,

  CURAND_RNG_QUASI_SOBOL32 = 201,

CURAND_RNG_QUASI_SCRAMBLED_SOBOL32 = 202,

CURAND_RNG_QUASI_SOBOL64 = 203,

CURAND_RNG_QUASI_SCRAMBLED_SOBOL64 = 204 }

▶ enum curandStatus {

CURAND_STATUS_SUCCESS = 0,

CURAND_STATUS_VERSION_MISMATCH = 100,

CURAND_STATUS_NOT_INITIALIZED = 101,

CURAND_STATUS_ALLOCATION_FAILED = 102,

CURAND_STATUS_TYPE_ERROR = 103,

CURAND_STATUS_OUT_OF_RANGE = 104,

CURAND_STATUS_LENGTH_NOT_MULTIPLE = 105,

CURAND_STATUS_DOUBLE_PRECISION_REQUIRED = 106,

CURAND_STATUS_LAUNCH_FAILURE = 201,

CURAND_STATUS_PREEXISTING_FAILURE = 202,

CURAND_STATUS_INITIALIZATION_FAILED = 203,

CURAND_STATUS_ARCH_MISMATCH = 204,

CURAND_STATUS_INTERNAL_ERROR = 999 }

▶ typedef unsigned int curandDirectionVectors32_t [32]

▶ typedef unsigned long long curandDirectionVectors64_t [64]

▶ typedef enum curandDirectionVectorSet curandDirectionVectorSet_t

▶ typedef struct curandGenerator_st ∗ curandGenerator_t

▶ typedef enum curandOrdering curandOrdering_t

▶ typedef enum curandRngType curandRngType_t

▶ typedef enum curandStatus curandStatus_t

Typedef Documentation

typedef unsigned int **curandDirectionVectors32_t[32]**

CURAND array of 32-bit direction vectors

typedef unsigned long long **curandDirectionVectors64_t[64]**

CURAND array of 64-bit direction vectors

typedef enum **curandDirectionVectorSet curandDirectionVec-torSet_t**

CURAND choice of direction vector set

typedef struct curandGenerator_st∗ **curandGenerator_t**

CURAND generator

typedef enum **curandOrdering curandOrdering_t**

CURAND orderings of results in memory

typedef enum **curandRngType curandRngType_t**

CURAND generator types

typedef enum **curandStatus curandStatus_t**

CURAND function call status types

Enumeration Type Documentation

enum **curandDirectionVectorSet**

CURAND choice of direction vector set
Enumerator:

>   ***CURAND_DIRECTION_VECTORS_32_JOEKUO6*** Specific set of 32-bit direction vectors generated from polynomials recommended by S. Joe and F. Y. Kuo, for up to 20,000 dimensions.

>   ***CURAND_SCRAMBLED_DIRECTION_VECTORS_32_JOEKUO6*** Specific set of 32-bit direction vectors generated from polynomials recommended by S. Joe and F. Y. Kuo, for up to 20,000 dimensions, and scrambled.

>   ***CURAND_DIRECTION_VECTORS_64_JOEKUO6*** Specific set of 64-bit direction vectors generated from polynomials recommended by S. Joe and F. Y. Kuo, for up to 20,000 dimensions.

>   ***CURAND_SCRAMBLED_DIRECTION_VECTORS_64_JOEKUO6*** Specific set of 64-bit direction vectors generated from polynomials recommended by S. Joe and F. Y. Kuo, for up to 20,000 dimensions, and scrambled.

enum **curandOrdering**

CURAND orderings of results in memory

Enumerator:

CURAND_ORDERING_PSEUDO_BEST Best ordering for pseudorandom results.

CURAND_ORDERING_PSEUDO_DEFAULT Specific default 4096 thread sequence for pseudorandom results.

CURAND_ORDERING_PSEUDO_SEEDED Specific seeding pattern for fast lower quality pseudorandom results.

CURAND_ORDERING_QUASI_DEFAULT Specific n-dimensional ordering for quasirandom results.

enum **curandRngType**

CURAND generator types

Enumerator:

CURAND_RNG_PSEUDO_DEFAULT Default pseudorandom generator.

CURAND_RNG_PSEUDO_XORWOW XORWOW pseudorandom generator.

CURAND_RNG_PSEUDO_MRG32K3A MRG32k3a pseudorandom generator.

CURAND_RNG_PSEUDO_MTGP32 Mersenne Twister pseudorandom generator.

CURAND_RNG_QUASI_DEFAULT Default quasirandom generator.

CURAND_RNG_QUASI_SOBOL32 Sobol32 quasirandom generator.

CURAND_RNG_QUASI_SCRAMBLED_SOBOL32 Scrambled Sobol32 quasirandom generator.

CURAND_RNG_QUASI_SOBOL64 Sobol64 quasirandom generator.

CURAND_RNG_QUASI_SCRAMBLED_SOBOL64 Scrambled Sobol64 quasirandom generator.

enum **curandStatus**

CURAND Host API datatypes CURAND function call status types

Enumerator:

CURAND_STATUS_SUCCESS No errors.

CURAND_STATUS_VERSION_MISMATCH Header file and linked library version do not match.

CURAND_STATUS_NOT_INITIALIZED Generator not initialized.

CURAND_STATUS_ALLOCATION_FAILED Memory allocation failed.

CURAND_STATUS_TYPE_ERROR Generator is wrong type.

CURAND_STATUS_OUT_OF_RANGE   Argument out of range.

CURAND_STATUS_LENGTH_NOT_MULTIPLE   Length requested is not a multple of dimension.

CURAND_STATUS_DOUBLE_PRECISION_REQUIRED   GPU does not have double precision required by MRG32k3a.

CURAND_STATUS_LAUNCH_FAILURE   Kernel launch failure.

CURAND_STATUS_PREEXISTING_FAILURE   Preexisting failure on library entry.

CURAND_STATUS_INITIALIZATION_FAILED   Initialization of CUDA failed.

CURAND_STATUS_ARCH_MISMATCH   Architecture mismatch, GPU does not support requested feature.

CURAND_STATUS_INTERNAL_ERROR   Internal library error.

Function Documentation

curandStatus_t curandCreateGenerator (curandGenerator_t * *generator,* curandRngType_t *rng_type*)

Creates a new random number generator of type `rng_type` and returns it in `*generator`.

Legal values for `rng_type` are:

- ▶ CURAND_RNG_PSEUDO_DEFAULT
- ▶ CURAND_RNG_PSEUDO_XORWOW
- ▶ CURAND_RNG_PSEUDO_MRG32K3A
- ▶ CURAND_RNG_PSEUDO_MTGP32
- ▶ CURAND_RNG_QUASI_DEFAULT
- ▶ CURAND_RNG_QUASI_SOBOL32
- ▶ CURAND_RNG_QUASI_SCRAMBLED_SOBOL32
- ▶ CURAND_RNG_QUASI_SOBOL64
- ▶ CURAND_RNG_QUASI_SCRAMBLED_SOBOL64

When `rng_type` is CURAND_RNG_PSEUDO_DEFAULT, the type chosen is CURAND_RNG_PSEUDO_XORWOW.

When `rng_type` is CURAND_RNG_QUASI_DEFAULT, the type chosen is CURAND_RNG_QUASI_SOBOL32.

The default values for `rng_type` = CURAND_RNG_PSEUDO_XORWOW are:

- ▶ `seed` $= 0$
- ▶ `offset` $= 0$

- ▶ ordering = CURAND_ORDERING_PSEUDO_DEFAULT

The default values for `rng_type` = CURAND_RNG_PSEUDO_MRG32K3A are:

- ▶ seed = 0
- ▶ offset = 0
- ▶ ordering = CURAND_ORDERING_PSEUDO_DEFAULT

The default values for `rng_type` = CURAND_RNG_PSEUDO_MTGP32 are:

- ▶ seed = 0
- ▶ offset = 0
- ▶ ordering = CURAND_ORDERING_PSEUDO_DEFAULT

The default values for `rng_type` = CURAND_RNG_QUASI_SOBOL32 are:

- ▶ dimensions = 1
- ▶ offset = 0
- ▶ ordering = CURAND_ORDERING_QUASI_DEFAULT

The default values for `rng_type` = CURAND_RNG_QUASI_SOBOL64 are:

- ▶ dimensions = 1
- ▶ offset = 0
- ▶ ordering = CURAND_ORDERING_QUASI_DEFAULT

The default values for `rng_type` =
CURAND_RNG_QUASI_SCRAMBBLED_SOBOL32 are:

- ▶ dimensions = 1
- ▶ offset = 0
- ▶ ordering = CURAND_ORDERING_QUASI_DEFAULT

The default values for `rng_type` = CURAND_RNG_QUASI_SCRAMBLED_SOBOL64
are:

- ▶ dimensions = 1
- ▶ offset = 0
- ▶ ordering = CURAND_ORDERING_QUASI_DEFAULT

Parameters:

generator - Pointer to generator

rng_type - Type of generator to create

Returns:

CURAND_STATUS_ALLOCATION_FAILED if memory could not be allocated

CURAND_STATUS_INITIALIZATION_FAILED if there was a problem setting up the GPU

CURAND_STATUS_VERSION_MISMATCH if the header file version does not match the dynamically linked library version

CURAND_STATUS_TYPE_ERROR if the value for `rng_type` is invalid

CURAND_STATUS_SUCCESS if generator was created successfully

curandStatus_t curandCreateGeneratorHost (curandGenerator_t * *generator*, curandRngType_t *rng_type*)

Creates a new host CPU random number generator of type `rng_type` and returns it in `*generator`.

