CUDA

CURAND Library

PG-05328-032_V01
August, 2010

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

Notice

Trademarks

Copyright

Table of Contents

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 pseudo-random or quasirandom numbers. The normal sequence of operations is as follows:

1. Create a new generator of the desired type (see Generator Types on page 8) with **curandCreateGenerator()**.
2. Set the generator options (see Generator Options on page 8); 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 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 two types of random number generators in CURAND. Type **CURAND_RNG_XORWOW** is a pseudorandom number generator implemented using the XORWOW algorithm, a member of the xor-shift family of pseudorandom number generators. **CURAND_RNG_SOBOL32** is a quasirandom number generator type. It is a Sobol' generator of 32-bit sequences in up to 20,000 dimensions.

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.

Order

The order parameter is used to choose how the results are ordered in global memory. There are two ordering choice for pseudorandom sequences: **CURAND_ORDERING_PSEUDO_DEFAULT** and **CURAND_ORDERING_PSEUDO_BEST**. 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, both pseudorandom number choices produce the same output ordering. 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.

The behavior of the three ordering parameters is summarized below:

- **CURAND_ORDERING_PSEUDO_BEST** (pseudorandom numbers)
  The output ordering of **CURAND_ORDERING_PSEUDO_BEST** is the same as **CURAND_ORDERING_PSEUDO_DEFAULT** in the current release.
- **CURAND_ORDERING_PSEUDO_DEFAULT** (pseudorandom numbers)
  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_QUASI_DEFAULT** (quasirandom numbers)
  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. Each output element is a 32-bit unsigned int 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.

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.

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

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

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

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;
}
```

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 Sequences and Quasirandom Sequences.

Pseudorandom Sequences

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

Bit Generation

```
__device__ unsigned int
curand (curandState *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 *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 optimal 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.

Distributions

```
__device__ float
curand_uniform (curandState *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 *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__ double
curand_uniform_double (curandState *state)
```

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

The two functions above are the double precision versions of **curand_uniform()** and **curand_normal()**. The double precisions pseudorandom functions use multiple calls to **curand()** to generate 53 random bits.

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

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

The above functions generate two 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 XOR-WOW, Sobol' sequences can be generated using the following functions:

```
__device__ void
curand_init (
    unsigned int *direction_vectors,
    unsigned int offset,
    curandStateSobol32 *state)

__device__ unsigned int
curand (curandStateSobol32 *state)

__device__ float
curand_uniform (curandStateSobol32 *state)

__device__ float
curand_normal (curandStateSobol32 *state)

__device__ double
curand_uniform_double (curandStateSobol32 *state)

__device__ double
curand_normal_double (curandStateSobol32 *state)
```

The **curand_init()** function sets up a quasirandom number generator state of type **curandStateSobol32**. There is no seed parameter, only direction vectors and offset. The direction vectors are an array of 32 unsigned integer

values. For the **curandStateSobol32** type, the sequence is exactly 2^{32} elements long where each element is 32 bits. Each call to **curand()** returns the next quasirandom element. Calls to **curand_uniform()** return quasirandom floats 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 with mean 0.0 and standard deviation 1.0.

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()** function, see Section 2.1.3.10. 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.

Skip-Ahead

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

```
__device__ void
skipahead (unsigned long long n, curandState *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 *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 much 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.

On Fermi architectures and later, users should explicitly set the stack size with **cudaThreadSetLimit(cudaLimitStackSize, size)** to avoid stack overflow problems. The default stack size is 1KB per thread. Initializing

random number generator state can require in the worst case up to 16KB of local memory space allocated on the stack per thread. The stack size set should be large enough to accomodate this need in addition to any stack space that may be used by the calling kernel. Once the state has been set up, random number generation itself requires very little stack space.

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;
    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;
}
```

References

The XORWOW generator was proposed by Marsaglia [1] and has been tested using the TestU01 "Crush" framework of tests [2]. The full suite of NIST pseudorandomness tests [3] has also been run. Sobol' sequences are generated using the direction vectors recommended by Joe and Kuo [4].

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

[2] 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.

[3] 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.

[4] 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.

CURAND Reference

2.1 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 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 quasirandom number generation.

