clRNG: A Random Number API with Multiple Streams for OpenCL

Pierre L'Ecuyer, David Munger, Nabil Kemerchou

DIRO, Université de Montréal, Montréal (QC), Canada, H3C 3J7

April 6, 2015

Abstract

We present clRNG, a library for uniform random number generation in OpenCL. *Streams* of random numbers act as virtual random number generators. They can be created on the host computer in unlimited numbers, and then used either on the host or on other computing devices by work items to generate random numbers. Each stream also has equally-spaced substreams, which are occasionally useful. The API is currently implemented for four different RNGs, namely the MRG31k3p, MRG32k3a, LFSR113, and Philox-4×32-10 generators.

1 Introduction

We introduce `clRNG`, an OpenCL library for generating uniform random numbers. It 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 [7] 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 (CRNs) [1, 4, 5, 9]. Streams can also be divided into segments of equal length called substreams, as in [5, 6, 9]. Users for which streams are sufficient can simply ignore the existence of substreams.

In the examples given here, we use the MRG31k3p from [10], whose implementation is described briefly in Section 5. 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.

In what follows, we specify the API and illustrate its usage. We start in Section 2 with small examples that show how to create and use streams. In Section 3, we give examples, based on a simple inventory model, which show why and how multiple streams and substreams are useful. The API is detailed in Section 5.

2 Small examples

2.1 Using streams on the host

We start with a small artificial 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 illustration.

The code, shown in Figure 1, includes the header for the MRG31k3p RNG. In the `main` function, we create an array of two streams named `streams` and a single stream named `single`. Then we repeat the following 100 times: we generate a uniform random number in (0,1) and an integer in $\{1, \ldots, 6\}$, and compute the indicator that the product is less than 2. We then print the average of those indicators. The uniform random numbers over (0,1) are generated by alternating the two streams from the array.

```
#include <mrg31k3p.h>

int main() {
   clrngMrg31k3pStream* streams = clrngMrg31k3pCreateStreams(NULL, 2, NULL, NULL);
   clrngMrg31k3pStream* single = clrngMrg31k3pCreateStreams(NULL, 1, NULL, NULL);
   int count = 0;
   for (int i = 0; i < 100; i++) {
      double u = clrngMrg31k3pRandomU01(&streams[i % 2]);
      int    x = clrngMrg31k3pRandomInteger(single, 1, 6);
      if (x * u < 2) count++;
   }
   printf("Average of indicators = %f\n", (double)count / 100.0);
   return 0;
}
```

Figure 1: Using streams on the host

```
#include <mrg31k3p.h>
    ...
   size_t streamBufferSize;
   clrngMrg31k3pStream* streams = clrngMrg31k3pCreateStreams(NULL, numWorkItems,
                                        &streamBufferSize, &err);
    ...
   // Create buffer to transfer streams to the device.
   cl_mem buf_in = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                   streamBufferSize, streams, &err);
    ...
   // Create buffer to transfer output back from the device.
   cl_mem buf_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY,
                                   numWorkItems * sizeof(cl_double), NULL, &err);
    ...
   // The kernel takes two arguments; set them to buf_in, buf_out.
   err  = clSetKernelArg(kernel, 0, sizeof(buf_in),  &buf_in);
   err |= clSetKernelArg(kernel, 1, sizeof(buf_out), &buf_out);
    ...
   // Enqueue the kernel on device.
   cl_event ev;
   err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &numWorkItems, NULL, 0, NULL, &ev);
    ...
   // Wait for all work items to finish.
   err = clWaitForEvents(1, &ev);
    ...
   // Retrieve the contents of the output buffer from the device.
   double* out = (double*) malloc (numWorkItems * sizeof(double));
   err = clEnqueueReadBuffer(queue, buf_out, CL_TRUE, 0,
                              numWorkItems * sizeof(out[0]), out, 0, NULL, NULL);
    ...
```

Figure 2: Streams in work items: the host code

2.2 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. It is also possible (and sometimes useful) to use more than one stream per work item, as we shall see later. 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.

Figure 2 shows pieces of the host code, whose first line includes the clRNG header for the MRG31k3p RNG. The code assumes that we have an integer variable `numWorkItems` which indicates the number of work items we want to use. It is also assumed that the OpenCL context and command queue objects have already been created and stored in the `context` and `queue` variables. We create an array of `numWorkItems` streams (this allocates memory for both the array and the stream objects). The creator returns in `streamBufferSize` the size of the buffer that this array occupies (it depends on how much space is required to store the

```
#include <mrg31k3p.clh>

__kernel void example(__global clrngMrg31k3pHostStream* streams, __global double* out) {
   int gid = get_global_id(0);
   clrngMrg31k3pStream private_stream_d;   // This is not a pointer!
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &private_stream_d, &streams[gid]);
   out[gid] = clrngMrg31k3pRandomU01(&private_stream_d) +
              clrngMrg31k3pRandomU01(&private_stream_d);
   }
```

Figure 3: Streams in work items: the device code

stream states), and an error code.

Then we create an OpenCL 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 the results of the computations in an array of `numWorkItems` values of type `cl_double`. (OpenCL buffer creation is not specific to `clRNG`, so it is not discussed here). We finally enqueue the kernel with these two buffers as kernel arguments. The creation, compilation and linkage by the OpenCL C compiler of the code of Figure 2 is not shown here. We simply assume that the OpenCL kernel associated to the `example` function given in Figure 3 is stored in the `kernel` variable.