Legal values for `rng_type` are:

- ▶ CURAND_RNG_PSEUDO_DEFAULT
- ▶ CURAND_RNG_PSEUDO_XORWOW
- ▶ CURAND_RNG_PSEUDO_MRG32K3A
- ▶ CURAND_RNG_PSEUDO_MTGP32
- ▶ CURAND_RNG_QUASI_DEFAULT
- ▶ CURAND_RNG_QUASI_SOBOL32

When `rng_type` is CURAND_RNG_PSEUDO_DEFAULT, the type chosen is CURAND_RNG_PSEUDO_XORWOW.

When `rng_type` is CURAND_RNG_QUASI_DEFAULT, the type chosen is CURAND_RNG_QUASI_SOBOL32.

The default values for `rng_type` = CURAND_RNG_PSEUDO_XORWOW are:

- ▶ `seed` = 0
- ▶ `offset` = 0
- ▶ `ordering` = CURAND_ORDERING_PSEUDO_DEFAULT

The default values for `rng_type` = CURAND_RNG_PSEUDO_MRG32K3A are:

- ▶ `seed` = 0
- ▶ `offset` = 0
- ▶ `ordering` = CURAND_ORDERING_PSEUDO_DEFAULT

The default values for `rng_type` = CURAND_RNG_PSEUDO_MTGP32 are:

- ▶ `seed` = 0
- ▶ `offset` = 0

► ordering = CURAND_ORDERING_PSEUDO_DEFAULT

The default values for `rng_type` = CURAND_RNG_QUASI_SOBOL32 are:

► `dimensions` = 1

► `offset` = 0

► ordering = CURAND_ORDERING_QUASI_DEFAULT

The default values for `rng_type` = CURAND_RNG_QUASI_SOBOL64 are:

► `dimensions` = 1

► `offset` = 0

► ordering = CURAND_ORDERING_QUASI_DEFAULT

The default values for `rng_type` = CURAND_RNG_QUASI_SCRAMBLED_SOBOL32 are:

► `dimensions` = 1

► `offset` = 0

► ordering = CURAND_ORDERING_QUASI_DEFAULT

The default values for `rng_type` = CURAND_RNG_QUASI_SCRAMBLED_SOBOL64 are:

► `dimensions` = 1

► `offset` = 0

► ordering = CURAND_ORDERING_QUASI_DEFAULT

Parameters:

generator - Pointer to generator

rng_type - Type of generator to create

Returns:

CURAND_STATUS_ALLOCATION_FAILED if memory could not be allocated
CURAND_STATUS_INITIALIZATION_FAILED if there was a problem setting up the GPU
CURAND_STATUS_VERSION_MISMATCH if the header file version does not match the dynamically linked library version
CURAND_STATUS_TYPE_ERROR if the value for `rng_type` is invalid
CURAND_STATUS_SUCCESS if generator was created successfully

curandStatus_t curandDestroyGenerator (curandGenerator_t generator)

Destroy an existing generator and free all memory associated with its state.

Parameters:

> **generator** - Generator to destroy

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_SUCCESS if generator was destroyed successfully

curandStatus_t curandGenerate (curandGenerator_t generator, unsigned int ∗ outputPtr, size_t num)

Use `generator` to generate `num` 32-bit results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 32-bit values with every bit random.

Parameters:

> **generator** - Generator to use
>
> **outputPtr** - Pointer to device memory to store CUDA-generated results, or Pointer to host memory to store CPU-generated resluts
>
> **num** - Number of random 32-bit values to generate

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from a previous kernel launch
> CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples is not a multiple of the quasirandom dimension
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
> CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGenerateLogNormal (curandGenerator_t generator, float ∗ outputPtr, size_t n, float mean, float stddev)

Use `generator` to generate `num` float results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the

results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 32-bit floating point values with log-normal distribution based on an associated normal distribution with mean `mean` and standard deviation `stddev`.

Normally distributed results are generated from pseudorandom generators with a Box-Muller transform, and so require `num` to be even. Quasirandom generators use an inverse cumulative distribution function to preserve dimensionality. The normally distributed results are transformed into log-normal distribution.

There may be slight numerical differences between results generated on the GPU with generators created with curandCreateGenerator() and results calculated on the CPU with generators created with curandCreateGeneratorHost(). These differences arise because of differences in results for transcendental functions. In addition, future versions of CURAND may use newer versions of the CUDA math library, so different versions of CURAND may give slightly different numerical values.

Parameters:

> *generator* - Generator to use
>
> *outputPtr* - Pointer to device memory to store CUDA-generated results, or Pointer to host memory to store CPU-generated resluts
>
> *n* - Number of floats to generate
>
> *mean* - Mean of associated normal distribution
>
> *stddev* - Standard deviation of associated normal distribution

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from a previous kernel launch
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
> CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples is not a multiple of the quasirandom dimension, or is not a multiple of two for pseudorandom generators
> CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGenerateLogNormalDouble (curandGenerator_t *generator,* double ∗ *outputPtr,* size_t *n,* double *mean,* double *stddev*)

Use `generator` to generate `num` double results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 64-bit floating point values with log-normal distribution based on an associated normal distribution with mean `mean` and standard deviation `stddev`.

Normally distributed results are generated from pseudorandom generators with a Box-Muller transform, and so require `num` to be even. Quasirandom generators use an inverse cumulative distribution function to preserve dimensionality. The normally distributed results are transformed into log-normal distribution.

There may be slight numerical differences between results generated on the GPU with generators created with curandCreateGenerator() and results calculated on the CPU with generators created with curandCreateGeneratorHost(). These differences arise because of differences in results for transcendental functions. In addition, future versions of CURAND may use newer versions of the CUDA math library, so different versions of CURAND may give slightly different numerical values.

Parameters:

> ***generator*** - Generator to use
>
> ***outputPtr*** - Pointer to device memory to store CUDA-generated results, or Pointer to host memory to store CPU-generated resluts
>
> ***n*** - Number of doubles to generate
>
> ***mean*** - Mean of normal distribution
>
> ***stddev*** - Standard deviation of normal distribution

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from a previous kernel launch
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
> CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples is not a multiple of the quasirandom dimension, or is not a multiple of two for pseudorandom generators
> CURAND_STATUS_ARCH_MISMATCH if the GPU does not support double precision
> CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGenerateLongLong (curandGenerator_t *generator,* unsigned long long ∗ *outputPtr,* size_t *num*)

Use `generator` to generate `num` 64-bit results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 64-bit values with every bit random.

Parameters:

> *generator* - Generator to use
>
> *outputPtr* - Pointer to device memory to store CUDA-generated results, or Pointer to host memory to store CPU-generated resluts
>
> *num* - Number of random 64-bit values to generate

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from a previous kernel launch
> CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples is not a multiple of the quasirandom dimension
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
> CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGenerateNormal (curandGenerator_t *generator*, float ∗ *outputPtr*, size_t *n*, float *mean*, float *stddev*)

Use `generator` to generate `num` float results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 32-bit floating point values with mean `mean` and standard deviation `stddev`.

Normally distributed results are generated from pseudorandom generators with a Box-Muller transform, and so require `num` to be even. Quasirandom generators use an inverse cumulative distribution function to preserve dimensionality.

There may be slight numerical differences between results generated on the GPU with generators created with curandCreateGenerator() and results calculated on the CPU with generators created with curandCreateGeneratorHost(). These differences arise because of differences in results for transcendental functions. In addition, future versions of CURAND may use newer versions of the CUDA math library, so different versions of CURAND may give slightly different numerical values.

Parameters:

> *generator* - Generator to use
>
> *outputPtr* - Pointer to device memory to store CUDA-generated results, or Pointer to host memory to store CPU-generated resluts
>
> *n* - Number of floats to generate
>
> *mean* - Mean of normal distribution
>
> *stddev* - Standard deviation of normal distribution

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from
> a previous kernel launch
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
> CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples
> is not a multiple of the quasirandom dimension, or is not a multiple of two for
> pseudorandom generators
> CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGenerateNormalDouble (curandGenerator_t *generator*, double * *outputPtr*, size_t *n*, double *mean*, double *stddev*)

Use `generator` to generate `num` double results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 64-bit floating point values with mean `mean` and standard deviation `stddev`.

Normally distributed results are generated from pseudorandom generators with a Box-Muller transform, and so require `num` to be even. Quasirandom generators use an inverse cumulative distribution function to preserve dimensionality.

