hcRNG Documentation

Release

MulticoreWare

May 10, 2016

Contents

CHAPTER 1

Introduction

The hcRNG library is an implementation of uniform random number generators targetting the AMD heterogenous hardware via HCC compiler runtime. The computational resources of underlying AMD heterogenous compute gets exposed and exploited through the HCC C++ frontend. Refer here for more details on HCC compiler.

The following list enumerates the current set of RNG generators that are supported so far.

1. MRG31k3p [4]
2. MRG32k3a [7]
3. LFSR113 [8]
4. Philox-4x32-10 [11]

Library provides multiple streams that are created on the host computer and used to generate random numbers either on the host or on computing devices by work items. Such multiple streams are essential for parallel simulation [6] and are often useful as well for simulation on a single processing element (or within a single work item), for example when comparing similar systems via simulation with common random numbers [1], [9], [10], [5] . Streams can also be divided into segments of equal length called substreams, as in [2], [5], [10] .

1.1 1. Getting Started

1.1.1 1.1. Introduction

The hcRNG library is an implementation of uniform random number generators targetting the AMD heterogenous hardware via HCC compiler runtime. The computational resources of underlying AMD heterogenous compute gets exposed and exploited through the HCC C++ frontend. Refer here for more details on HCC compiler.

The following list enumerates the current set of RNG generators that are supported so far.

1. MRG31k3p [4]
2. MRG32k3a [7]
3. LFSR113 [8]
4. Philox-4x32-10 [11]

Library provides multiple streams that are created on the host computer and used to generate random numbers either on the host or on computing devices by work items. Such multiple streams are essential for parallel simulation [6] and are often useful as well for simulation on a single processing element (or within a single work item), for example when comparing similar systems via simulation with common random numbers [1], [9], [10], [5] . Streams can also be divided into segments of equal length called substreams, as in [2], [5], [10] .

1.1.2 1.2. Prerequisites

This section lists the known set of hardware and software requirements to build this library

1.2.1. Hardware

- CPU: mainstream brand, Better if with >=4 Cores Intel Haswell based CPU
- System Memory >= 4GB (Better if >10GB for NN application over multiple GPUs)
- Hard Drive > 200GB (Better if SSD or NVMe driver for NN application over multiple GPUs)
- Minimum GPU Memory (Global) > 2GB

1.2.2. GPU cards supported

- dGPU: AMD R9 Fury X, R9 Fury, R9 Nano
- APU: AMD Kaveri or Carrizo

1.2.3. AMD Driver and Runtime

- Radeon Open Compute Kernel (ROCK) driver : https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver
- HSA runtime API and runtime for Boltzmann: https://github.com/RadeonOpenCompute/ROCR-Runtime

1.2.4. System software

- Ubuntu 14.04 trusty
- GCC 4.6 and later
- CPP 4.6 and later (come with GCC package)
- python 2.7 and later
- HCC 0.9 from here

1.2.5. Tools and Misc

- git 1.9 and later
- cmake 2.6 and later (2.6 and 2.8 are tested)
- firewall off
- root privilege or user account in sudo group

1.2.6. Ubuntu Packages

- libc6-dev-i386
- liblapack-dev
- graphicsmagick

1.1.3 1.3. Tested Environments

This sections enumerates the list of tested combinations of Hardware and system softwares.

1.3.1. Driver versions

- Boltzmann Early Release Driver + dGPU
  - Radeon Open Compute Kernel (ROCK) driver : https://github.com/RadeonOpenCompute/ROCK-Kernel-Driver
  - HSA runtime API and runtime for Boltzmann: https://github.com/RadeonOpenCompute/ROCR-Runtime
- Traditional HSA driver + APU (Kaveri)

1.3.2. GPU Cards

- Radeon R9 Nano
- Radeon R9 FuryX
- Radeon R9 Fury
- Kaveri and Carizo APU

1.3.3. Desktop System

- Supermicro SYS-7048GR-TR Tower 4 W9100 GPU
- ASUS X99-E WS motherboard with 4 AMD FirePro W9100
- Gigabyte GA-X79S 2 AMD FirePro W9100 GPU's

1.3.4. Server System

- Supermicro SYS 2028GR-THT 6 R9 NANO
- Supermicro SYS-1028GQ-TRT 4 R9 NANO
- Supermicro SYS-7048GR-TR Tower 4 R9 NANO

1.1.4 1.4. Installation steps

The following are the steps to use the library

- ROCM 1.0 Kernel, Driver and Compiler Installation (if not done until now)
- Library installation.

1.4.1. ROCM 1.0 Installation

To Know more about ROCM refer https://github.com/RadeonOpenCompute/ROCm/blob/master/README.md

a. Installing Debian ROCM repositories

Before proceeding, make sure to completely uninstall any pre-release ROCm packages.

Refer https://github.com/RadeonOpenCompute/ROCm#removing-pre-release-packages for instructions to remove pre-release ROCM packages.

Steps to install rocm package are,

```
wget -qO - http://packages.amd.com/rocm/apt/debian/rocm.gpg.key |
sudo apt-key add -
```

```
sudo sh -c 'echo deb [arch=amd64] http://packages.amd.com/rocm/apt/debian/
trusty main > /etc/apt/sources.list.d/rocm.list'
```

```
sudo apt-get update
```

```
sudo apt-get install rocm
```

and Reboot the system

b. Verifying the Installation

Once Reboot, to verify that the ROCm stack completed successfully you can execute HSA vector_copy sample application:

- cd /opt/rocm/hsa/sample
- make
- ./vector_copy

1.4.2. Library Installation

1. Install using Prebuilt debian

```
wget https://bitbucket.org/multicoreware/hcrng/downloads/hcrng-master-c751168-Linux
```

```
sudo dpkg -i hcrng-master-c751168-Linux.deb
```

2. Build debian from source

```
git clone https://bitbucket.org/multicoreware/hcrng.git && cd
hcrng

chmod +x build.sh && ./build.sh
```

build.sh execution builds the library and generates a debian under build directory. Additionally to run the unit test along with installation invoke the following command,

```
./build.sh --test=on
```

To uninstall the library, invoke the following series of commands

chmod +x clean.sh

./clean.sh

1.1.5 1.5. Generators and prefixes

The API is, in large part, the same for every generator, with only the prefix of the type and function names that changes across generators. For example, to use the MRG31k3p generator, one needs to include the corresponding header file (which is normally the lowercase name of the generator with a .h extension on the host or the device) and use type and function names that start with hcrngMrg31k3p:

```
#include <hcRNG/mrg31k3p.h>
double foo(hcrngMrg31k3pStream* stream) {
return hcrngMrg31k3pRandomU01(stream);
}
```

The above function just returns a number uniformly distributed in (0,1) generated using the stream passed as its argument. To use the LFSR113 generator instead of MRG31k3p, one must change the include directive and use type and function names that start with hcrngLfsr113:

```
#include <hcRNG/lfsr113.h>
double foo(hcrngLfsr113Stream* stream) {
return hcrngLfsr113RandomU01(stream);
}
```

In the generator API reference, the generator-specific part of the prefix is not shown. The hcRNG.h header file declares common function across different generators and also utility library functions.