In the device code, partially shown in Figure 3, we include the device-side clRNG header for the chosen RNG (it ends with `.clh` instead of `.h`). Pointers to the global memory buffers received from the host and to the output array are passed to the kernel as arguments. (The correspondence between the kernel arguments and the buffers on the host is specified in the host code, not shown here). For each work item, we make a private copy of its stream named `private_stream_d`, in its private memory, so we can generate random numbers on the device. The private memory must be allocated at compile time (its declaration must allocate memory); this is why `private_stream_d` is not declared as a pointer (and we name it `private_stream_d` instead of `private_stream` to emphasize that it is "data" rather than a pointer to data). The kernel just generates two random numbers, computes the sum, and places it in `out[gid]` as a `cl_double`. The host can then recover the array of size `numWorkItems` that contains these sums; this is done with the last two lines of Figure 2.

3 Examples with a Simple Inventory Model

We now take a slightly more realistic example in which performance can be improved by performing simulation runs in parallel on a GPU device. This requires generating the random numbers directly on the devices when they are needed. The model is simplified compared with models that are commonly simulated by computer, but it provides an illustration of the typical use of multiple random streams and substreams in parallel computing, and in particular to

```
double inventorySimulateOneRun (int m, int s, int S,
       clrngMrg31k3pStream *stream_demand, clrngMrg31k3pStream *stream_order) {
   // Simulates the inventory model for m days, with the (s,S) policy,
   // and returns the average profit per day.
   int Xj = S, Yj;              // Stock Xj in the morning and Yj in the evening.
   double profit = 0.0;         // Cumulated profit.
   for (int j = 0; j < m; j++) {
      // Generate and subtract the demand for the day.
      Yj = Xj - clrngMrg31k3pRandomInteger (stream_demand, 0, L);
      if (Yj < 0)
         Yj = 0;                // Lost demand.
      profit += c * (Xj - Yj) - h * Yj;
      if ((Yj < s) && (clrngMrg31k3pRandomU01 (stream_order) < p)) {
         // We have a successful order.
         profit -= K + k * (S - Yj);
         Xj = S;
      } else
         Xj = Yj;
   }
   return profit / m;
}
```

Figure 4: Code to simulate the simple inventory system over m days

properly implement CRNs.

3.1 The Inventory Model and its Simulation

This simple inventory model is a modified version of the model given in Section 3.3 of the "Examples" document provided in [5]; see `http://simul.iro.umontreal.ca/ssj/examples/examples.pdf`. The Poisson distribution is replaced by a discrete uniform, so we can generate all the required random numbers directly from the uniform RNGs. Many simulation models encountered in finance (for example) have a similar structure to this one in terms of the need for multiple streams.

In this model, demands for a product on successive days are independent uniform random variables over the set $\{0, 1, \ldots, L\}$. If X_j denotes the stock level at the beginning of day j and D_j is the demand on that day, then there are $\min(D_j, X_j)$ sales, $\max(0, D_j - X_j)$ lost sales, and the stock at the end of day j is $Y_j = \max(0, X_j - D_j)$. There is a revenue c for each sale and a cost h for each unsold item at the end of the day. The inventory is controlled using a (s, S) policy: If $Y_j < s$, order $S - Y_j$ items, otherwise do not order. When an order is made in the evening, with probability p it arrives during the night and can be used for the next day, and with probability $1 - p$ it never arrives (in which case a new order will have to be made the next evening). When the order arrives, there is a fixed cost K plus a marginal cost of k per item. The stock at the beginning of the first day is $X_0 = S$.

We want to simulate this system for m successive days (note that these m days are not independent), for a given set of parameters and a given control policy (s, S), and replicate this simulation n times independently to estimate the expected profit per day over a time horizon of m days. Later, we will compare different choices of (s, S) by simulating each one n times, using CRNs across the values of (s, S). We will see that to do this properly, controlling the streams and substreams is crucial. Comparing values of (s, S) with CRNs is a basic ingredient in case we want to *optimize* (s, S) efficiently via simulation [12].

Figure 4 shows a C function that simulates the system for m days and returns the average cost per day. The device code will differ from the host code only by the presence of the `__kernel` and `__global` keywords. The procedure uses two different random streams: `stream_demand`, to generate the demands D_j, and `stream_order`, to decide which orders are received. For the latter, a uniform random variate over $(0, 1)$ is generated each time an order is made and the order is received if this variate is less than p. Why use two different streams? Because we will want to simulate the same model with the same random numbers for different values of (s, S) and we want to make sure that the same random numbers are used for the same purpose (e.g., each D_j should be the same) when (s, S) is changed, even if it changes the decisions of when and how to order. If we use a single stream for everything, a random number used to generate a demand for a given pair (s, S) could be used to decide if an order has arrived for another pair (s, S). We will see the impact of this later on. Here we only use two streams, but in practice, when simulating large systems, we may need hundreds of distinct streams to simulate different parts of the system. For example, one may think that the inventory system has hundreds or thousands of different products, that they are stored in different warehouses and the delivery times are random, etc.