There may be slight numerical differences between results generated on the GPU with generators created with curandCreateGenerator() and results calculated on the CPU with generators created with curandCreateGeneratorHost(). These differences arise because of differences in results for transcendental functions. In addition, future versions of CURAND may use newer versions of the CUDA math library, so different versions of CURAND may give slightly different numerical values.

Parameters:

> *generator* - Generator to use
>
> *outputPtr* - Pointer to device memory to store CUDA-generated results, or Pointer to
>   host memory to store CPU-generated resluts
>
> *n* - Number of doubles to generate
>
> *mean* - Mean of normal distribution
>
> *stddev* - Standard deviation of normal distribution

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from
> a previous kernel launch
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason

CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples is not a multiple of the quasirandom dimension, or is not a multiple of two for pseudorandom generators
CURAND_STATUS_ARCH_MISMATCH if the GPU does not support double precision
CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGenerateSeeds (curandGenerator_t *generator*)

Generate the starting state of the generator. This function is automatically called by generation functions such as curandGenerate() and curandGenerateUniform(). It can be called manually for performance testing reasons to separate timings for starting state generation and random number generation.

Parameters:

> *generator* - Generator to update

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from a previous kernel launch
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
> CURAND_STATUS_SUCCESS if the seeds were generated successfully

curandStatus_t curandGenerateUniform (curandGenerator_t *generator*, float ∗ *outputPtr*, size_t *num*)

Use `generator` to generate `num` float results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 32-bit floating point values between `0.0f` and `1.0f`, excluding `0.0f` and including `1.0f`.

Parameters:

> *generator* - Generator to use
>
> *outputPtr* - Pointer to device memory to store CUDA-generated results, or Pointer to host memory to store CPU-generated resluts
>
> *num* - Number of floats to generate

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created

CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from a previous kernel launch
CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples is not a multiple of the quasirandom dimension
CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGenerateUniformDouble (curandGenerator_t *generator*, double * *outputPtr*, size_t *num*)

Use `generator` to generate `num` double results into the device memory at `outputPtr`. The device memory must have been previously allocated and be large enough to hold all the results. Launches are done with the stream set using curandSetStream(), or the null stream if no stream has been set.

Results are 64-bit double precision floating point values between `0.0` and `1.0`, excluding `0.0` and including `1.0`.

Parameters:

> *generator* - Generator to use
>
> *outputPtr* - Pointer to device memory to store CUDA-generated results, or Pointer to host memory to store CPU-generated resluts
>
> *num* - Number of doubles to generate

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_PREEXISTING_FAILURE if there was an existing error from a previous kernel launch
> CURAND_STATUS_LAUNCH_FAILURE if the kernel launch failed for any reason
> CURAND_STATUS_LENGTH_NOT_MULTIPLE if the number of output samples is not a multiple of the quasirandom dimension
> CURAND_STATUS_ARCH_MISMATCH if the GPU does not support double precision
> CURAND_STATUS_SUCCESS if the results were generated successfully

curandStatus_t curandGetDirectionVectors32 (curandDirectionVectors32_t * *vectors*[ ], curandDirectionVectorSet_t *set*)

Get a pointer to an array of direction vectors that can be used for quasirandom number generation. The resulting pointer will reference an array of direction vectors in host memory.

The array contains vectors for many dimensions. Each dimension has 32 vectors. Each individual vector is an unsigned int.

Legal values for `set` are:

- ▶ CURAND_DIRECTION_VECTORS_32_JOEKUO6 (20,000 dimensions)
- ▶ CURAND_SCRAMBLED_DIRECTION_VECTORS_32_JOEKUO6 (20,000 dimensions)

Parameters:

vectors - Address of pointer in which to return direction vectors

set - Which set of direction vectors to use

Returns:

CURAND_STATUS_OUT_OF_RANGE if the choice of set is invalid
CURAND_STATUS_SUCCESS if the pointer was set successfully

curandStatus_t curandGetDirectionVectors64 (curandDirectionVectors64_t ∗ *vectors*[ ], curandDirectionVectorSet_t *set*)

Get a pointer to an array of direction vectors that can be used for quasirandom number generation. The resulting pointer will reference an array of direction vectors in host memory.

The array contains vectors for many dimensions. Each dimension has 64 vectors. Each individual vector is an unsigned long long.

Legal values for `set` are:

- ▶ CURAND_DIRECTION_VECTORS_64_JOEKUO6 (20,000 dimensions)
- ▶ CURAND_SCRAMBLED_DIRECTION_VECTORS_64_JOEKUO6 (20,000 dimensions)

Parameters:

vectors - Address of pointer in which to return direction vectors

set - Which set of direction vectors to use

Returns:

CURAND_STATUS_OUT_OF_RANGE if the choice of set is invalid
CURAND_STATUS_SUCCESS if the pointer was set successfully

curandStatus_t curandGetScrambleConstants32 (unsigned int ∗∗ *constants*)

Get a pointer to an array of scramble constants that can be used for quasirandom number generation. The resulting pointer will reference an array of unsinged ints in host memory.

The array contains constants for many dimensions. Each dimension has a single unsigned int constant.

Parameters:

> *constants* - Address of pointer in which to return scramble constants

Returns:

> CURAND_STATUS_SUCCESS if the pointer was set successfully

curandStatus_t curandGetScrambleConstants64 (unsigned long long ∗∗ *constants*)

Get a pointer to an array of scramble constants that can be used for quasirandom number generation. The resulting pointer will reference an array of unsinged long longs in host memory.

The array contains constants for many dimensions. Each dimension has a single unsigned long long constant.

Parameters:

> *constans* - Address of pointer in which to return scramble constants

Returns:

> CURAND_STATUS_SUCCESS if the pointer was set successfully

curandStatus_t curandGetVersion (int ∗ *version*)

Return in ∗`version` the version number of the dynamically linked CURAND library. The format is the same as CUDART_VERSION from the CUDA Runtime. The only supported configuration is CURAND version equal to CUDA Runtime version.

Parameters:

> *version* - CURAND library version

Returns:

> CURAND_STATUS_SUCCESS if the version number was successfully returned

curandStatus_t curandSetGeneratorOffset (curandGenerator_t *generator*, unsigned long long *offset*)

Set the absolute offset of the pseudo or quasirandom number generator.

All values of offset are valid. The offset position is absolute, not relative to the current position in the sequence.

Parameters:

>   *generator* - Generator to modify

>   *offset* - Absolute offset position

Returns:

>   CURAND_STATUS_NOT_INITIALIZED if the generator was never created
>   CURAND_STATUS_SUCCESS if generator offset was set successfully

curandStatus_t curandSetGeneratorOrdering (curandGenerator_t *generator*, curandOrdering_t *order*)

Set the ordering of results of the pseudo or quasirandom number generator.

Legal values of `order` for pseudorandom generators are:

>   ▶ CURAND_ORDERING_PSEUDO_DEFAULT

>   ▶ CURAND_ORDERING_PSEUDO_BEST

>   ▶ CURAND_ORDERING_PSEUDO_SEEDED

Legal values of `order` for quasirandom generators are:

>   ▶ CURAND_ORDERING_QUASI_DEFAULT

Parameters:

>   *generator* - Generator to modify

>   *order* - Ordering of results

Returns:

>   CURAND_STATUS_NOT_INITIALIZED if the generator was never created
>   CURAND_STATUS_OUT_OF_RANGE if the ordering is not valid
>   CURAND_STATUS_SUCCESS if generator ordering was set successfully

curandStatus_t curandSetPseudoRandomGeneratorSeed (curandGenerator_t *generator,* unsigned long long *seed*)

Set the seed value of the pseudorandom number generator. All values of seed are valid. Different seeds will produce different sequences. Different seeds will often not be statistically correlated with each other, but some pairs of seed values may generate sequences which are statistically correlated.

Parameters:

> ***generator*** - Generator to modify
>
> ***seed*** - Seed value

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_TYPE_ERROR if the generator is not a pseudorandom number generator
> CURAND_STATUS_SUCCESS if generator seed was set successfully

curandStatus_t curandSetQuasiRandomGeneratorDimensions (curandGenerator_t *generator,* unsigned int *num_dimensions*)

Set the number of dimensions to be generated by the quasirandom number generator.

Legal values for `num_dimensions` are 1 to 20000.

Parameters:

> ***generator*** - Generator to modify
>
> ***num_dimensions*** - Number of dimensions

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_OUT_OF_RANGE if num_dimensions is not valid
> CURAND_STATUS_TYPE_ERROR if the generator is not a quasirandom number generator
> CURAND_STATUS_SUCCESS if generator ordering was set successfully

curandStatus_t curandSetStream (curandGenerator_t *generator,* cudaStream_t *stream*)

Set the current stream for CURAND kernel launches. All library functions will use this stream until set again.