1.5.1. Small examples

In the examples given below, we use the MRG31k3p from [4]. In general, a stream object contains three states: the initial state of the stream (or seed), the initial state of the current substream (by default it is equal to the seed), and the current state. With MRG31k3p, each state is comprised of six 31-bit integers. Each time a random number is generated, the current state advances by one position. There are also functions to reset the state to the initial one, or to the beginning of the current substream, or to the start of the next substream. Streams can be created and manipulated in arrays of arbitrary sizes. For a single stream, one uses an array of size 1. One can separately declare and allocate memory for an array of streams, create (initialize) the streams, clone them, copy them to preallocated space, etc.

1.5.2. Using streams on the host

We start with a small example in which we just create a few streams, then use them to generate numbers on the host computer and compute some quantity. This could be done as well by using only a single stream, but we use more just for the purpose of illustration.

The code includes the header for the MRG31k3p RNG.

```
#include <hcRNG/mrg31k3p.h>
```

We create an array of two streams named streams and a single stream named single.

```
hcrngMrg31k3pStream* streams = hcrngMrg31k3pCreateStreams(NULL, 2, NULL, NULL);
hcrngMrg31k3pStream* single = hcrngMrg31k3pCreateStreams(NULL, 1, NULL, NULL);
```

Then we repeat the following 100 times: we generate a uniform random number in (0,1) and an integer in {1,...,6}, and compute the indicator that the product is less than 2.

```
int count = 0;
for (int i = 0; i < 100; i++) {
  double u = hcrngMrg31k3pRandomU01(&streams[i % 2]);
  int x = hcrngMrg31k3pRandomInteger(single, 1, 6);
  if (x * u < 2) count++;
}
```

The uniform random numbers over (0,1) are generated by alternating the two streams from the array. We then print the average of those indicators.

```
printf("Average of indicators = %f\n", (double)count / 100.0);
```

1.5.3. Using streams in work items

In our second example, we create an array of streams and use them in work items that execute in parallel on a GPU device, one distinct stream per work item. Note that it is also possible (and sometimes useful) to use more than one stream per work item. We show only fragments of the code, to illustrate what we do with the streams. This code is only for illustration; the program does no useful computation.

In the code, we first include the hcRNG header for the MRG31k3p RNG:

```
#include <hcRNG/mrg31k3p.h>
```

Now suppose we have an integer variable numWorkItems that indicates the number of work items we want to use. We create an array of numWorkItems streams (and allocate memory for both the array and the stream objects). The creator returns in the variable streamBufferSize the size of the buffer that this array occupies (it depends on how much space is required to store the stream states), and an error code.

```
size_t streamBufferSize;
hcrngMrg31k3pStream* streams = hcrngMrg31k3pCreateStreams(NULL, numWorkItems, &streamBufferSize, (hcr
```

Then we create an HCC buffer of size streamBufferSize and fill it with a copy of the array of streams, to pass to the device. We also create and pass a buffer that will be used by the device to return an array of numWorkItems values of type float.

Create buffer to transfer streams to the device. acc[1] denotes that a pointer is created in device (1st GPU). To create a pointer in device, it is mandatory to enumerate a list of accelerators and select the desired accelerator.

```
std::vector<hc::accelerator>acc = hc::accelerator::get_all();
accelerator_view accl_view = (acc[1].create_view());
hcrngMrg31k3pStream* buf_in = hc::am_alloc(sizeof(hcrngMrg31k3pStream) * numWorkItems, acc[1], 0);
hc::am_copy(streams_buffer, streams, numWorkItems * sizeof(hcrngMrg31k3pStream));
```

Create buffer to transfer output back from the device.

```
float* buf_out = hc::am_alloc(sizeof(float) * numberCount, acc[1], 0);
```

Call the kernel function. Kernels of type float and double has suffixes "_single" and "_double" respectively.

```
hcrngMrg31k3pDeviceRandomU01Array_single(accl_view, numWorkItems, buf_in, numberCount, buf_out);
```

The host can then recover the array of size numWorkItems that contains these outputs(numberCount). RNG-Specific API's

```
hc::am_copy(RandomOutput, buf_out, numberCount * sizeof(float));
```

hcRNG_template describes the random streams API as it is intended to be implemented using different types of RNG's or even using quasi-Monte Carlo (QMC) point sets.

In the description of this API, every data type and function name is assigned the prefix hcrng. It is understood that, in the implementation for each RNG type, the prefix hcrng is to be expanded with another prefix that indicates the type of RNG (or other method) used.

As this API is not polymorphic, replacing an RNG type with another one in client code requires changing the code to match hcRNG function names and data types to match those of the replacement RNG. We also intend to propose a generic (in the polymorphic sense) interface to the hcRNG library.

1.5.4. Stream Objects and Stream States

The library defines, among others, two closely related types of structures: stream objects (hcrngStream) and stream states (hcrngStreamState). The definitions of both structures depend on the specific type of RNG that they pertain to. Stream states correspond to the seeds of conventional RNG's, to counter values in counter-based RNG's, or to point and coordinate indices in QMC methods. Normally, the client should not deal with stream states directly, but use instead the higher-level stream objects. Stream objects are intended to store several stream states: the current and initial stream states, but also current substream state when support for substreams is available. Stream objects may also store other properties of the RNG, such as encryption keys for cryptography-based RNG's.

1.5.5. Arrays of Stream Objects

Many functions are defined only for arrays of stream objects and not for single stream objects. It is always possible to use these functions for single stream objects by specifying a unit array size.

1.5.6. Defining Preprocessors

When a kernel is called, the stream states it needs are normally passed by the host and stored in global memory. If default settings are not suitable for the user's needs, optional library behavior can be selected by defining specific preprocessor macros before including the hcRNG header. For example, to enable single precision on the device while using the MRG31k3p generator, use:

```
#define HCRNG_SINGLE_PRECISION
```

1.2 2. hcRNG API Reference

This section provides a brief description of APIs and helper routines hosted by the library.

1.2.1 2.1. hcRNG_template File Reference

Template of the specialized interface for specific generators More...

#include <hcRNG.h>
#include <stdio.h>

2.1.1. Enumerations

- enum **hcrngStatus_**

  typedef enum **hcrngStatus_** hcrngStatus

  Error codes. More...

2.1.2. Data Structures

- struct **hcrngStreamState**

  Stream state [host/device]. More...