- 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 }
- enum curandOrdering {
  CURAND_ORDERING_PSEUDO_BEST = 100,
  CURAND_ORDERING_PSEUDO_DEFAULT = 101,
  CURAND_ORDERING_QUASI_DEFAULT = 201 }
- enum curandRngType {
  CURAND_RNG_PSEUDO_DEFAULT = 100,
  CURAND_RNG_PSEUDO_XORWOW = 101,

CURAND_RNG_QUASI_DEFAULT = 200,
CURAND_RNG_QUASI_SOBOL32 = 201 }
- 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_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 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

2.1.1 Typedef Documentation

2.1.1.1 typedef unsigned int curandDirectionVectors32_t[32]

CURAND array of 32-bit direction vectors

2.1.1.2 typedef enum curandDirectionVectorSet curandDirectionVectorSet_t

CURAND choice of direction vector set

2.1.1.3 typedef struct curandGenerator_st* curandGenerator_t

CURAND generator

2.1.1.4 `typedef enum` curandOrdering curandOrdering_t

CURAND orderings of results in memory

2.1.1.5 `typedef enum` curandRngType curandRngType_t

CURAND generator types

2.1.1.6 `typedef enum` curandStatus curandStatus_t

CURAND function call status types

2.1.2 Enumeration Type Documentation

2.1.2.1 `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.

2.1.2.2 `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_QUASI_DEFAULT Specific n-dimensional ordering for quasirandom results.

2.1.2.3 enum curandRngType

CURAND generator types

Enumerator:

CURAND_RNG_PSEUDO_DEFAULT Default pseudorandom generator.
CURAND_RNG_PSEUDO_XORWOW XORWOW pseudorandom generator.
CURAND_RNG_QUASI_DEFAULT Default quasirandom generator.
CURAND_RNG_QUASI_SOBOL32 Sobol32 quasirandom generator.

2.1.2.4 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_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.

2.1.3 Function Documentation

2.1.3.1 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_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_QUASI_SOBOL32 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

2.1.3.2 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_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_QUASI_SOBOL32 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

2.1.3.3 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

2.1.3.4 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

2.1.3.5 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
num - 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

2.1.3.6 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
num - 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

2.1.3.7 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

2.1.3.8 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

2.1.3.9 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

2.1.3.10 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)

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

2.1.3.11 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

2.1.3.12 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

2.1.3.13 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_BEST
- CURAND_ORDERING_PSEUDO_DEFAULT

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

2.1.3.14 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

2.1.3.15 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

2.1.3.16 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

2.2 Device API

Typedefs

- typedef struct curandStateXORWOW curandState_t
- typedef struct curandStateSobol32 curandStateSobol32_t
- typedef struct curandStateXORWOW curandStateXORWOW_t

Functions

- __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 (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, curandStateXORWOW_t *state)
  Initialize XORWOW state.
- __device__ float curand_normal (curandStateSobol32_t *state)
  Return a normally distributed float from an Sobol32 generator.
- __device__ float curand_normal (curandStateXORWOW_t *state)
  Return a normally distributed float from an XORWOW generator.
- __device__ float2 curand_normal2 (curandStateXORWOW_t *state)
  Return two normally distributed floats from an XORWOW generator.
- __device__ double2 curand_normal2_double (curandStateXORWOW_t *state)
  Return two normally distributed doubles from an XORWOW generator.
- __device__ double curand_normal_double (curandStateSobol32_t *state)
  Return a normally distributed double from an Sobol32 generator.
- __device__ double curand_normal_double (curandStateXORWOW_t *state)
  Return a normally distributed double from an XORWOW generator.
- __device__ float curand_uniform (curandStateSobol32_t *state)
  Return a uniformly distributed float from a Sobol32 generator.
- __device__ float curand_uniform (curandStateXORWOW_t *state)
  Return a uniformly distributed float from an XORWOW generator.
- __device__ double curand_uniform_double (curandStateSobol32_t *state)
  Return a uniformly distributed double from a Sobol32 generator.
- __device__ double curand_uniform_double (curandStateXORWOW_t *state)
  Return a uniformly distributed double from an XORWOW generator.

- __device__ void skipahead (unsigned int n, curandStateSobol32_t *state)
  Update Sobol32 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, curandStateXORWOW_t *state)
  Update XORWOW state to skip ahead `n` *subsequences.*

2.2.1 Typedef Documentation

2.2.1.1 typedef struct curandStateXORWOW curandState_t

Default RNG

2.2.1.2 typedef struct curandStateSobol32 curandStateSobol32_t

CURAND Sobol32 state

2.2.1.3 typedef struct curandStateXORWOW curandStateXORWOW_t

CURAND XORWOW state

2.2.2 Function Documentation

2.2.2.1 __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.

2.2.2.2 `__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.

2.2.2.3 `__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

2.2.2.4 __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

2.2.2.5 __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`

2.2.2.6 __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`

2.2.2.7 __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`

2.2.2.8 __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`

2.2.2.9 __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`

2.2.2.10 __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`

2.2.2.11 __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`

2.2.2.12 __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`

2.2.2.13 __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`

2.2.2.14 __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`

2.2.2.15 __device__ void skipahead (unsigned int *n*, curandStateSobol32_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

2.2.2.16 __device__ void skipahead (unsigned long long *n*, 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

2.2.2.17 `__device__ void skipahead_sequence (unsigned long long n, 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