Parameters:

> **generator** - Generator to modify

> **stream** - Stream to use or NULL for null stream

Returns:

> CURAND_STATUS_NOT_INITIALIZED if the generator was never created
> CURAND_STATUS_SUCCESS if stream was set successfully

Device API

Typedefs

- ▶ typedef struct curandStateXORWOW curandState_t
- ▶ typedef struct curandStateMRG32k3a curandStateMRG32k3a_t
- ▶ typedef struct curandStateMtgp32 curandStateMtgp32_t
- ▶ typedef struct curandStateScrambledSobol32 curandStateScrambledSobol32_t
- ▶ typedef struct curandStateScrambledSobol64 curandStateScrambledSobol64_t
- ▶ typedef struct curandStateSobol32 curandStateSobol32_t
- ▶ typedef struct curandStateSobol64 curandStateSobol64_t
- ▶ typedef struct curandStateXORWOW curandStateXORWOW_t

Functions

- ▶ __device__ unsigned int curand (curandStateMtgp32_t ∗state)

    Return 32-bits of pseudorandomness from a mtgp32 generator.

- ▶ __device__ unsigned int curand (curandStateMRG32k3a_t ∗state)

    Return 32-bits of pseudorandomness from an MRG32k3a generator.

- ▶ __device__ unsigned long long curand (curandStateScrambledSobol64_t ∗state)

    Return 64-bits of quasirandomness from a scrambled Sobol64 generator.

- ▶ __device__ unsigned long long curand (curandStateSobol64_t ∗state)

    Return 64-bits of quasirandomness from a Sobol64 generator.

- ▶ __device__ unsigned int curand (curandStateScrambledSobol32_t ∗state)

    Return 32-bits of quasirandomness from a scrambled Sobol32 generator.

► __device__ unsigned int curand (curandStateSobol32_t *state)

    Return 32-bits of quasirandomness from a Sobol32 generator.

► __device__ unsigned int curand (curandStateXORWOW_t *state)

    Return 32-bits of pseudorandomness from an XORWOW generator.

► __device__ void curand_init (curandDirectionVectors64_t direction_vectors, unsigned long long scramble_c, unsigned long long offset, curandStateScrambledSobol64_t *state)

    Initialize Scrambled Sobol64 state.

► __device__ void curand_init (curandDirectionVectors64_t direction_vectors, unsigned long long offset, curandStateSobol64_t *state)

    Initialize Sobol64 state.

► __device__ void curand_init (curandDirectionVectors32_t direction_vectors, unsigned int scramble_c, unsigned int offset, curandStateScrambledSobol32_t *state)

    Initialize Scrambled Sobol32 state.

► __device__ void curand_init (curandDirectionVectors32_t direction_vectors, unsigned int offset, curandStateSobol32_t *state)

    Initialize Sobol32 state.

► __device__ void curand_init (unsigned long long seed, unsigned long long subsequence, unsigned long long offset, curandStateMRG32k3a_t *state)

    Initialize MRG32k3a state.

► __device__ void curand_init (unsigned long long seed, unsigned long long subsequence, unsigned long long offset, curandStateXORWOW_t *state)

    Initialize XORWOW state.

► __device__ float curand_log_normal (curandStateScrambledSobol64_t *state, float mean, float stddev)

    Return a log-normally distributed float from a scrambled Sobol64 generator.

► __device__ float curand_log_normal (curandStateSobol64_t *state, float mean, float stddev)

    Return a log-normally distributed float from a Sobol64 generator.

▶ _ _device_ _ float curand_log_normal (curandStateScrambledSobol32_t ∗state, float mean, float stddev)

 Return a log-normally distributed float from a scrambled Sobol32 generator.

▶ _ _device_ _ float curand_log_normal (curandStateSobol32_t ∗state, float mean, float stddev)

 Return a log-normally distributed float from a Sobol32 generator.

▶ _ _device_ _ float curand_log_normal (curandStateMtgp32_t ∗state, float mean, float stddev)

 Return a log-normally distributed float from an MTGP32 generator.

▶ _ _device_ _ float curand_log_normal (curandStateMRG32k3a_t ∗state, float mean, float stddev)

 Return a log-normally distributed float from an MRG32k3a generator.

▶ _ _device_ _ float curand_log_normal (curandStateXORWOW_t ∗state, float mean, float stddev)

 Return a log-normally distributed float from an XORWOW generator.

▶ _ _device_ _ float2 curand_log_normal2 (curandStateMRG32k3a_t ∗state, float mean, float stddev)

 Return two normally distributed floats from an MRG32k3a generator.

▶ _ _device_ _ float2 curand_log_normal2 (curandStateXORWOW_t ∗state, float mean, float stddev)

 Return two normally distributed floats from an XORWOW generator.

▶ _ _device_ _ double2 curand_log_normal2_double (curandStateMRG32k3a_t ∗state, double mean, double stddev)

 Return two log-normally distributed doubles from an MRG32k3a generator.

▶ _ _device_ _ double2 curand_log_normal2_double (curandStateXORWOW_t ∗state, double mean, double stddev)

 Return two log-normally distributed doubles from an XORWOW generator.

▶ _ _device_ _ double curand_log_normal_double (curandStateScrambledSobol64_t ∗state, double mean, double stddev)

 Return a log-normally distributed double from a scrambled Sobol64 generator.

► __device__ double curand_log_normal_double (curandStateSobol64_t ∗state, double mean, double stddev)

  Return a log-normally distributed double from a Sobol64 generator.

► __device__ double curand_log_normal_double (curandStateScrambledSobol32_t ∗state, double mean, double stddev)

  Return a log-normally distributed double from a scrambled Sobol32 generator.

► __device__ double curand_log_normal_double (curandStateSobol32_t ∗state, double mean, double stddev)

  Return a log-normally distributed double from a Sobol32 generator.

► __device__ double curand_log_normal_double (curandStateMtgp32_t ∗state, double mean, double stddev)

  Return a log-normally distributed double from an MTGP32 generator.

► __device__ double curand_log_normal_double (curandStateMRG32k3a_t ∗state, double mean, double stddev)

  Return a log-normally distributed double from an MRG32k3a generator.

► __device__ double curand_log_normal_double (curandStateXORWOW_t ∗state, double mean, double stddev)

  Return a log-normally distributed double from an XORWOW generator.

► __device__ float curand_mtgp32_single (curandStateMtgp32_t ∗state)

  Return a uniformly distributed float from a mtgp32 generator.

► __device__ float curand_mtgp32_single_specific (curandStateMtgp32_t ∗state)

  Return a uniformly distributed float from a specific position in a mtgp32 generator.

► __device__ unsigned int curand_mtgp32_specific (curandStateMtgp32_t ∗state, unsigned char index, unsigned char n)

  Return 32-bits of pseudorandomness from a specific position in a mtgp32 generator.

► __device__ float curand_normal (curandStateScrambledSobol64_t ∗state)

  Return a normally distributed float from a scrambled Sobol64 generator.

► __device__ float curand_normal (curandStateSobol64_t ∗state)

Return a normally distributed float from a Sobol64 generator.

▶ __device__ float curand_normal (curandStateScrambledSobol32_t *state)

  Return a normally distributed float from a scrambled Sobol32 generator.

▶ __device__ float curand_normal (curandStateSobol32_t *state)

  Return a normally distributed float from a Sobol32 generator.

▶ __device__ float curand_normal (curandStateMtgp32_t *state)

  Return a normally distributed float from a MTGP32 generator.

▶ __device__ float curand_normal (curandStateMRG32k3a_t *state)

  Return a normally distributed float from an MRG32k3a generator.

▶ __device__ float curand_normal (curandStateXORWOW_t *state)

  Return a normally distributed float from an XORWOW generator.

▶ __device__ float2 curand_normal2 (curandStateMRG32k3a_t *state)

  Return two normally distributed floats from an MRG32k3a generator.

▶ __device__ float2 curand_normal2 (curandStateXORWOW_t *state)

  Return two normally distributed floats from an XORWOW generator.

▶ __device__ double2 curand_normal2_double (curandStateMRG32k3a_t *state)

  Return two normally distributed doubles from an MRG32k3a generator.

▶ __device__ double2 curand_normal2_double (curandStateXORWOW_t *state)

  Return two normally distributed doubles from an XORWOW generator.

▶ __device__ double curand_normal_double (curandStateScrambledSobol64_t *state)

  Return a normally distributed double from a scrambled Sobol64 generator.

▶ __device__ double curand_normal_double (curandStateSobol64_t *state)

  Return a normally distributed double from a Sobol64 generator.

▶ __device__ double curand_normal_double (curandStateScrambledSobol32_t *state)

Return a normally distributed double from a scrambled Sobol32 generator.

▶ __device__ double curand_normal_double (curandStateSobol32_t *state)

Return a normally distributed double from an Sobol32 generator.

▶ __device__ double curand_normal_double (curandStateMtgp32_t *state)

Return a normally distributed double from an MTGP32 generator.

▶ __device__ double curand_normal_double (curandStateMRG32k3a_t *state)

Return a normally distributed double from an MRG32k3a generator.

▶ __device__ double curand_normal_double (curandStateXORWOW_t *state)

Return a normally distributed double from an XORWOW generator.

▶ __device__ float curand_uniform (curandStateScrambledSobol64_t *state)

Return a uniformly distributed float from a scrambled Sobol64 generator.

▶ __device__ float curand_uniform (curandStateSobol64_t *state)

Return a uniformly distributed float from a Sobol64 generator.