- struct **hcrngStream**

  Stream object [host/device]. More...

- struct **hcrngStreamCreator**

  Stream creator object. More...

2.1.3. Functions

2.1.3.1. Helper functions

- const char* **hcrngGetErrorString** ()

  Retrieve the last error message. More...

- const char* **hcrngGetLibraryRoot** ()

  Retrieve the library installation path. More...

2.1.3.2. Stream Creators

Functions to create, destroy and modify stream creator objects (factory pattern).

- hcrngStreamCreator* **hcrngCopyStreamCreator** (const hcrngStreamCreator* creator, hcrngStatus* err)

  Duplicate an existing stream creator object. More...

- hcrngStatus **hcrngDestroyStreamCreator** (hcrngStreamCreator* creator)

  Destroy a stream creator object. More...

- hcrngStatus **hcrngRewindStreamCreator** (hcrngStreamCreator* creator)

  Reset a stream creator to its original initial state. More...

- hcrngStatus **hcrngSetBaseCreatorState** (hcrngStreamCreator* creator, const hcrngStreamState* baseState)

  Change the base stream state of a stream creator. More...

- hcrngStatus **hcrngChangeStreamsSpacing** (hcrngStreamCreator* creator, int e, int c)

  Change the spacing between successive streams. More...

2.1.3.3. Stream Allocation, Destruction and Initialization

Functions to create or destroy random streams and arrays of random streams.

- hcrngStream* **hcrngAllocStreams** (size_t count, size_t* bufSize, hcrngStatus* err)

  Reserve memory for one or more stream objects. More...

- hcrngStatus **hcrngDestroyStreams** (hcrngStream* streams)

  Destroy one or many stream objects. More...

- hcrngStream* **hcrngCreateStreams** (hcrngStreamCreator* creator, size_t count, size_t* bufSize, hcrngStatus* err)

  Allocate memory for and create new RNG stream objects. More...

- hcrngStatus **hcrngCreateOverStreams** (hcrngStreamCreator* creator, size_t count, hcrngStream* streams)

  Create new RNG stream objects in already allocated memory. More...

- hcrngStream* **hcrngCopyStreams** (size_t count, const hcrngStream* streams, hcrngStatus* err)

  Clone RNG stream objects. More...

- hcrngStatus **hcrngCopyOverStreams** (size_t count, hcrngStream* destStreams, const hcrngStream* srcStreams)

  Copy RNG stream objects in already allocated memory [device]. More...

2.1.3.4. Stream Output

Functions to read successive values from a random stream.

- double **hcrngRandomU01** (hcrngStream* stream)

  Generate the next random value in (0,1) [device]. More...

- int **hcrngRandomInteger** (hcrngStream* stream, int i, int j)

  Generate the next random integer value [device]. More...

- hcrngStatus **hcrngRandomU01Array** (hcrngStream* stream, size_t count, double* buffer)

  Fill an array with successive random values in (0,1) [device]. More...

- hcrngStatus **hcrngRandomIntegerArray** (hcrngStream* stream, int i, int j, size_t count, int* buffer)

  Fill an array with successive random integer values [device]. More...

2.1.3.5. Stream Navigation

Functions to roll back or advance streams by many steps.

- hcrngStatus **hcrngRewindStreams** (size_t count, hcrngStream* streams)

  Reinitialize streams to their initial states [device]. More...

- hcrngStatus **hcrngRewindSubstreams** (size_t count, hcrngStream* streams)

  Reinitialize streams to their initial substream states [device]. More...

- hcrngStatus **hcrngForwardToNextSubstreams** (size_t count, hcrngStream* streams)

  Advance streams to the next substreams [device]. More...

- hcrngStream* **hcrngMakeSubstreams** (hcrngStream* stream, size_t count, size_t* bufSize, hcrngStatus* err)

  Allocate and make an array of substreams of a stream. More...

- hcrngStatus **hcrngMakeOverSubstreams** (hcrngStream* stream, size_t count, hcrngStream* substreams)

  Make an array of substreams of a stream. More...

- hcrngStatus **hcrngAdvanceStreams** (size_t count, hcrngStream* streams, int e, int c)

  Advance the state of streams by many steps. More...

2.1.3.6. Work Functions

Kernel functions to generate Random numbers.

- hcrngStatus **hcrngDeviceRandomU01Array_single** (hc::accelerator_view &accl_view, size_t streamCount, hcrngStream* streams, size_t numberCount, float* outBuffer, int streamlength = 0, size_t streams_per_thread = 1)

- hcrngStatus **hcrngDeviceRandomU01Array_double** (hc::accelerator_view &accl_view, size_t streamCount, hcrngStream* streams, size_t numberCount, double* outBuffer, int streamlength = 0, size_t streams_per_thread = 1)

The last two arguments are default arguments and can be used in case of multistream usage. More...

2.1.3.7. Miscellaneous Functions

- hcrngStatus **hcrngWriteStreamInfo** (const hcrngStream* stream, FILE* file)

  Format and output information about a stream object to a file. More...

2.1.4. Detailed Description

Template of the specialized interface for specific generators.

The function and type names in this API all start with hcrng. In each specific implementation, this prefix is expanded to a specific prefix; e.g., hcrngMrg31k3p for the MRG31k3p generator.

In the standard case, streams and substreams are defined as in [10], [2], [5] . The sequence of successive states of the base RNG over its entire period of length ρ is divided into streams whose starting points are Z steps apart. The sequence for each stream (of length Z) is further divided into substreams of length W. The integers Z and W have default values that have been carefully selected to avoid detectable dependence between successive streams and substreams, and are large enough to make sure that streams and substreams will not be exhausted in practice. It is strongly recommended to never change these values (even if the software allows it). The initial state of the first stream (the seed of the library) has a default value. It can be changed by invoking hcrngSetBaseCreatorState() before creating a first stream.

A stream object is a structure that contains the current state of the stream, its initial state (at the beginning of the stream), and the initial state of the current substream. Whenever the user creates a new stream, the software automatically jumps ahead by Z steps to find its initial state, and the three states in the stream object are set to it. The form of the state depends on the type of RNG.

Some functions are available on both the host and the devices (they can be used within a kernel) whereas others (such as stream creation) are available only on the host. Many functions are defined only for arrays of streams; for a single stream, it suffices to specify an array of size 1.When a kernel is called, one should pass a copy of the streams from the host to the global memory of the device. Another copy of the stream state uses it in the kernel code to generate random numbers.