To simulate n independent runs of this system, and make sure that the same random numbers are used for the same purpose for all choices of (s, S) in any given run, the standard approach is to use one distinct substream for each run, for each of the two streams [5, 6, 9]. This ensures that for any given run number, the streams start at the same place for all choices of (s, S). Another approach is to use a new pair of streams for each run, for a total of $2n$ distinct streams. For this small inventory example, this also works well, so one may argue that substreams are not really necessary. But for large simulation models that require many streams, using the same streams (and different substreams) across all runs is often much more convenient from a programming viewpoint and requires the creation of much fewer streams. The notion of streams with substreams also provides a good match with quasi-Monte Carlo (QMC) points, so using them can facilitate the transition from Monte Carlo to QMC in a simulation; see [5].

We will run experiments for this system with the following values: $L = 100$, $c = 2$, $h = 0.1$, $K = 10$, $k = 1$, and $p = 0.95$. We assume that L, c, h, K, k, and p have been set elsewhere in the code. We put s and S as parameters because we will vary them. We will also try different values of m and n.

We will consider the following two types of simulation experiments. In the first type, we select (m, s, S, n) and we want to estimate the expected average profit per day over m days. For this, we compute the average $\bar{P}_n$ and empirical variance S_n^2 of the n values $P_1, \ldots, P_n$ returned by the

```
void computeCI (int n, double *stat_tally) {
   // Computes and prints the average, variance, and a 95% CI on the mean
   // for the n values in stat_tally.
   // The variance is computed in a simple (unsafe) way, via a sum of squares.
   double sum = 0.0;
   double sum_squares = 0.0;
   for (int i = 0; i < n; i++) {
      sum += stat_tally[i];
      sum_squares += stat_tally[i] * stat_tally[i];
   }
   double average = sum / n;
   double variance = (sum_squares - average * sum) / (n - 1);
   double halfwidth = 1.96 * sqrt (variance / n); // CI half-width.
   printf("numObs\t\tmean\t\tvariance\t95%% confidence interval\n");
   printf("%d\t\t%f\t%f", n, average, variance);
   printf("\t[%f, %f]\n", average - halfwidth, average + halfwidth);
}
```

Figure 5: Computing and printing the average and a confidence interval

`inventorySimulateOneRun` function, and we compute and report a 95% confidence interval (CI) on the expected value. The boundaries of this CI are given by $(\bar{P}_n \pm 1.96 S_n/\sqrt{n})$. Figure 5 shows code to compute this, assuming that `stat_tally` points to an array that contains the n observations.

In the second type, we select m, n, and a set $\{(s_k, S_k),\ k = 0, \ldots, p-1\}$ of distinct policies (s, S), and we want to estimate the expected average profit for each policy, with CRNs across policies. As a special application of this, we may have $p = 2$ and want to estimate the *difference* in expected average profit between two policies (s_0, S_0), and (s_1, S_1).

We will show how to make such experiments first on a single CPU, then on a GPU device with several work items. This will illustrate various ways of using streams and substreams, and why they are useful.

3.2 Simulating n independent runs on a single CPU

Figure 6 shows three different ways of simulating n runs for a single policy, on a single CPU. The first approach, in function `inventorySimulateRunsOneStream()`, uses the first substream of a single stream to generate all the random numbers. The parameter `stream` contains this single stream, which is never reset to a new substream.

The second approach, in function `inventorySimulateRunsSubstreams`, uses two streams, named `stream_demand` and `stream_order`, and resets these two streams to a new substream after each run. This is a commonly used approach [5, 9]. It requires only two streams, regardless of the value of n.

```
void inventorySimulateRunsOneStream (int m, int s, int S, int n,
                         clrngMrg31k3pStream *stream, double *stat_profit) {
   // Performs n independent simulation runs of the system for m days with the
   // (s,S) policy, using a single stream and the same substream for everything,
   // and saves daily profit values.
   for (int i = 0; i < n; i++)
      stat_profit[i] = inventorySimulateOneRun (m, s, S, stream, stream);
}

void inventorySimulateRunsSubstreams (int m, int s, int S, int n,
           clrngMrg31k3pStream *stream_demand, clrngMrg31k3pStream *stream_order,
                          double *stat_profit) {
   // Similar to inventorySimulateRuns, but using two streams and their substreams.
   for (int i = 0; i < n; i++) {
      stat_profit[i] = inventorySimulateOneRun (m, s, S, stream_demand, stream_order);
      clrngMrg31k3pForwardToNextSubstreams(1, stream_demand);
      clrngMrg31k3pForwardToNextSubstreams(1, stream_order);
   }
}

void inventorySimulateRunsManyStreams (int m, int s, int S, int n,
      clrngMrg31k3pStream *streams_demand, clrngMrg31k3pStream *streams_order,
      double *stat_profit) {
   // Same as inventorySimulateRuns, but with two arrays of n streams each,
   // using a new pair of streams for each run.
   for (int i = 0; i < n; i++) {
      stat_profit[i] = inventorySimulateOneRun (m, s, S, &streams_demand[i],
                                                         &streams_order[i]);
   }
}
```

Figure 6: Simulating n runs on a single CPU, in three different ways

The third approach, in function `inventorySimulateRunsManyStreams`, uses a new pair of streams for each run, and only the first substream of each. This requires $2n$ distinct streams in total, whereas the first and second approaches require only one and two streams, respectively. Here we use n streams for the demands and n streams to determine which orders arrive. When n is very large, e.g., in the millions or more, creating and storing all those streams may add significant overhead. The piece of code in Figure 7 uses the latter function to simulate on the host (the other cases are similar).