▶ __device__ float curand_uniform (curandStateScrambledSobol32_t *state)

Return a uniformly distributed float from a scrambled Sobol32 generator.

▶ __device__ float curand_uniform (curandStateSobol32_t *state)

Return a uniformly distributed float from a Sobol32 generator.

▶ __device__ float curand_uniform (curandStateMtgp32_t *state)

Return a uniformly distributed float from a MTGP32 generator.

▶ __device__ float curand_uniform (curandStateMRG32k3a_t *state)

Return a uniformly distributed float from an MRG32k3a generator.

▶ __device__ float curand_uniform (curandStateXORWOW_t *state)

Return a uniformly distributed float from an XORWOW generator.

▶ __device__ double curand_uniform_double (curandStateScrambledSobol64_t *state)

Return a uniformly distributed double from a scrambled Sobol64 generator.

▶ _ _device_ _ double curand_uniform_double (curandStateSobol64_t ∗state)

  Return a uniformly distributed double from a Sobol64 generator.

▶ _ _device_ _ double curand_uniform_double (curandStateScrambledSobol32_t ∗state)

  Return a uniformly distributed double from a scrambled Sobol32 generator.

▶ _ _device_ _ double curand_uniform_double (curandStateSobol32_t ∗state)

  Return a uniformly distributed double from a Sobol32 generator.

▶ _ _device_ _ double curand_uniform_double (curandStateMtgp32_t ∗state)

  Return a uniformly distributed double from a MTGP32 generator.

▶ _ _device_ _ double curand_uniform_double (curandStateMRG32k3a_t ∗state)

  Return a uniformly distributed double from an MRG32k3a generator.

▶ _ _device_ _ double curand_uniform_double (curandStateXORWOW_t ∗state)

  Return a uniformly distributed double from an XORWOW generator.

▶ template<typename T >
  _ _device_ _ void skipahead (unsigned long long n, T state)

  Update Sobol64 state to skip n *elements.*

▶ template<typename T >
  _ _device_ _ void skipahead (unsigned int n, T state)

  Update Sobol32 state to skip n *elements.*

▶ _ _device_ _ void skipahead (unsigned long long n, curandStateMRG32k3a_t ∗state)

  Update MRG32k3a state to skip n *elements.*

▶ _ _device_ _ void skipahead (unsigned long long n, curandStateXORWOW_t ∗state)

  Update XORWOW state to skip n *elements.*

▶ _ _device_ _ void skipahead_sequence (unsigned long long n, curandStateMRG32k3a_t ∗state)

  Update MRG32k3a state to skip ahead n *sequences.*

▶ _ _device_ _ void skipahead_sequence (unsigned long long n, curandStateXORWOW_t *state)

Update XORWOW state to skip ahead n *subsequences.*

▶ _ _device_ _ void skipahead_subsequence (unsigned long long n, curandStateMRG32k3a_t *state)

Update MRG32k3a state to skip ahead n *subsequences.*

Variables

▶ unsigned int mtgp32_params_fast::flt_tmp_tbl [16]
▶ unsigned int mtgp32_params_fast::mask
▶ unsigned char mtgp32_params_fast::poly_sha1 [21]
▶ int mtgp32_params_fast::pos
▶ int mtgp32_params_fast::sh1
▶ int mtgp32_params_fast::sh2
▶ unsigned int mtgp32_params_fast::tbl [16]
▶ unsigned int mtgp32_params_fast::tmp_tbl [16]

Typedef Documentation

typedef struct curandStateXORWOW **curandState_t**

Default RNG

typedef struct curandStateMRG32k3a **curandStateMRG32k3a_t**

CURAND MRG32K3A state

typedef struct curandStateMtgp32 **curandStateMtgp32_t**

CURAND MTGP32 state

typedef struct curandStateScrambledSobol32 **curandStateScrambledSobol32_t**

CURAND Scrambled Sobol32 state

typedef struct curandStateScrambledSobol64 **curandStateScrambled-Sobol64_t**

CURAND Scrambled Sobol64 state

typedef struct curandStateSobol32 **curandStateSobol32_t**

CURAND Sobol32 state

typedef struct curandStateSobol64 **curandStateSobol64_t**

CURAND Sobol64 state

typedef struct curandStateXORWOW **curandStateXORWOW_t**

CURAND XORWOW state

Function Documentation

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

Return 32-bits of pseudorandomness from the mtgp32 generator in `state`, increment position of generator by the number of threads in the block. Note the number of threads in the block can not exceed 256.

Parameters:

    state - Pointer to state to update

Returns:

    32-bits of pseudorandomness as an unsigned int, all bits valid to use.

__device__ unsigned int curand (**curandStateMRG32k3a_t** * *state*)

Return 32-bits of pseudorandomness from the MRG32k3a generator in `state`, increment position of generator by one.

Parameters:

    state - Pointer to state to update

Returns:

    32-bits of pseudorandomness as an unsigned int, all bits valid to use.

__device__ unsigned long long curand (**curandStateScrambledSobol64_t** ∗ *state*)

Return 64-bits of quasirandomness from the scrambled Sobol32 generator in `state`, increment position of generator by one.

Parameters:

> *state* - Pointer to state to update

Returns:

> 64-bits of quasirandomness as an unsigned long long, all bits valid to use.

__device__ unsigned long long curand (**curandStateSobol64_t** ∗ *state*)

Return 64-bits of quasirandomness from the Sobol64 generator in `state`, increment position of generator by one.

Parameters:

> *state* - Pointer to state to update

Returns:

> 64-bits of quasirandomness as an unsigned long long, all bits valid to use.

__device__ unsigned int curand (**curandStateScrambledSobol32_t** ∗ *state*)

Return 32-bits of quasirandomness from the scrambled Sobol32 generator in `state`, increment position of generator by one.

Parameters:

> *state* - Pointer to state to update

Returns:

> 32-bits of quasirandomness as an unsigned int, all bits valid to use.

__device__ unsigned int curand (**curandStateSobol32_t** ∗ *state*)

Return 32-bits of quasirandomness from the Sobol32 generator in `state`, increment position of generator by one.

Parameters:

> *state* - Pointer to state to update

Returns:

32-bits of quasirandomness as an unsigned int, all bits valid to use.

__device__ unsigned int curand (**curandStateXORWOW__t** ∗ *state*)

Return 32-bits of pseudorandomness from the XORWOW generator in `state`, increment position of generator by one.

Parameters:

state - Pointer to state to update

Returns:

32-bits of pseudorandomness as an unsigned int, all bits valid to use.

__device__ void curand_init (**curandDirectionVectors64__t** *direction_vectors*, unsigned long long *scramble_c*, unsigned long long *offset*, **curandStateScrambledSobol64__t** ∗ *state*)

Initialize Sobol64 state in `state` with the given `direction vectors` and `offset`.

The direction vector is a device pointer to an array of 64 unsigned long longs. All input values of `offset` are legal.

Parameters:

direction_vectors - Pointer to array of 64 unsigned long longs representing the direction vectors for the desired dimension

scramble_c Scramble constant

offset - Absolute offset into sequence

state - Pointer to state to initialize

__device__ void curand_init (**curandDirectionVectors64__t** *direction_vectors*, unsigned long long *offset*, **curandStateSobol64__t** ∗ *state*)

Initialize Sobol64 state in `state` with the given `direction vectors` and `offset`.

The direction vector is a device pointer to an array of 64 unsigned long longs. All input values of `offset` are legal.

Parameters:

direction_vectors - Pointer to array of 64 unsigned long longs representing the direction vectors for the desired dimension

offset - Absolute offset into sequence

state - Pointer to state to initialize

__device__ void curand_init (**curandDirectionVectors32_t** *direction_vectors,* unsigned int *scramble_c,* unsigned int *offset,* **curandStateScrambledSobol32_t** * *state*)

Initialize Sobol32 state in `state` with the given `direction vectors` and `offset`.

The direction vector is a device pointer to an array of 32 unsigned ints. All input values of `offset` are legal.

Parameters:

> *direction_vectors* - Pointer to array of 32 unsigned ints representing the direction vectors for the desired dimension
>
> *scramble_c* Scramble constant
>
> *offset* - Absolute offset into sequence
>
> *state* - Pointer to state to initialize

__device__ void curand_init (**curandDirectionVectors32_t** *direction_vectors,* unsigned int *offset,* **curandStateSobol32_t** * *state*)

Initialize Sobol32 state in `state` with the given `direction vectors` and `offset`.

The direction vector is a device pointer to an array of 32 unsigned ints. All input values of `offset` are legal.

Parameters:

> *direction_vectors* - Pointer to array of 32 unsigned ints representing the direction vectors for the desired dimension
>
> *offset* - Absolute offset into sequence
>
> *state* - Pointer to state to initialize

__device__ void curand_init (unsigned long long *seed,* unsigned long long *subsequence,* unsigned long long *offset,* **curandStateMRG32k3a_t** * *state*)

Initialize MRG32k3a state in `state` with the given `seed`, `subsequence`, and `offset`.