To use the hcRNG library from within a user-defined kernel, the user must include the hcRNG header file corresponding to the desired RNG via an include directive. Other specific preprocessor macros can be placed before including the header file to change settings of the library when the default values are not suitable for the user. The following options are currently available:

HCRNG_SINGLE_PRECISION : With this option, all the random numbers returned by hcrngRandomU01() and hcrngRandomU01Array(), and generated by hcrngDeviceRandomU01Array(), will be of type float instead of double (the default setting). This option can be activated and affects all implemented RNGs.

To generate single-precision floating point numbers also on the host, still using the MRG31k3p generator, the host code should contain:

```
#define HCRNG_SINGLE_PRECISION
#include <mrg31k3p.h>
```

The functions described here are all available on the host, in all implementations, unless specified otherwise. Only some of the functions and types are also available on the device in addition to the host; they are tagged with [device]. Other functions are only available on the device; they are tagged with [device-only]. Some functions return an error code in err. Implemented RNG's

The following table lists the RNG's that are currently implemented in hcRNG with the name of the corresponding header file.

RNG	Prefix	Host/Device Header File
MRG31k3p	Mrg31k3p	mrg31k3p.h
MRG32k3a	Mrg32k3a	mrg32k3a.h
LFSR113	Lfsr113	lfsr113.h
Philox-4×32-10	Philox432	philox432.h

2.1.4.1. The MRG31k3p Generator

The MRG31k3p generator is defined in [4] . In its specific implementation, the function and type names start with hcrngMrg31k3p. For this RNG, a state is a vector of six 31-bit integers, represented internally as unsigned int. The entire period length of approximately 2^185 is divided into approximately 2^51 non-overlapping streams of length Z=2^134. Each stream is further partitioned into substreams of length W=2^72. The state (and seed) of each stream is a vector of six 31-bit integers. This size of state is appropriate for having streams running in work items on GPU cards, for example, while providing a sufficient period length for most applications.

2.1.4.2. The MRG32k3a Generator

MRG32k3a is a combined multiple recursive generator (MRG) proposed by L'Ecuyer [7], implemented here in 64-bit integer arithmetic. This RNG has a period length of approximately 2^191, and is divided into approximately 2^64 non-overlapping streams of length Z=2^127, and each stream is subdivided in 2^51 substreams of length W=2^76. These are the same numbers as in [5] . The state of a stream at any given step is a six-dimensional vector of 32-bit integers, but those integers are stored as unsigned long (64-bit integers) in the present implementation (so they use twice the space). The generator has 32 bits of resolution. Note that in the original version proposed in [7] and [5], the recurrences are implemented in double instead, and the state is stored in six 32-bit integers. The change in implementation is to avoid using double's, which are not available on many GPU devices, and also because the 64-bit implementation is much faster than that in double when 64-bit integer arithmetic is available on the hardware.

2.1.4.3. The LFSR113 Generator

The LFSR113 generator is defined in [8]. In its implementation, the function and type names start with hcrngLfsr113. For this RNG, a state vector of four 31-bit integers, represented internally as unsigned int. The period length of approximately 2^113 is divided into approximately 2^23 non-overlapping streams of length Z=2^90. Each stream is further partitioned into 2^35 substreams of length W=2^55. Note that the functions hcrngLfsr113ChangeStreamsSpacing() and hcrngLfsr113AdvancedStreams() are not implemented in the current version.

2.1.4.4. The Philox-4×32-10 Generator

The counter-based Philox-4×32-10 generator is defined in [11]. Unlike the previous three generators, its design is not supported by a theoretical analysis of equidistribution. It has only been subjected to empirical testing with the TestU01 software [3] (the other three generators also have). In its implementation, the function and type names start with hcrngPhilox432. For this RNG, a state is a 128-bit counter with a 64-bit key, and a 2-bit index used to iterate over the four 32-bit outputs generated for each counter value. The counter is represented internally as a vector of four 32-bit unsigned int values and the index, as a single unsigned int value. In the current hcRNG version, the key is the same for all streams, so it is not stored in each stream object but rather hardcoded in the implementation. The period length of 2^130 is divided into 2^28 non-overlapping streams of length Z=2^102. Each stream is further partitioned into 2^36 substreams of length W=2^66. The key (all bits to 0), initial counter and order in which the four outputs per counter value are returned are chosen to generate the same values, in the same order, as Random123's Engine module [11], designed for use with the standard C++11 random library. Note that the function hcrngPhilox432ChangeStreamsSpacing() supports only values of c that are multiples of 4, with either e=0 or e 2.

2.1.5. Function Documentation

2.1.5.1. hcrngCopyStreamCreator()

```
hcrngStreamCreator* hcrngCopyStreamCreator ( const hcrngStreamCreator *        creator,
                                               hcrngStatus *     err
                                             )
```

Duplicate an existing stream creator object.

Create an identical copy (a clone) of the stream creator creator. To create a copy of the default creator, put NULL as the creator parameter. All the new stream creators returned by hcrngCopyStreamCreator(NULL, NULL) will create the same sequence of random streams, unless the default stream creator is used to create streams between successive calls to this function.

In/out	Parameters	Description
[in]	creator	Stream creator object to be copied, or NULL to copy the default stream creator.
[out]	err	Error status variable, or NULL.

Returns, The newly created stream creator object.

2.1.5.2. hcrngDestroyStreamCreator()

```
hcrngStatus hcrngDestroyStreamCreator ( hcrngStreamCreator *   creator )
```

Destroy a stream creator object. Release the resources associated to a stream creator object.

In/out	Parameters	Description
[out]	creator	Stream creator object to be destroyed.

Returns, Error status

2.1.5.3. hcrngRewindStreamCreator()

```
hcrngStatus hcrngRewindStreamCreator ( hcrngStreamCreator *    creator )
```

Reset a stream creator to its original initial state, so it can re-create the same streams over again.

In/out	Parameters	Description
[in]	creator	Stream creator object to be reset.

Returns, Error status

2.1.5.4. hcrngSetBaseCreatorState()

```
hcrngStatus hcrngSetBaseCreatorState ( hcrngStreamCreator *    creator,
                                       const hcrngStreamState *        baseState
                                     )
```

Change the base stream state of a stream creator.

Set the base state of the stream creator, which can be seen as the seed of the underlying RNG. This will be the initial state (or seed) of the first stream created by this creator. Then, for most conventional RNGs, the initial states of successive streams will be spaced equally, by Z steps in the RNG sequence. The type and size of the baseState parameter depends on the type of RNG. The base state always has a default value, so this function does not need to be invoked.

In/out	Parameters	Description
[in,out]	creator	Stream creator object.
[in]	baseState	New initial base stream state. Can be set to NULL to use the library default.

Returns, Error status

> **Warning:** It is recommended to use the library default base state.

2.1.5.5. hcrngChangeStreamsSpacing()

```
hcrngStatus hcrngChangeStreamsSpacing ( hcrngStreamCreator *   creator,
                                        int   e,
                                        int   c
                                      )
```

Change the spacing between successive streams.