3.3 Simulating n independent runs using n work items on a GPU

We now look at how to simulate the n independent replications with n work items running in parallel on a GPU device. Figure 8 shows a kernel to be executed on each work item to simulate one run. Figure 9 gives the skeleton of a function `inventorySimulateRunsGPU()` that prepares and launches the kernel to simulate n runs using $2n$ streams (we skip the details

```
 ...
clrngMrg31k3pStream *streams_demand = clrngMrg31k3pCreateStreams (NULL, n, NULL, NULL);
clrngMrg31k3pStream *streams_order  = clrngMrg31k3pCreateStreams (NULL, n, NULL, NULL);
double *stat_profit = (double *) malloc (n * sizeof (double));
inventorySimulateRunsManyStreams (m, s, S, n, streams_demand, streams_order, stat_profit);
computeCI (n, stat_profit);
 ...
```

Figure 7: Simulating n runs on a single CPU, third approach

```
__kernel void inventorySimulateGPU (int m, int s, int S,
            __global clrngMrg31k3pStreams *streams_demand,
            __global clrngMrg31k3pStreams *streams_order,
            __global double *stat_profits) {
   // Each of the n work items executes the following code.
   int gid = get_global_id (0);    // Id of this work item.
   // Make local copies of the stream states, in private memory.
   clrngMrg31k3pStreams stream_demand_d, stream_order_d; // Not pointers!
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d, &streams_demand[gid]);
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d,  &streams_order[gid]);
   stat_profits[gid] = inventorySimulateOneRun (m, s, S, &stream_demand_d,
                                                          &stream_order_d);
}
```

Figure 8: Kernel that calculates the daily profits on a work item

of this function). The figure then shows how to invoke this function to simulate the n runs by emulating the second and third approaches seen earlier, namely: (a) two streams with n substreams for each stream, and (b) $2n$ distinct streams. Here, we want to produce on the device *exactly* the same results as on a single CPU, for these two cases (a) and (b). We explain how to implement these two cases on n work items, and the consider other cases where we want to use only $n_1 \ll n$ work items (this may happen if n is very large).

Case (a). We need to construct an array that contains the starting points of the first n substreams, for each of the two streams, and pass them to the device. When executing the kernel, we need to maintain only the current state of each substream in the private memory of the work item. But each substream must be seen as a stream within the work item, because the functions that generate random numbers require a stream object as their first parameter. For this reason, by invoking `clrngMrg31k3pMakeSubstreams()`, we "create" two arrays of n streams which are actually n substreams of the same stream, and are copied from the host to the global memory of the device (inside the function `inventorySimulateRunsGPU()`). Each work item then picks its two streams from there and copies their current states (only) to its private memory. When random numbers are generated, only the states in private memory are changed.

```
// This function (details not show here) performs n runs in parallel on a GPU device,
// with two arrays of n streams, and saves the daily profit values.
void inventorySimulateRunsGPU (int m, int s, int S, int n,
       clrngMrg31k3pStreams *streams_demand, clrngMrg31k3pStreams *streams_order,
       double *stat_profit) {
   // Create structure that contains context, program, queue, etc.
      ...
   // Launch the kernel inventorySimulateGPU to execute on the GPU.
      ...
}

// (a) Simulate n runs on n work items using two streams and their substreams.
clrngMrg31k3pStream* stream_demand = clrngMrg31k3pCreateStreams(NULL, 1, NULL, NULL);
clrngMrg31k3pStream* stream_order  = clrngMrg31k3pCreateStreams(NULL, 1, NULL, NULL);
clrngMrg31k3pStream* substreams_demand = clrngMrg31k3pMakeSubstreams(stream_demand, n,
                                                                     NULL, NULL);
clrngMrg31k3pStream* substreams_order  = clrngMrg31k3pMakeSubstreams(stream_order,  n,
                                                                     NULL, NULL);
double *stat_profit = (double *) malloc (n * sizeof (double));
inventorySimulateRunsGPU (m, s, S, n, substreams_demand, substreams_order, stat_profit);
 ...

// (b) Simulate n runs on n work items using n distinct streams.
clrngMrg31k3pStream* streams_demand = clrngMrg31k3pCreateStreams(NULL, n, NULL, NULL);
clrngMrg31k3pStream* streams_order  = clrngMrg31k3pCreateStreams(NULL, n, NULL, NULL);
double *stat_profit = (double *) malloc (n * sizeof (double));
inventorySimulateRunsGPU (m, s, S, n, streams_demand, streams_order, stat_profit);
 ...
```

Figure 9: Simulating n runs on the GPU (a) with two streams and (b) with $2n$ streams

Perhaps one could think that instead of precomputing and storing the starting points of all these substreams in global memory, one could store only the stream in global memory, and each work item would find its appropriate substream. But then each work item would have to invoke `clrngMrg31k3pForwardToNextSubstreams()` a number of times that corresponds to its `gid` on its own copy of the stream. This would be highly inefficient, especially on a GPU device, where work items that take distinct execution paths, in this case by calling `clrngMrg31k3pForwardToNextSubstreams()` a distinct number of times, execute sequentially and not simulutaneously.