All input values of `seed`, `subsequence`, and `offset` are legal. `subsequence` will be truncated to 51 bits to avoid running into the next sequence

A value of 0 for `seed` sets the state to the values of the original published version of the `MRG32k3a` algorithm.

Parameters:

> **seed** - Arbitrary bits to use as a seed
>
> **subsequence** - Subsequence to start at
>
> **offset** - Absolute offset into sequence
>
> **state** - Pointer to state to initialize

__device__ void curand_init (unsigned long long *seed*, unsigned long long *subsequence*, unsigned long long *offset*, curandStateXORWOW_t ∗ *state*)

Initialize XORWOW state in `state` with the given `seed`, `subsequence`, and `offset`.

All input values of `seed`, `subsequence`, and `offset` are legal. Large values for `subsequence` and `offset` require more computation and so will take more time to complete.

A value of 0 for `seed` sets the state to the values of the original published version of the `xorwow` algorithm.

Parameters:

> **seed** - Arbitrary bits to use as a seed
>
> **subsequence** - Subsequence to start at
>
> **offset** - Absolute offset into sequence
>
> **state** - Pointer to state to initialize

__device__ float curand_log_normal (curandStateScrambledSobol64_t ∗ *state*, float *mean*, float *stddev*)

Return a single log-normally distributed float derived from a normal distribution with mean `mean` and standard deviation `stddev` from the scrambled Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results, then converts to log-normal distribution.

Parameters:

> **state** - Pointer to state to update
>
> **mean** - Mean of the related normal distribution
>
> **stddev** - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed float with mean `mean` and standard deviation `stddev`

__device__ float curand_log_normal (**curandStateSobol64_t** ∗ *state,* float *mean,* float *stddev*)

Return a single log-normally distributed float derived from a normal distribution with mean `mean` and standard deviation `stddev` from the Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results, then converts to log-normal distribution.

Parameters:

> *state* - Pointer to state to update
>
> *mean* - Mean of the related normal distribution
>
> *stddev* - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed float with mean `mean` and standard deviation `stddev`

__device__ float curand_log_normal (**curandStateScrambledSobol32_t** ∗ *state,* float *mean,* float *stddev*)

Return a single log-normally distributed float derived from a normal distribution with mean `mean` and standard deviation `stddev` from the scrambled Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate a normally distributed result, then transforms the result to log-normal.

Parameters:

> *state* - Pointer to state to update
>
> *mean* - Mean of the related normal distribution
>
> *stddev* - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed float with mean `mean` and standard deviation `stddev`

__device__ float curand_log_normal (**curandStateSobol32_t** ∗ *state,* float *mean,* float *stddev*)

Return a single log-normally distributed float derived from a normal distribution with mean `mean` and standard deviation `stddev` from the Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate a normally distributed result, then transforms the result to log-normal.

Parameters:

> ***state*** - Pointer to state to update
>
> ***mean*** - Mean of the related normal distribution
>
> ***stddev*** - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed float with mean `mean` and standard deviation `stddev`

__device__ float curand_log_normal (curandStateMtgp32_t ∗ *state*, float *mean*, float *stddev*)

Return a single log-normally distributed float derived from a normal distribution with mean `mean` and standard deviation `stddev` from the MTGP32 generator in `state`, increment position of generator.

The implementation uses the inverse cumulative distribution function to generate a normally distributed result, then transforms the result to log-normal.

Parameters:

> ***state*** - Pointer to state to update
>
> ***mean*** - Mean of the related normal distribution
>
> ***stddev*** - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed float with mean `mean` and standard deviation `stddev`

__device__ float curand_log_normal (curandStateMRG32k3a_t ∗ *state*, float *mean*, float *stddev*)

Return a single log-normally distributed float derived from a normal distribution with mean `mean` and standard deviation `stddev` from the MRG32k3a generator in `state`, increment position of generator by one.

The implementation uses a Box-Muller transform to generate two normally distributed results, transforms them to log-normal distribution, then returns them one at a time. See curand_log_normal2() for a more efficient version that returns both results at once.

Parameters:

> ***state*** - Pointer to state to update
>
> ***mean*** - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed float with mean `mean` and standard deviation `stddev`

__device__ float curand_log_normal (**curandStateXORWOW_t** $*$ *state,* float *mean,* float *stddev*)

Return a single log-normally distributed float derived from a normal distribution with mean `mean` and standard deviation `stddev` from the XORWOW generator in `state`, increment position of generator by one.

The implementation uses a Box-Muller transform to generate two normally distributed results, transforms them to log-normal distribution, then returns them one at a time. See curand_log_normal2() for a more efficient version that returns both results at once.

Parameters:

state - Pointer to state to update

mean - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed float with mean `mean` and standard deviation `stddev`

__device__ float2 curand_log_normal2 (**curandStateMRG32k3a_t** $*$ *state,* float *mean,* float *stddev*)

Return two log-normally distributed floats derived from a normal distribution with mean `mean` and standard deviation `stddev` from the MRG32k3a generator in `state`, increment position of generator by two.

The implementation uses a Box-Muller transform to generate two normally distributed results, then transforms them to log-normal.

Parameters:

state - Pointer to state to update

mean - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed float2 where each element is from a distribution with mean `mean` and standard deviation `stddev`

__device__ float2 curand_log_normal2 (**curandStateXORWOW_t** ∗ *state*, float *mean*, float *stddev*)

Return two log-normally distributed floats derived from a normal distribution with mean `mean` and standard deviation `stddev` from the XORWOW generator in `state`, increment position of generator by two.

The implementation uses a Box-Muller transform to generate two normally distributed results, then transforms them to log-normal.

Parameters:

state - Pointer to state to update

mean - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed float2 where each element is from a distribution with mean `mean` and standard deviation `stddev`

__device__ double2 curand_log_normal2_double (**curandStateMRG32k3a_t** ∗ *state*, double *mean*, double *stddev*)

Return two log-normally distributed doubles derived from a normal distribution with mean `mean` and standard deviation `stddev` from the MRG32k3a generator in `state`, increment position of generator by two.

The implementation uses a Box-Muller transform to generate two normally distributed results, and transforms them to log-normal distribution,.

Parameters:

state - Pointer to state to update

mean - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed double2 where each element is from a distribution with mean `mean` and standard deviation `stddev`

__device__ double2 curand_log_normal2_double (**curandStateXORWOW_t** ∗ *state*, double *mean*, double *stddev*)

Return two log-normally distributed doubles derived from a normal distribution with mean `mean` and standard deviation `stddev` from the XORWOW generator in `state`, increment position of generator by two.

The implementation uses a Box-Muller transform to generate two normally distributed results, and transforms them to log-normal distribution,.

Parameters:

> *state* - Pointer to state to update
>
> *mean* - Mean of the related normal distribution
>
> *stddev* - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed double2 where each element is from a distribution with mean `mean` and standard deviation `stddev`

__device__ double curand_log_normal_double (curandStateScrambledSobol64_t ∗ *state,* double *mean,* double *stddev*)

Return a single normally distributed double derived from a normal distribution with mean `mean` and standard deviation `stddev` from the scrambled Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> *state* - Pointer to state to update
>
> *mean* - Mean of the related normal distribution
>
> *stddev* - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed double with mean `mean` and standard deviation `stddev`

__device__ double curand_log_normal_double (curandStateSobol64_t ∗ *state,* double *mean,* double *stddev*)

Return a single normally distributed double derived from a normal distribution with mean `mean` and standard deviation `stddev` from the Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> *state* - Pointer to state to update
>
> *mean* - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed double with mean `mean` and standard deviation `stddev`

__device__ double curand_log_normal_double (**curandStateScrambledSobol32_t** ∗ *state,* double *mean,* double *stddev*)

Return a single log-normally distributed double derived from a normal distribution with mean `mean` and standard deviation `stddev` from the scrambled Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results, and transforms them into log-normal distribution.

Parameters:

state - Pointer to state to update

mean - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed double with mean `mean` and standard deviation `stddev`

__device__ double curand_log_normal_double (**curandStateSobol32_t** ∗ *state,* double *mean,* double *stddev*)

Return a single log-normally distributed double derived from a normal distribution with mean `mean` and standard deviation `stddev` from the Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results, and transforms them into log-normal distribution.

Parameters:

state - Pointer to state to update

mean - Mean of the related normal distribution

stddev - Standard deviation of the related normal distribution

Returns:

Log-normally distributed double with mean `mean` and standard deviation `stddev`

__device__ double curand_log_normal_double (curandStateMtgp32_t ∗ *state*, double *mean*, double *stddev*)

Return a single log-normally distributed double derived from a normal distribution with mean `mean` and standard deviation `stddev` from the MTGP32 generator in `state`, increment position of generator.

The implementation uses the inverse cumulative distribution function to generate normally distributed results, and transforms them into log-normal distribution.

Parameters:

> *state* - Pointer to state to update
>
> *mean* - Mean of the related normal distribution
>
> *stddev* - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed double with mean `mean` and standard deviation `stddev`

__device__ double curand_log_normal_double (curandStateMRG32k3a_t ∗ *state*, double *mean*, double *stddev*)

Return a single normally distributed double derived from a normal distribution with mean `mean` and standard deviation `stddev` from the MRG32k3a generator in `state`, increment position of generator.