This function should be used only in exceptional circumstances. It changes the spacing Z between the initial states of the successive streams from the default value to Z=2e+c if e>0, or to Z=c if e=0. One must have e0 but c can take negative values. The default spacing values have been carefully selected for each RNG to avoid overlap and dependence between streams, and it is highly recommended not to change them.

In/out	Parameters	Description
[in,out]	creator	Stream creator object.
[in]	e	Value of e.
[in]	c	Value of c.

Returns, Error status

> **Warning:** It is recommended to use the library default spacing and not to invoke this function.

2.1.5.6. hcrngAllocStreams()

```
hcrngStream* hcrngAllocStreams ( size_t        count,
                                 size_t *      bufSize,
                                 hcrngStatus *         err
                               )
```

Reserve memory space for count stream objects, without creating the stream objects. Returns a pointer to the allocated buffer and returns in bufSize the size of the allocated buffer, in bytes.

In/out	Parameters	Description
[in]	count	Number of stream objects to allocate.
[out]	bufSize	Size in bytes of the allocated buffer, or NULL if not needed.
[out]	err	Error status variable, or NULL.

Returns, Pointer to the newly allocated buffer.

2.1.5.7. hcrngDestroyStreams()

```
hcrngStatus hcrngDestroyStreams ( hcrngStream*  streams )
```

Destroy one or many stream objects. Release the memory space taken by those stream objects.

In/out	Parameters	Description
[in,out]	streams	Stream object buffer to be released.

Returns, Error status

Examples: Multistream.cpp, and RandomArray.cpp.

2.1.5.8. hcrngCreateStreams()

```
hcrngStream* hcrngCreateStreams ( hcrngStreamCreator *         creator,
                                  size_t       count,
                                  size_t *     bufSize,
                                  hcrngStatus *        err
                                )
```

Allocate memory for and create new RNG stream objects.

Create and return an array of count new streams using the specified creator. This function also reserves the memory space required for the structures and initializes the stream states. It returns in bufSize the size of the allocated buffer, in bytes. To use the default creator, put NULL as the creator parameter. To create a single stream, just put set count to 1.

In/out	Parameters	Description
[in,out]	creator	Stream creator object, or NULL to use the default stream creator.
[in]	count	Size of the array (use 1 for a single stream object).
[out]	bufSize	Size in bytes of the allocated buffer, or NULL if not needed.
[out]	err	Error status variable, or NULL.

Returns, The newly created array of stream object.

Examples: Multistream.cpp, and RandomArray.cpp.

2.1.5.9. hcrngCreateOverStreams()

```
hcrngStatus hcrngCreateOverStreams ( hcrngStreamCreator *      creator,
                                     size_t    count,
                                     hcrngStream *     streams
                                   )
```

Create new RNG stream objects in already allocated memory.

This function is similar to hcrngCreateStreams(), except that it does not reserve memory for the structure. It creates the array of new streams in the preallocated streams buffer, which could have been reserved earlier via either hcrngAllocStreams() or hcrngCreateStreams(). It permits the client to reuse memory that was previously allocated for other streams.

In/out	Parameters	Description
[in,out]	creator	Stream creator object, or NULL to use the default stream creator.
[in]	count	Size of the array (use 1 for a single stream object).
[out]	streams	Buffer in which the new stream(s) will be stored.

Returns, Error status

2.1.5.10. hcrngCopyStreams()

```
hcrngStream* hcrngCopyStreams ( size_t         count,
                                const hcrngStream *    streams,
                                hcrngStatus *          err
                              )
```

Clone RNG stream objects. Create an identical copy (a clone) of each of the count stream objects in the array streams. This function allocates memory for all the new structures before cloning, and returns a pointer to the new structure.

In/out	Parameters	Description
[in]	count	Number of random number in the array (use 1 for a single stream object).
[in]	streams	Stream object or array of stream objects to be cloned.
[out]	err	Error status variable, or NULL.

Returns, The newly created stream object or array of stream objects.

2.1.5.11. hcrngCopyOverStreams()

```
hcrngStatus hcrngCopyOverStreams ( size_t      count,
                                   hcrngStream *       destStreams,
                                   const hcrngStream *         srcStreams
                                 )
```

Copy RNG stream objects in already allocated memory [device]. Copy (or restore) the stream objects srcStreams into the buffer destStreams, and each of the count stream objects from the array srcStreams into the buffer destStreams. This function does not allocate memory for the structures in destStreams; it assumes that this has already been done.

In/out	Parameters	Description
[in]	count	Number of stream objects to copy (use 1 for a single stream object).
[out]	destStreams	Destination buffer into which to copy (its content will be overwritten).
[in]	srcStreams	Stream object or array of stream objects to be copied.

Returns, Error status

2.1.5.12. hcrngRandomU01()

```
double hcrngRandomU01 ( hcrngStream *  stream  )
```

Generate the next random value in (0,1) [device]. Generate and return a (pseudo)random number from the uniform distribution over the interval (0,1), using stream. If this stream is from an RNG, the stream state is advanced by one step before producing the (pseudo)random number. By default, the returned value is of type double. But if the option HCRNG_SINGLE_PRECISION is defined, the returned value will be of type float. Setting this option changes the type of the returned value for all RNGs and all functions that use hcrngRandomU01().

In/out	Parameters	Description
[in,out]	stream	Stream used to generate the random value.

Returns, A random floating-point value uniformly distributed in (0,1)

Examples: Multistream.cpp, and RandomArray.cpp.

2.1.5.13. hcrngRandomInteger()

```
int hcrngRandomInteger ( hcrngStream *          stream,
                         int    i,
                         int    j
                       )
```

Generate the next random integer value [device]. Generate and return a (pseudo)random integer from the discrete uniform distribution over the integers {i,. . . ,j}, using stream, by calling hcrngRandomU01() once and transforming the output by inversion. That is, it returns i + (int)((j-i+1) * hcrngRandomU01(stream)).

In/out	Parameters	Description
[in,out]	stream	Stream used to generate the random value.
[in]	i	Smallest integer value (inhcusive).
[in]	j	Largest integer value (inhcusive).

Returns, A random integer value uniformly distributed in {i,. . . ,j}.

2.1.5.14. hcrngRandomU01Array()

```
hcrngStatus hcrngRandomU01Array ( hcrngStream *        stream,
                                  size_t       count,
                                  double *     buffer
                                )
```

Fill an array with successive random values in (0,1) [device].Fill preallocated buffer with count successive (pseudo)random numbers. Equivalent to calling hcrngRandomU01() count times to fill the buffer. If HCRNG_SINGLE_PRECISION is defined, the buffer argument is of type float and will be filled by count values of type float instead.