Case (b). Here, two arrays of n distinct streams are created and passed, instead of using substreams. The code is much simpler in this case.

```
#define  CLRNG_ENABLE_SUBSTREAMS
#include <mrg31k3p.clh>

__kernel void inventorySimulSubstreamsGPU (int m, int s, int S, int n2,
            __global clrngMrg31k3pStreams *streams_demand,
            __global clrngMrg31k3pStreams *streams_order,
            __global double *stat_profit) {
   // Each of the n work items executes the following to simulate n2 runs.
   int gid = get_global_id (0);    // Id of this work item.
   int n1  = get_global_size (0);  // Total number of work items.
   // Make local copies of the stream states, in private memory.
   clrngMrg31k3pStreams stream_demand_d, stream_order_d;
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d, &streams_demand[gid]);
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d,  &streams_order[gid]);
   for (int i = 0; i < n2; i++) {
      stat_profit[i * n1 + gid] = inventorySimulateOneRun (m, s, S,
                                     &stream_demand_d, &stream_order_d);
      clrngMrg31k3pForwardToNextSubstreams(1, &stream_demand_d);
      clrngMrg31k3pForwardToNextSubstreams(1, &stream_order_d);
   }
}

// This function (details not show here) performs n = n1 * n2 runs on a GPU device,
// with two arrays of n1 streams, and saves the daily profit values.
void inventorySimulateRunsSubstreamsGPU (int m, int s, int S, int n1, int n2,
       clrngMrg31k3pStreams *streams_demand, clrngMrg31k3pStreams *streams_order,
                 double *stat_profit) {
   // Create structure that contains context, program, queue, etc.
      ...
   // Launch the kernel inventorySimulateGPU to execute on the GPU.
      ...
}

// (c) Simulate n = n1 * n2 runs on n1 work items using n2 substreams on each.
double *stat_profit = (double *) malloc (n * sizeof (double));
clrngMrg31k3pStream* streams_demand = clrngMrg31k3pCreateStreams(NULL, n1, NULL, NULL);
clrngMrg31k3pStream* streams_order  = clrngMrg31k3pCreateStreams(NULL, n1, NULL, NULL);
inventorySimulateRunsSubstreamsGPU (m, s, S, n1, n2, streams_demand, streams_order, stat_profit);
computeCI (n, stat_profit);
 ...
```

Figure 10: Simulating n runs on the GPU (a) with two streams and (b) with $2n$ streams

3.4 Simulating n independent runs using $n_1 < n$ work items on a GPU

Case (c). Suppose now that n is very large and that instead of using n work items, we want to use n_1 work items to perform n_2 runs per work item, where $n_1 n_2 = n$. This would make sense when n is very large, say several millions. A natural way of implementing this is to use distinct streams across the work items, and n_2 substreams within each work item, without storing the initial states of those substreams in an array. The work item will run a loop very similar to that of `inventorySimulateRunsSubstreams()` in Figure 6. This is shown in Figure 10. The code marked `__kernel` simulates n_2 independent runs and is executed on each of the n_1 work items. The results of each run are stored in the `stat_profit` array. Note that the results of the n_1 runs that are simulated in parallel on the work items are stored in n_1 successive positions in the output buffer; this makes the access more efficient. The function `inventorySimulateRunsSubstreamsGPU` constructs the kernel required to run the simulations on the device. The code marked (c), below this function, shows how to call it. The advantage of doing this is that jumping to the next substream n_2 times within the work item is likely to be faster than creating n_2 streams on the host, copying them to global memory, and getting a new stream n_2 times from global memory, at least on a GPU. On an APU or other shared-memory computer, there may be no significant difference between the two approaches. Our experimental results will confirm this. This case (c) illustrates the usefulness of having substreams in the work items.

Case (d). To obtain exactly the same results as in Case (b), but using n_1 work items to make the computations, one can create $2n$ distinct streams, and use $2n_2$ distinct streams on each work item instead of using substreams. We can create the streams as in (b), and in the kernel, each work item will pick a new stream for each run. The downside of this is the need to store many more streams in global memory than in Case (c), so on a GPU, we expect this to be slower than Case (c) when n_2 is large. Our experiments will confirm this.

Case (e). To obtain exactly the same results as in (a), but using n_1 work items, we must use only two streams and n substreams for each, one substream for each run on each work item. For each stream, the n initial states of the substreams can be computed in advance and stored, exactly as in (a), then picked by the work items when needed. This also requires more memory, as in case (d).

Case (f). In what we have examined so far, we have different streams for the different types of random numbers required by the model, and substreams are only associated with simulation runs. This is what is typically done for a single processor. We could also consider an approach (f) where the roles of streams and substreams are interchanged. That is, we use one single stream for each simulation run, and one substream for each type of random numbers. Figure 11 shows a kernel that does that. In the present example, this means two substreams only, but this number can be much larger in more complex models. The advantage is that only one

stream needs to be passed to each work item when launching the kernel. One disadvantage is that one must then compute many substreams and transform each one into a stream inside each work item. This way of using streams and substreams may be convenient to the user in certain parallel applications. For serial simulation on a single processor, however, the need to copy each substream into a stream makes it unattractive and needlessly complicated.

```
__kernel void inventorySimulateGPU (int m, int s, int S,
            __global clrngMrg31k3pStreams *streams,
            __global double *stat_profits) {
   int gid = get_global_id (0);     // Id of this work item.
   // Make local copies of the substream states, in private memory.
   clrngMrg31k3pStreams stream_demand_d, stream_order_d; // Not pointers!
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d, &streams[gid]);
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d, &streams[gid]);
   clrngMrg31k3pForwardToNextSubstreams(1, &stream_order);
   stat_profits[gid] = inventorySimulateOneRun (m, s, S, &stream_demand_d,
                                                          &stream_order_d);
}
```