The implementation uses a Box-Muller transform to generate two normally distributed results, transforms them to log-normal distribution, then returns them one at a time. See curand_log_normal2_double() for a more efficient version that returns both results at once.

Parameters:

> *state* - Pointer to state to update
>
> *mean* - Mean of the related normal distribution
>
> *stddev* - Standard deviation of the related normal distribution

Returns:

> Log-normally distributed double with mean `mean` and standard deviation `stddev`

__device__ double curand_log_normal_double (curandStateXORWOW_t ∗ *state*, double *mean*, double *stddev*)

Return a single normally distributed double derived from a normal distribution with mean `mean` and standard deviation `stddev` from the XORWOW generator in `state`, increment position of generator.

The implementation uses a Box-Muller transform to generate two normally distributed results, transforms them to log-normal distribution, then returns them one at a time. See curand_log_normal2_double() for a more efficient version that returns both results at once.

Parameters:

    state - Pointer to state to update

    mean - Mean of the related normal distribution

    stddev - Standard deviation of the related normal distribution

Returns:

    Log-normally distributed double with mean `mean` and standard deviation `stddev`

__device__ float curand_mtgp32_single ($\mathrm{curandStateMtgp32\_t} * state$)

Return a uniformly distributed float between `0.0f` and `1.0f` from the mtgp32 generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

Note: This alternate derivation of a uniform float is provided for completeness with the original source

Parameters:

    state - Pointer to state to update

Returns:

    uniformly distributed float between `0.0f` and `1.0f`

__device__ float curand_mtgp32_single_specific ($\mathrm{curandStateMtgp32\_t} *$ *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from position `index` of the mtgp32 generator in `state`, and increment position of generator by `n` positions, which must be the total number of positions upddated in the state by the thread block, for this invocation. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

Note 1: Thread indices must range from 0...`n` - 1. The number of positions updated may not exceed 256. A thread block may update more than one state, but a given state may not be updated by more than one thread block.

Note 2: This alternate derivation of a uniform float is provided for completeness with the original source

Parameters:

> *state* - Pointer to state to update
>
> *index* - Index (0..255) of the position within the state to draw from and update
>
> *n* - The total number of postions in this state that are being updated by this invocation

Returns:

> uniformly distributed float between `0.0f` and `1.0f`

__device__ unsigned int curand_mtgp32_specific (**curandStateMtgp32_t** ∗ *state,* unsigned char *index,* unsigned char *n*)

Return 32-bits of pseudorandomness from position `index` of the mtgp32 generator in `state`, increment position of generator by `n` positions, which must be the total number of positions upddated in the state by the thread block, for this invocation.

Note : Thread indices must range from 0... n - 1. The number of positions updated may not exceed 256. A thread block may update more than one state, but a given state may not be updated by more than one thread block.

Parameters:

> *state* - Pointer to state to update
>
> *index* - Index (0..255) of the position within the state to draw from and update
>
> *n* - The total number of postions in this state that are being updated by this invocation

Returns:

> 32-bits of pseudorandomness as an unsigned int, all bits valid to use.

__device__ float curand_normal (**curandStateScrambledSobol64_t** ∗ *state*)

Return a single normally distributed float with mean `0.0f` and standard deviation `1.0f` from the scrambled Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> *state* - Pointer to state to update

Returns:

> Normally distributed float with mean `0.0f` and standard deviation `1.0f`

__device__ float curand_normal (**curandStateSobol64_t** $*$ *state*)

Return a single normally distributed float with mean `0.0f` and standard deviation `1.0f` from the Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed float with mean `0.0f` and standard deviation `1.0f`

__device__ float curand_normal (**curandStateScrambledSobol32_t** $*$ *state*)

Return a single normally distributed float with mean `0.0f` and standard deviation `1.0f` from the scrambled Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed float with mean `0.0f` and standard deviation `1.0f`

__device__ float curand_normal (**curandStateSobol32_t** $*$ *state*)

Return a single normally distributed float with mean `0.0f` and standard deviation `1.0f` from the Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed float with mean `0.0f` and standard deviation `1.0f`

__device__ float curand_normal (**curandStateMtgp32_t** ∗ *state*)

Return a single normally distributed float with mean `0.0f` and standard deviation `1.0f` from the MTGP32 generator in `state`, increment position of generator.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed float with mean `0.0f` and standard deviation `1.0f`

__device__ float curand_normal (**curandStateMRG32k3a_t** ∗ *state*)

Return a single normally distributed float with mean `0.0f` and standard deviation `1.0f` from the MRG32k3a generator in `state`, increment position of generator by one.

The implementation uses a Box-Muller transform to generate two normally distributed results, then returns them one at a time. See curand_normal2() for a more efficient version that returns both results at once.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed float with mean `0.0f` and standard deviation `1.0f`

__device__ float curand_normal (**curandStateXORWOW_t** ∗ *state*)

Return a single normally distributed float with mean `0.0f` and standard deviation `1.0f` from the XORWOW generator in `state`, increment position of generator by one.

The implementation uses a Box-Muller transform to generate two normally distributed results, then returns them one at a time. See curand_normal2() for a more efficient version that returns both results at once.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed float with mean `0.0f` and standard deviation `1.0f`

__device__ float2 curand_normal2 (**curandStateMRG32k3a_t** ∗ *state*)

Return two normally distributed floats with mean `0.0f` and standard deviation `1.0f` from the MRG32k3a generator in `state`, increment position of generator by two.

The implementation uses a Box-Muller transform to generate two normally distributed results.

Parameters:

> *state* - Pointer to state to update

Returns:

> Normally distributed float2 where each element is from a distribution with mean `0.0f` and standard deviation `1.0f`

__device__ float2 curand_normal2 (**curandStateXORWOW_t** ∗ *state*)

Return two normally distributed floats with mean `0.0f` and standard deviation `1.0f` from the XORWOW generator in `state`, increment position of generator by two.

The implementation uses a Box-Muller transform to generate two normally distributed results.

Parameters:

> *state* - Pointer to state to update

Returns:

> Normally distributed float2 where each element is from a distribution with mean `0.0f` and standard deviation `1.0f`

__device__ double2 curand_normal2_double (**curandStateMRG32k3a_t** ∗ *state*)

Return two normally distributed doubles with mean `0.0` and standard deviation `1.0` from the MRG32k3a generator in `state`, increment position of generator.

The implementation uses a Box-Muller transform to generate two normally distributed results.

Parameters:

> *state* - Pointer to state to update

Returns:

> Normally distributed double2 where each element is from a distribution with mean `0.0` and standard deviation `1.0`

__device__ double2 curand_normal2_double (**curandStateXORWOW _ t** ∗ *state*)

Return two normally distributed doubles with mean `0.0` and standard deviation `1.0` from the XORWOW generator in `state`, increment position of generator.

The implementation uses a Box-Muller transform to generate two normally distributed results.

Parameters:

   state - Pointer to state to update

Returns:

   Normally distributed double2 where each element is from a distribution with mean `0.0` and standard deviation `1.0`

__device__ double curand_normal_double (**curandStateScrambled-Sobol64 _ t** ∗ *state*)

Return a single normally distributed double with mean `0.0` and standard deviation `1.0` from the scrambled Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

   state - Pointer to state to update

Returns:

   Normally distributed double with mean `0.0` and standard deviation `1.0`

__device__ double curand_normal_double (**curandStateSobol64 _ t** ∗ *state*)

Return a single normally distributed double with mean `0.0` and standard deviation `1.0` from the Sobol64 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

   state - Pointer to state to update

Returns:

   Normally distributed double with mean `0.0` and standard deviation `1.0`

__device__ double curand_normal_double (curandStateScrambled-Sobol32_t * *state*)

Return a single normally distributed double with mean `0.0` and standard deviation `1.0` from the scrambled Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed double with mean `0.0` and standard deviation `1.0`

__device__ double curand_normal_double (curandStateSobol32_t * *state*)

Return a single normally distributed double with mean `0.0` and standard deviation `1.0` from the Sobol32 generator in `state`, increment position of generator by one.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed double with mean `0.0` and standard deviation `1.0`

__device__ double curand_normal_double (curandStateMtgp32_t * *state*)

Return a single normally distributed double with mean `0.0` and standard deviation `1.0` from the MTGP32 generator in `state`, increment position of generator.

The implementation uses the inverse cumulative distribution function to generate normally distributed results.

Parameters:

> ***state*** - Pointer to state to update

Returns:

> Normally distributed double with mean `0.0` and standard deviation `1.0`

__device__ double curand_normal_double (**curandStateMRG32k3a_t ∗** *state*)

Return a single normally distributed double with mean `0.0` and standard deviation `1.0` from the XORWOW generator in `state`, increment position of generator.

The implementation uses a Box-Muller transform to generate two normally distributed results, then returns them one at a time. See curand_normal2_double() for a more efficient version that returns both results at once.