In/out	Parameters	Description
[in,out]	stream	Stream used to generate the random values.
[in]	count	Number of values in the array.
[out]	buffer	Destination buffer (must be pre-allocated).

Returns, Error status

2.1.5.15. hcrngRandomIntegerArray()

```
hcrngStatus hcrngRandomIntegerArray ( hcrngStream *    stream,
                                      int      i,
                                      int      j,
                                      size_t   count,
                                      int *    buffer
                                    )
```

Fill an array with successive random integer values [device].Same as hcrngRandomU01Array(), but for integer values in {i,. . . ,j}. Equivalent to calling hcrngRandomInteger() count times to fill the buffer.

In/out	Parameters	Description
[in,out]	stream	Stream used to generate the random values.
[in]	i	Smallest integer value (inhcusive).
[in]	j	Largest integer value (inhcusive).
[in]	count	Number of values in the array.
[out]	buffer	Destination buffer (must be pre-allocated).

Returns, Error status

2.1.5.16. hcrngRewindStreams()

```
hcrngStatus hcrngRewindStreams ( size_t        count,
                                 hcrngStream *         streams
                               )
```

Reinitialize streams to their initial states [device]. Reinitialize all the streams in streams to their initial states. The current substream also becomes the initial one.

In/out	Parameters	Description
[in]	count	Number of stream objects in the array (use 1 for a single stream object).
[in,out]	streams	Stream object or array of stream objects to be reset to the start of the stream(s).

Returns, Error status

> **Warning:** This function can be slow on the device, because it reads the initial state from global memory.

2.1.5.17. hcrngRewindSubstreams()

```
hcrngStatus hcrngRewindSubstreams ( size_t     count,
                                    hcrngStream *      streams
                                  )
```

Reinitialize streams to their initial substream states [device]. Reinitialize all the streams in streams to the initial states of their current substream.

In/out	Parameters	Description
[in]	count	Number of stream objects in the array (use 1 for a single stream object).
[in,out]	streams	Stream object or array of stream objects to be reset to the beginning of the current substream(s).

Returns, Error status

Examples: Multistream.cpp

2.1.5.18. hcrngForwardToNextSubstreams()

```
hcrngStatus hcrngForwardToNextSubstreams ( size_t      count,
                                           hcrngStream *       streams
                                         )
```

Advance streams to the next substreams [device]. Reinitialize all the streams in streams to the initial states of their next substream. The current states and the initial states of the current substreams are changed.

In/out	Parameters	Description
[in]	count	Number of stream objects in the array (use 1 for a single stream object).
[in,out]	streams	Stream object or array of stream objects to be advanced to the next substream(s)

Returns, Error status

Examples: Multistream.cpp

2.1.5.19. hcrngMakeSubstreams()

```
hcrngStream* hcrngMakeSubstreams ( hcrngStream *       stream,
                                   size_t      count,
                                   size_t *    bufSize,
                                   hcrngStatus *       err
                                 )
```

Allocate and make an array of substreams of a stream.

Make and return an array of count copies of stream, whose current (and initial substream) states are the initial states of count successive substreams of stream. The first substream in the returned array is simply a copy of stream. This function also reserves the memory space required for the structures and initializes the stream states. It returns in bufSize the size of the allocated buffer, in bytes. To create a single stream, just set count to 1. When this function is invoked, the substream state and initial state of stream are advanced by count substreams.

2.1.5.20. hcrngMakeOverSubstreams()

```
hcrngStatus hcrngMakeOverSubstreams ( hcrngStream *    stream,
                                      size_t   count,
                                      hcrngStream *    substreams
                                    )
```

Make an array of substreams of a stream.

This function is similar to hcrngMakeStreams(), except that it does not reserve memory for the structure. It creates the array of new streams in the preallocated substreams buffer, which could have been reserved earlier via either hcrngAllocStreams(), hcrngMakeSubstreams() or hcrngCreateStreams(). It permits the client to reuse memory that was previously allocated for other streams.

2.1.5.21. hcrngAdvanceStreams()

```
hcrngStatus hcrngAdvanceStreams ( size_t       count,
                                  hcrngStream *        streams,
                                  int          e,
                                  int          c
                                )
```

Advance the state of streams by many steps.

This function should be used only in very exceptional circumstances. It advances the state of the streams in array streams by k steps, without modifying the states of other streams, nor the initial stream and substream states for those streams. If e>0, then k=2e+c; if e<0, then k=2|e|+c; and if e=0, then k=c. Note that c can take negative values. We discourage the use of this procedure to customize the length of streams and substreams. It is better to use the default spacing, which has been carefully selected for each RNG type.

In/out	Parameters	Description
[in]	count	Number of stream objects in the array (use 1 for a single stream object).
[in,out]	streams	Stream object or array of stream objects to be advanced.
[in]	e	Value of e.
[in]	c	Value of c.

Returns, Error status

> **Warning:** Check the implementation for all cases e>0, e=0 and e<0.

2.1.5.22. hcrngDeviceRandomU01Array()

```
hcrngStatus hcrngDeviceRandomU01Array_single ( hc::accelerator_view &accl_view,
                                               size_t              streamCount,
                                               hcrngStream*         streams,
                                               size_t               numberCount,
                                               float*               outBuffer,
                                               int                  streamlength = 0,
                                               size_t               streams_per_thread = 1 )
hcrngStatus hcrngDeviceRandomU01Array_double ( hc::accelerator_view &accl_view,
                                               size_t              streamCount,
                                               hcrngStream*         streams,
                                               size_t               numberCount,
                                               double*              outBuffer,
                                               int                  streamlength = 0,
                                               size_t               streams_per_thread = 1 )
```

Fill a buffer of random numbers.

Fill the buffer pointed to by outBuffer with numberCount uniform random numbers of type double (or of type float if HCRNG_SINGLE_PRECISION is defined), using streamCount work items. In the current implementation, numberCount must be a multiple of streamCount. It is adviced to call the kernel depending on the type of output buffer. Kernels of type float and double has suffixes "_single" and "_double" respectively.

In/out	Parameters	Description
[in]	accl_view	Using accelerator and accelerator_view Objects
[in]	streamCount	Number of streams in stream_array.
[in]	streams	HCC device pointer that contains an array of stream objects.
[in]	numberCount	Number of random number to store in the device pointer.
[out]	outBuffer	HCC device pointer in which the generated numbers will be stored.
[in]	stream_length	[Default argument] The length of the subtsream. stream_length = 0 (do not use substreams) stream_length = > 0 (go to next substreams after stream_length values) stream_length = < 0 (restart substream after stream_length values)
[in]	streams_per_thread	[Default argument] Number of streams a thread should handle. Must be a multiple of streamCount.

Returns, Error status

Examples: Multistream.cpp, and RandomArray.cpp.