Figure 11: A kernel in which streams and substreams are interchanged

We report on experiments that compare cases (a) to (d) for $n = 2^{22}$ (about 4 millions), $m = 10$ and 100, and three choices of (n_1, n_2) for cases (c) and (d). We report the time for running the simulation on two different GPUs, and also on one CPU (or host) alone, and compare. For the "CPU" (host), we used a single core on an AMD A10-7850K APU. The first GPU was the GPU on this APU processor (on the same chip as the CPU); it is an AMD Radeon R7 Graphics with 720 Mhz clock frequency and 2GB of global memory, with 8 compute units and a total of 512 processing elements. The second GPU was an AMD Radeon HD 7900 Series with 925Mhz clock frequency and 3GB of global memory; it is a discrete GPU device located on a separate card, with 32 compute units and a total of 2048 processing elements. We call them GPU-A and GPU-D, respectively.

Table 1 first gives the statistical results (average, variance, and confidence interval) for the first method on the CPU (all other methods give approximately the same results). Then, for each case considered, it gives the timings (in seconds), and the speedup factor with respect to running n runs on a single CPU with a single stream, which is approach (1). The speedup factor is defined as the running time on the CPU divided by the running time for the considered method (e.g., a factor of 100 means that the considered method is faster by a factor 100). The given timings are averages over three trials. For cases (c) and (d), the best results are usually obtained with n_1 somewhere around 2^{16} to 2^{18} work items. On the discrete GPU, method (c) does much better than (d), which confirms our expectations (see the descriptions of those methods).

Table 1: Simulation results and timings for simulating n runs with n_1 work items for a single policy. In the second panel, for each entry, the first number is the computing time in seconds and the second number (in parentheses) is the speedup factor with respect to running n runs on a single CPU with a single stream, which is approach (1). Approaches (1) to (3) on the CPU and Cases (a) to (d) on the two GPUs are those explained earlier.

m	n	average $\bar{P}_n$	variance S_n^2	95% CI
100	2^{22}	36.583	9.0392	[36.5808, 36.5865]

	computing time in seconds (speedup factor)				
	n_1	$(m,n) = (10, 2^{22})$		$(m,n) = (100, 2^{22})$	
CPU-host (1)		4.32	(1)	43.24	(1)
CPU-host (2)		7.21	(0.6)	46.85	(0.9)
CPU-host (3)		4.43	(1.0)	44.25	(1.0)
GPU-A (a)		0.370	(12)	0.682	(63)
GPU-A (b)		0.287	(15)	0.638	(68)
GPU-A (c)	2^{14}	0.177	(24)	0.709	(61)
GPU-A (c)	2^{16}	0.170	(25)	0.674	(64)
GPU-A (c)	2^{18}	0.176	(25)	0.677	(64)
GPU-A (d)	2^{14}	0.243	(18)	0.593	(73)
GPU-A (d)	2^{16}	0.227	(19)	0.549	(79)
GPU-A (d)	2^{18}	0.231	(19)	0.524	(82)
GPU-D (a)		0.219	(20)	0.259	(167)
GPU-D (b)		0.217	(20)	0.269	(161)
GPU-D (c)	2^{14}	0.041	(103)	0.095	(456)
GPU-D (c)	2^{16}	0.039	(110)	0.086	(501)
GPU-D (c)	2^{18}	0.046	(93)	0.093	(466)
GPU-D (d)	2^{14}	0.230	(19)	0.339	(128)
GPU-D (d)	2^{16}	0.219	(20)	0.282	(153)
GPU-D (d)	2^{18}	0.275	(16)	0.268	(161)

```
// Assume that for k=0,...,p-1, (s[k], S[k]) defines policy k and has been initialized.

// Simulate n = n1 * n2 runs on n1 work items using n2 substreams, for p policies.
// We use the same 2 * n1 streams for all policies.
clrngMrg31k3pStream* streams_demand = clrngMrg31k3pCreateStreams(NULL, n1, NULL, NULL);
clrngMrg31k3pStream* streams_order  = clrngMrg31k3pCreateStreams(NULL, n1, NULL, NULL);
double *stat_profit = (double *) malloc (p * n * sizeof (double));
for (int k = 0; k < p; k++) {
   inventorySimulateRunsSubstreamsGPU (m, s[k], S[k], n1, n2,
                                        streams_demand, streams_order, &stat_profit[k*n]);
   clrngMrg31k3pRewindStreams(n1, stream_demand);
   clrngMrg31k3pRewindStreams(n1, stream_order);
   computeCI (n, &stat_profit[k*n]);
}

// Here we can compare the different policies.
// For example, in the simplest case where p=2, we may do:
double *stat_diff = (double *) malloc(n * sizeof (double));
for (int i = 0; i < n; i++)
   stat_diff[i] = stat_profit[n+i] - stat_profit[i];
computeCI (n, stat_diff);
 ...
```

Figure 12: Simulating n runs for p policies on a device, with CRNs across policies, with n_1 work items and n_2 runs per work item, for $n = n_1 n_2$.