Parameters:

> *state* - Pointer to state to update

Returns:

> Normally distributed double with mean `0.0` and standard deviation `1.0`

__device__ double curand_normal_double (**curandStateXORWOW_t ∗** *state*)

Return a single normally distributed double with mean `0.0` and standard deviation `1.0` from the XORWOW generator in `state`, increment position of generator.

The implementation uses a Box-Muller transform to generate two normally distributed results, then returns them one at a time. See curand_normal2_double() for a more efficient version that returns both results at once.

Parameters:

> *state* - Pointer to state to update

Returns:

> Normally distributed double with mean `0.0` and standard deviation `1.0`

__device__ float curand_uniform (**curandStateScrambledSobol64_t ∗** *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from the scrambled Sobol64 generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()`.

Parameters:

> *state* - Pointer to state to update

Returns:

uniformly distributed float between `0.0f` and `1.0f`

__device__ float curand_uniform (**curandStateSobol64_t** ∗ *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from the Sobol64 generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()`.

Parameters:

state - Pointer to state to update

Returns:

uniformly distributed float between `0.0f` and `1.0f`

__device__ float curand_uniform (**curandStateScrambledSobol32_t** ∗ *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from the scrambled Sobol32 generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()`.

Parameters:

state - Pointer to state to update

Returns:

uniformly distributed float between `0.0f` and `1.0f`

__device__ float curand_uniform (**curandStateSobol32_t** ∗ *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from the Sobol32 generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()`.

Parameters:

state - Pointer to state to update

Returns:

> uniformly distributed float between `0.0f` and `1.0f`

__device__ float curand_uniform (**curandStateMtgp32_t** ∗ *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from the MTGP32 generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

Parameters:

> **state** - Pointer to state to update

Returns:

> uniformly distributed float between `0.0f` and `1.0f`

__device__ float curand_uniform (**curandStateMRG32k3a_t** ∗ *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from the MRG32k3a generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

The implementation returns up to 23 bits of mantissa, with the minimum return value 2^{-32}

Parameters:

> **state** - Pointer to state to update

Returns:

> uniformly distributed float between `0.0f` and `1.0f`

__device__ float curand_uniform (**curandStateXORWOW_t** ∗ *state*)

Return a uniformly distributed float between `0.0f` and `1.0f` from the XORWOW generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

The implementation may use any number of calls to `curand()` to get enough random bits to create the return value. The current implementation uses one call.

Parameters:

> **state** - Pointer to state to update

Returns:

> uniformly distributed float between `0.0f` and `1.0f`

__device__ double curand_uniform_double (curandStateScrambled-Sobol64_t *state)

Return a uniformly distributed double between `0.0` and `1.0` from the scrambled Sobol64 generator in `state`, increment position of generator. Output range excludes `0.0` but includes `1.0`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()` to preserve the quasirandom properties of the sequence.

Parameters:

state - Pointer to state to update

Returns:

uniformly distributed double between `0.0` and `1.0`

__device__ double curand_uniform_double (curandStateSobol64_t *state)

Return a uniformly distributed double between `0.0` and `1.0` from the Sobol64 generator in `state`, increment position of generator. Output range excludes `0.0` but includes `1.0`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()` to preserve the quasirandom properties of the sequence.

Parameters:

state - Pointer to state to update

Returns:

uniformly distributed double between `0.0` and `1.0`

__device__ double curand_uniform_double (curandStateScrambled-Sobol32_t *state)

Return a uniformly distributed double between `0.0` and `1.0` from the scrambled Sobol32 generator in `state`, increment position of generator. Output range excludes `0.0` but includes `1.0`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()` to preserve the quasirandom properties of the sequence.

Parameters:

state - Pointer to state to update

Returns:

    uniformly distributed double between `0.0` and `1.0`

__device__ double curand_uniform_double (**curandStateSobol32_t** ∗ *state*)

Return a uniformly distributed double between `0.0` and `1.0` from the Sobol32 generator in `state`, increment position of generator. Output range excludes `0.0` but includes `1.0`. Denormalized floating point outputs are never returned.

The implementation is guaranteed to use a single call to `curand()` to preserve the quasirandom properties of the sequence.

Parameters:

    state - Pointer to state to update

Returns:

    uniformly distributed double between `0.0` and `1.0`

__device__ double curand_uniform_double (**curandStateMtgp32_t** ∗ *state*)

Return a uniformly distributed double between `0.0f` and `1.0f` from the MTGP32 generator in `state`, increment position of generator. Output range excludes `0.0f` but includes `1.0f`. Denormalized floating point outputs are never returned.

Parameters:

    state - Pointer to state to update

Returns:

    uniformly distributed double between `0.0f` and `1.0f`

__device__ double curand_uniform_double (**curandStateMRG32k3a_t** ∗ *state*)

Return a uniformly distributed double between `0.0` and `1.0` from the MRG32k3a generator in `state`, increment position of generator. Output range excludes `0.0` but includes `1.0`. Denormalized floating point outputs are never returned.

Note the implementation returns at most 32 random bits of mantissa as outlined in the seminal paper by L'Ecuyer.

Parameters:

    state - Pointer to state to update

Returns:

> uniformly distributed double between `0.0` and `1.0`

__device__ double curand_uniform_double (**curandStateXORWOW_t** ∗ *state*)

Return a uniformly distributed double between `0.0` and `1.0` from the XORWOW generator in `state`, increment position of generator. Output range excludes `0.0` but includes `1.0`. Denormalized floating point outputs are never returned.

The implementation may use any number of calls to `curand()` to get enough random bits to create the return value. The current implementation uses exactly two calls.

Parameters:

> *state* - Pointer to state to update

Returns:

> uniformly distributed double between `0.0` and `1.0`

template<typename T > __device__ void skipahead (unsigned long long *n*, T *state*)

Update the Sobol64 state in `state` to skip ahead `n` elements.

All values of `n` are valid.

Parameters:

> *n* - Number of elements to skip
>
> *state* - Pointer to state to update

template<typename T > __device__ void skipahead (unsigned int *n*, T *state*)

Update the Sobol32 state in `state` to skip ahead `n` elements.

All values of `n` are valid.

Parameters:

> *n* - Number of elements to skip
>
> *state* - Pointer to state to update

__device__ void skipahead (unsigned long long *n*, **curandStateMRG32k3a_t** ∗ *state*)

Update the MRG32k3a state in `state` to skip ahead `n` elements.

All values of **n** are valid. Large values require more computation and so will take more time to complete.

Parameters:

> **n** - Number of elements to skip
>
> **state** - Pointer to state to update

__device__ void skipahead (unsigned long long *n,* $\mathbf{curandStateXORWOW\_t}$ $*$ *state*)

Update the XORWOW state in `state` to skip ahead **n** elements.

All values of **n** are valid. Large values require more computation and so will take more time to complete.

Parameters:

> **n** - Number of elements to skip
>
> **state** - Pointer to state to update

__device__ void skipahead_sequence (unsigned long long *n,* $\mathbf{curandStateMRG32k3a\_t} * state$)

Update the MRG32k3a state in `state` to skip ahead **n** sequences. Each sequence is 2^{127} elements long, so this means the function will skip ahead $2^{127} \cdot n$ elements.

All values of **n** are valid. Large values require more computation and so will take more time to complete.

Parameters:

> **n** - Number of sequences to skip
>
> **state** - Pointer to state to update

__device__ void skipahead_sequence (unsigned long long *n,* $\mathbf{curandStateXORWOW\_t} * state$)

Update the XORWOW state in `state` to skip ahead **n** subsequences. Each subsequence is 2^{67} elements long, so this means the function will skip ahead $2^{67} \cdot n$ elements.

All values of **n** are valid. Large values require more computation and so will take more time to complete.

Parameters:

> **n** - Number of subsequences to skip

state - Pointer to state to update

__device__ void skipahead_subsequence (unsigned long long *n*, **curandStateMRG32k3a_t** ∗ *state*)

Update the MRG32k3a state in `state` to skip ahead `n` subsequences. Each subsequence is 2^{76} elements long, so this means the function will skip ahead $2^{76} \cdot n$ elements.

Valid values of `n` are 0 to $2^5 1$. Note `n` will be masked to 51 bits

Parameters:

n - Number of subsequences to skip

state - Pointer to state to update

Variable Documentation

unsigned int mtgp32_params_fast::flt_tmp_tbl[16] `[inherited]`

a small matrix for tempering and converting to float.

unsigned int mtgp32_params_fast::mask `[inherited]`

This is a mask for state space

unsigned char mtgp32_params_fast::poly_sha1[21] `[inherited]`

SHA1 digest

int mtgp32_params_fast::pos `[inherited]`

pick up position.

int mtgp32_params_fast::sh1 `[inherited]`

shift value 1. $0 < \text{sh1} < 32$.

int mtgp32_params_fast::sh2 `[inherited]`

shift value 2. $0 < \text{sh2} < 32$.

unsigned int mtgp32_params_fast::tbl[16] `[inherited]`

a small matrix.

unsigned int mtgp32_params_fast::tmp_tbl[16] `[inherited]`

a small matrix for tempering.