Warning: In the current implementation, numberCount must be a multiple of streamCount and streams_per_thread must be a multiple of streamCount. The array streams is left unchanged, as there is no write-back from the device code. stream_length and streams_per_thread are default arguments and can be used for multistream random number generation.

2.1.5.23. hcrngWriteStreamInfo()

```
hcrngStatus hcrngWriteStreamInfo ( const hcrngStream *         stream,
                                   FILE *      file
                                 )
```

Format and output information about a stream object to a file.

In/out	Parameters	Description
[in]	stream	Stream object about which to write information.
[in]	file	File to which to output. Can be set to stdout or stderr for standard output and error.

Returns, Error status

1.2.2 2.2. hcRNG TYPES

2.2.1. Enumerations

2.2.1.1. HCRNG STATUS (hcrngStatus)

```
typedef enum hcrngStatus_ {
  HCRNG_SUCCESS                   = 0,
  HCRNG_OUT_OF_RESOURCES          = -1,
  HCRNG_INVALID_VALUE             = -2,
  HCRNG_INVALID_RNG_TYPE          = -3,
  HCRNG_INVALID_STREAM_CREATOR    = -4,
  HCRNG_INVALID_SEED              = -5,
  HCRNG_FUNCTION_NOT_IMPLEMENTED = -6
} hcrngStatus;
```

This enumeration is the set of hcRNG error codes.

Enumerator	
HCRNG_SUCCESS	the operation completed successfully.
HCRNG_OUT_OF_RESOURCES	resource allocation failed.
HCRNG_INVALID_VALUE	unsupported numerical value was passed to function.
HCRNG_INVALID_RNG_TYPE	unsupported rng type specified.
HCRNG_INVALID_STREAM_CREATOR	Stream creator is invalid.
HCRNG_INVALID_SEED	Seed value is greater than particular generators' predefined values.
HCRNG_FUNCTION_NOT_IMPLEMENTED	an internal hcRNG function not implemented.

2.2.2. Data Structures

2.2.2.1. hcrngStreamState

```
struct hcrngStreamState
```

Stream state [host/device]. Contains the state of a random stream. The definition of a state depends on the type of generator.

Examples: Multistream.cpp.

2.2.2.2. hcrngStream

```
struct hcrngStream
```

Stream object [host/device]. A structure that contains the current information on a stream object. It generally depends on the type of generator. It typically stores the current state, the initial state of the stream, and the initial state of the current substream.

Examples: Multistream.cpp, and RandomArray.cpp.

2.2.2.3. hcrngStreamCreator

```
struct hcrngStreamCreator
```

Stream creator object. For each type of RNG, there is a single default creator of streams, and this should be sufficient for most applications. Multiple creators could be useful for example to create the same successive stream objects multiple times in the same order, instead of storing them in an array and reusing them, or to create copies of the same streams in the same order at different locations in a distributed system, e.g., when simulating similar systems with common random numbers. Stream creators are created according to an abstract factory pattern.

2.2.3. Helper Functions

2.2.3.1. hcrngGetLibraryRoot()

```
const char* hcrngGetLibraryRoot ()
```

Retrieve the library installation path.

Returns, Value of the HCRNG_ROOT environment variable, if defined, else the current directory (.) of execution of the program.

2.2.3.2. hcrngGetErrorString()

```
const char* hcrngGetErrorString ()
```

Retrieve the last error message. The buffer containing the error message is internally allocated and must not be freed by the client.

Returns, Error message or NULL.

1.2.3 2.3. Examples

2.3.1. Host-only Code Example

```
//This example shows how to invoke Random generators from Host
//Example with Mrg31k3p Random number generator
#include <stdio.h>
#include <iostream>
#include <hcRNG/mrg31k3p.h>

int main() {

//Create Streams
  hcrngMrg31k3pStream* streams = hcrngMrg31k3pCreateStreams(NULL, 2, NULL, NULL);
  hcrngMrg31k3pStream* single = hcrngMrg31k3pCreateStreams(NULL, 1, NULL, NULL);

//Initialize the count
  int count = 0;
  for (int i = 0; i < 100; i++) {
//Calling RandomU01 function from host that generates random numbers on "streams"
      double u = hcrngMrg31k3pRandomU01(&streams[i % 2]);

//Calling RandomInteger funcion from host that generated random numbers on a single stream
      int    x = hcrngMrg31k3pRandomInteger(single, 1, 6);
      if (x * u < 2) count++;
  }
  std::cout << "Average of indicators = " << (double)count / 100.0 << std::endl;
  return 0;
}
```

2.3.2. Random-array generation Code example

```
//This example is a simple random array generation and it compares host output with device output
//Random number generator Mrg31k3p
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>
#include <hcRNG/mrg31k3p.h>
#include <hcRNG/hcRNG.h>
#include <hc.hpp>
#include <hc_am.hpp>

using namespace hc;

#define HCRNG_SINGLE_PRECISION
#ifdef HCRNG_SINGLE_PRECISION
typedef float fp_type;
#else
typedef double fp_type;
#endif

int main()
{
      hcrngStatus status = HCRNG_SUCCESS;
      bool ispassed = 1;
      size_t streamBufferSize;
      // Number oi streams
```

```
      size_t streamCount = 10;
      //Number of random numbers to be generated
      //numberCount must be a multiple of streamCount
      size_t numberCount = 100;

      //Enumerate the list of accelerators
      std::vector<hc::accelerator>acc = hc::accelerator::get_all();
      accelerator_view accl_view = (acc[1].create_view());

      //Allocate memory for host pointers
      fp_type *Random1 = (fp_type*) malloc(sizeof(fp_type) * numberCount);
      fp_type *Random2 = (fp_type*) malloc(sizeof(fp_type) * numberCount);
      fp_type *outBufferDevice = hc::am_alloc(sizeof(fp_type) * numberCount, acc[1], 0);

      //Create streams
      hcrngMrg31k3pStream *streams = hcrngMrg31k3pCreateStreams(NULL, streamCount, &streamBufferSize,
      hcrngMrg31k3pStream *streams_buffer = hc::am_alloc(sizeof(hcrngMrg31k3pStream) * streamCount, a
      hc::am_copy(streams_buffer, streams, streamCount* sizeof(hcrngMrg31k3pStream));

      //Invoke random number generators in device (here strean_length and streams_per_thread argumen
#ifdef HCRNG_SINGLE_PRECISION
      status = hcrngMrg31k3pDeviceRandomU01Array_single(accl_view, streamCount, streams_buffer, numbe
#else
      status = hcrngMrg31k3pDeviceRandomU01Array_double(accl_view, streamCount, streams_buffer, numbe
#endif
      if(status) std::cout << "TEST FAILED" << std::endl;
      hc::am_copy(Random1, outBufferDevice, numberCount * sizeof(fp_type));

      //Invoke random number generators in host
      for (size_t i = 0; i < numberCount; i++)
          Random2[i] = hcrngMrg31k3pRandomU01(&streams[i % streamCount]);

      // Compare host and device outputs
      for(int i =0; i < numberCount; i++) {
         if (Random1[i] != Random2[i]) {
              ispassed = 0;
              std::cout <<" RANDDEVICE[" << i<< "] " << Random1[i] << "and RANDHOST[" << i <<"] misma
              break;
          }
          else
              continue;
      }
      if(!ispassed) std::cout << "TEST FAILED" << std::endl;

      //Free host resources
      free(Random1);
      free(Random2);

      //Release device resources
      hc::am_free(outBufferDevice);
      hc::am_free(streams_buffer);

      return 0;
}
```