3.5 Comparing Several Inventory Policies

We now give an example that illustrates why multiple streams and substreams are useful when comparing several similar systems via simulation using CRNs, and what difference in statistical accuracy CRN can make.

Simulating the policies sequentially. Suppose we want to compare several inventory policies, say $\{(s_k, S_k),\ k = 0, \ldots, p-1\}$, and run each policy on a GPU device, with CRNs across policies. We want to simulate each policy n times, using n_1 work items for each policy and n_2 independent runs per work item, where $n = n_1 n_2$ as before. One simple way to do this is to simulate one policy after the other, using n_1 work items in parallel, as shown in Figure 12. This code reuses the function `inventorySimulateRunsSubstreamsGPU()` from Figure 10.

The two statements `clrngMrg31k3pRewindStreams()` in the loop reset all the streams to their initial states after simulating each policy. Combined with the `clrngMrg31k3pResetNext-Substreams()` statements in the `inventorySimulSubstreamsGPU` kernel of Figure 10, this ensures that exactly the same random numbers will be used for any given run, for all policies. That is, for any policy, run i on work item `gid`$= j$ will (re)use substream i of stream j. This procedure ensures the synchronization of random numbers across policies even if different policies would use a different number of random numbers from certain streams for a given run

number. This is the main reason why we reset the streams to a new substream after each run.

In fact, the statements `clrngMrg31k3pRewindStreams()` could be removed here, because during the execution of `inventorySimulateRunsSubstreamsGPU()`, only a *copy* of each stream is modified on the GPU; the streams contained in the two arrays located on the host are still at their initial states and it would suffice to pass them again to the GPU via the kernel without invoking `clrngMrg31k3pRewindStreams`. We nevertheless leave the redundant `clrngMrg31k3pRewindStreams()` statements in the code, because in other situations, such as when everything is run on the host, the simulations might work directly with the original streams, and then the streams *must* be reset to their initial states.

Simulating the policies in parallel. The code of Figure 12 should be satisfactory if n is large enough. But in case n is not large and p is large, we may prefer to simulate the p policies in parallel instead. In Figure 13, we use $n_1 p$ work items and each one makes n_2 simulation runs. We use $2n_1$ streams and n_2 substreams per work item, as in Figure 12, and the codes from the two figures should produce exactly the same results, provided that they use the same $2n_1$ streams starting from the same seeds.

The buffer `stat_profit` in Figure 13 has dimension np and it contains the results in the following order. The first $n_1 p$ values are for the first run on each work item, then the next $n_1 p$ values are for the second run, etc. Within each block of $n_1 p$ values, the first n_1 successive values are for the first policy, the next n_1 values are for the second policy, etc. Thus, all the results of a given run for any given policy are in a block of n_1 successive positions in the array. And all the results that are computed in parallel at the same iteration number i of the loop are in a block of $n_1 p$ successive positions.

Comparing two policies: CRNs vs IRNs We performed an experiment with $p = 2$, with a code like in Figure 12, to estimate the expected difference in average daily profits for policies $(s_0, S_0) = (80, 198)$ and $(s_1, S_1) = (80, 200)$, with $m = 100$ and $n = 2^{22}$. We obtained $\bar{P}_n = 0.05860$, $S_n^2 = 0.190$, and a 95% confidence interval for the difference given by (0.05818, 0.05902).

To assess the benefit of using well-synchronized CRNs when comparing policies, we performed the same simulation experiments but with independent streams of random numbers across policies. To do this, we simply create new *independent* streams instead of resetting the streams to their initial states, after simulating each policy. That is, we replace the statements `clrngMrg31k3pRewindStreams` in Figure 12 by `clrngMrg31k3pCreateOverStreams` statements. For this, with the same values as above, we obtained $\bar{P}_n = 0.0569$, $S_n^2 = 18.1$, and a 95% confidence interval for the difference given by (0.0528, 0.0610). The variance S_n^2 is approximately 95 times larger than with CRNs. This means that with properly synchronized CRNs, we need approximately 95 times fewer simulation runs to obtain the same accuracy than with independent random numbers. This is a significant saving.

```
#define  CLRNG_ENABLE_SUBSTREAMS
#include <mrg31k3p.clh>

__kernel void inventorySimulPoliciesGPU (int m, int p, int *s, int *S, int n2,
            __global clrngMrg31k3pStreams *streams_demand,
            __global clrngMrg31k3pStreams *streams_order,
            __global double *stat_profit) {
   // Each of the n1*p work items executes the following to simulate n2 runs.
   int gid = get_global_id(0);     // Id of this work item.
   int n1p = get_global_size(0);   // Total number of work items.
   int n1  = n1p / p;              // Number of streams.
   int k   = gid / n1;             // Index of the policy this work item uses.
   int j   = gid % n1;             // Index of the stream that this work item uses.

   // Make local copies of the stream states, in private memory.
   clrngMrg31k3pStream stream_demand_d, stream_order_d;
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_demand_d, &streams_demand[j]);
   clrngMrg31k3pCopyOverStreamsFromGlobal(1, &stream_order_d, &streams_order[j]);

   for (int i = 0; i < n2; i++) {
       stat_profit[i * n1p + gid] = inventorySimulateOneRun(m, s[k], S[k],
                                              &stream_demand_d, &stream_order_d);
       clrngMrg31k3pForwardToNextSubstreams(1, &stream_demand_d);
       clrngMrg31k3pForwardToNextSubstreams(1, &stream_order_d);
   }
}

// This function (details not show here) performs n = n1 * n2 runs on a GPU device,
// with two arrays of n1 streams, and saves the daily profit values.
void inventorySimulateRunsPoliciesGPU (int m, int p, int *s, int *S, int n1, int n2,
       clrngMrg31k3pStreams *streams_demand, clrngMrg31k3pStreams *streams_order,
                 double *stat_profit) {
   // Create structure that contains context, program, queue, etc.
      ...
   // Launch the kernel inventorySimulPoliciesGPU to execute on the GPU.
      ...
}
```

Figure 13: Simulating n runs for p policies on a device, with CRNs across policies, with $n_1 p$ work items in parallel and n_2 runs per work item, for $n = n_1 n_2$.