2.3.3. Multistream usage Code example

```
//Example on Multistream random number generation with Mrg31k3p generator
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <assert.h>
#include <hcRNG/mrg31k3p.h>
#include <hcRNG/hcRNG.h>
#include <hc.hpp>
#include <hc_am.hpp>

using namespace hc;

#define HCRNG_SINGLE_PRECISION
#ifdef HCRNG_SINGLE_PRECISION
typedef float fp_type;
#else
typedef double fp_type;
#endif

//Multistream generation with host
void multistream_fill_array(size_t spwi, size_t gsize, size_t quota, int substream_length, hcrngMrg31
{
  for (size_t i = 0; i < quota; i++) {
    for (size_t gid = 0; gid < gsize; gid++) {
      //Create streams
        hcrngMrg31k3pStream* s = &streams[spwi * gid];
        fp_type* out = &out_[spwi * (i * gsize + gid)];
        //Do nothing when subtsream_length is equal to 0
        if ((i > 0) && (substream_length > 0) && (i % substream_length == 0))
        //Forward to next substream when substream_length is greater than 0
            hcrngMrg31k3pForwardToNextSubstreams(spwi, s);
        else if ((i > 0) && (substream_length < 0) && (i % (-substream_length) == 0))
        //Rewind substreams when substream_length is smaller than 0
            hcrngMrg31k3pRewindSubstreams(spwi, s);
        //Generate Random Numbers
        for (size_t sid = 0; sid < spwi; sid++) {
            out[sid] = hcrngMrg31k3pRandomU01(&s[sid]);
        }
    }
  }
}

int main()
{
      hcrngStatus status = HCRNG_SUCCESS;
      bool ispassed = 1;
      size_t streamBufferSize;
      //Number of streams
      size_t streamCount = 10;
      //Number of Random numbers to be generated (numberCount should be a multiple of streamCount)
      size_t numberCount = 100;
      //Substream length
      //Substream_length       = 0   // do not use substreams
      //Substream_length       = > 0   // go to next substreams after Substream_length values
      //Substream_length       = < 0  // restart substream after Substream_length values
      int stream_length = 5;
      size_t streams_per_thread = 2;
```

```
      //Enumerate the list of accelerators
      std::vector<hc::accelerator>acc = hc::accelerator::get_all();
      accelerator_view accl_view = (acc[1].create_view());

      //Allocate Host pointers
      fp_type* Random1 = (fp_type*) malloc(sizeof(fp_type) * numberCount);
      fp_type* Random2 = (fp_type*) malloc(sizeof(fp_type) * numberCount);
      //Allocate buffer for Device output
      fp_type* outBufferDevice_substream = hc::am_alloc(sizeof(fp_type) * numberCount, acc[1], 0);
      hcrngMrg31k3pStream* streams = hcrngMrg31k3pCreateStreams(NULL, streamCount, &streamBufferSize,
      hcrngMrg31k3pStream* streams_buffer = hc::am_alloc(sizeof(hcrngMrg31k3pStream) * streamCount, a
      hc::am_copy(streams_buffer, streams, streamCount* sizeof(hcrngMrg31k3pStream));

      //Invoke Random number generator function in Device
#ifdef HCRNG_SINGLE_PRECISION
      status = hcrngMrg31k3pDeviceRandomU01Array_single(accl_view, streamCount, streams_buffer, numbe
#else
      status = hcrngMrg31k3pDeviceRandomU01Array_double(accl_view, streamCount, streams_buffer, numbe
#endif
      //Status check
      if(status) std::cout << "TEST FAILED" << std::endl;
      hc::am_copy(Random1, outBufferDevice_substream, numberCount * sizeof(fp_type));

      //Invoke random number generator from Host
      multistream_fill_array(streams_per_thread, streamCount/streams_per_thread, numberCount/streamCo

      //Compare Host and device outputs
      for(int i =0; i < numberCount; i++) {
         if (Random1[i] != Random2[i]) {
              ispassed = 0;
              std::cout <<" RANDDEVICE_SUBSTREAM[" << i<< "] " << Random1[i] << "and RANDHOST_SUBSTRE
              break;
          }
          else
              continue;
      }
      if(!ispassed) std::cout << "TEST FAILED" << std::endl;

      //Free host resources
      free(Random1);
      free(Random2);

      //Release device resources
      hc::am_free(outBufferDevice_substream);
      hc::am_free(streams_buffer);

      return 0;
}
```

1.2.4 2.4. Bibliography

[1] A. M. Law. Simulation Modeling and Analysis. McGraw-Hill, New York, fifth edition, 2014.

[2] P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. ACM Transactions on Mathematical Software, 17(1):98–111, 1991.

[3] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. ACM

Transactions on Mathematical Software, 33(4):Article 22, August 2007.

[4] P. L'Ecuyer and R. Touzin. Fast combined multiple recursive generators with multipliers of the form a = ± 2q ± 2r. In Proceedings of the 2000 Winter Simulation Conference, pages 683–689, Piscataway, NJ, 2000. IEEE Press.

[5] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. Operations Research, 50(6):1073–1075, 2002.

[6] P. L'Ecuyer, B. Oreshkin, and R. Simard. Random numbers for parallel computers: Requirements and methods, 2014. http://www.iro.umontreal.ca/ lecuyer/myftp/papers/parallel-rng-imacs.pdf.

[7] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. Operations Research, 47(1):159–164, 1999.

[8] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. Mathematics of Computation, 68(225):261–269, 1999.

[9] P. L'Ecuyer. Variance reduction's greatest hits. In Proceedings of the 2007 European Simulation and Modeling Conference, pages 5–12, Ghent, Belgium, 2007. EUROSIS.

[10] P. L'Ecuyer. SSJ: A Java Library for Stochastic Simulation, 2008. Software user's guide, available at http://www.iro.umontreal.ca/ lecuyer.

[11] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 16:1–16:12, New York, 2011. ACM.