	156	157	158	**159**	160	161	162	163	164	165	166	167
50	37.94537	37.94888	37.95166	37.95319	37.95274	37.95318	37.94887	37.94584	37.94361	37.94074	37.93335	37.92832
51	37.95740	37.96169	37.96379	37.96524	37.96546	37.96379	37.96293	37.95726	37.95295	37.94944	37.94536	37.93685
52	37.96725	37.97117	37.97402	37.97476	37.97492	37.97387	37.97100	37.96879	37.96184	37.95627	37.95154	37.94626
53	37.97356	37.97852	37.98098	37.98243	37.98187	37.98079	37.97848	37.97436	37.97088	37.96268	37.95589	37.94995
54	37.97593	37.98241	37.98589	37.98692	37.98703	37.98522	37.98290	37.97931	37.97397	37.96925	37.95986	37.95186
55	37.97865	37.98235	37.98740	**37.98940**	37.98909	37.98790	37.98483	37.98125	37.97641	37.96992	37.96401	37.95343
56	37.97871	37.98269	37.98494	37.98857	37.98917	37.98757	37.98507	37.98073	37.97594	37.96989	37.96227	37.95519
57	37.97414	37.98035	37.98293	37.98377	37.98603	37.98528	37.98239	37.97858	37.97299	37.96703	37.95981	37.95107
58	37.96869	37.97207	37.97825	37.97944	37.97895	37.97987	37.97776	37.97358	37.96848	37.96170	37.95461	37.94622
59	37.95772	37.96302	37.96630	37.97245	37.97234	37.97055	37.97010	37.96664	37.96122	37.95487	37.94695	37.93871
60	37.94434	37.94861	37.95371	37.95691	37.96309	37.96167	37.95860	37.95678	37.95202	37.94540	37.93785	37.92875
61	37.92200	37.93169	37.93591	37.94085	37.94401	37.95021	37.94751	37.94312	37.94000	37.93398	37.92621	37.91742

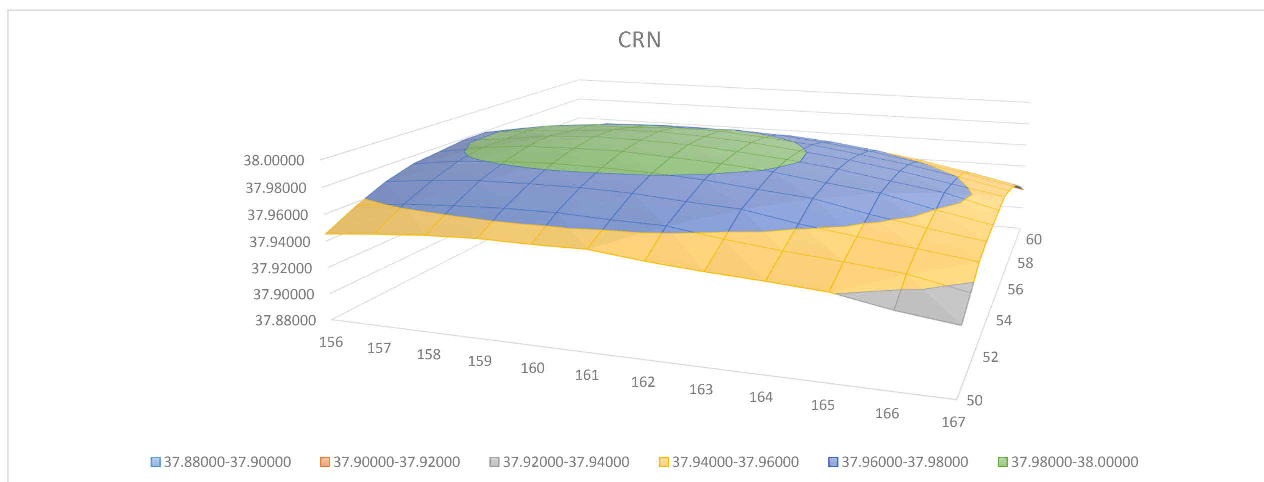

Figure 14: Comparing 144 policies with CRNs

Comparing many policies and estimating the cost function. To illustrate the use of simulation with CRNs to compare and possibly optimize policies, we now simulate a large number of policies (s, S) that belong to a grid, and plot the estimated cost as a function of (s, S). To define the grid, we take all 12 values of $s \in \{50, 51, \dots, 61\}$ and all 12 values of $S \in \{156, 157, \dots, 167\}$, and consider the cartesian product of these two sets, that is, the $p = 144$ policies defined by selecting one value from each set. Before calling the appropriate function in Figure 12 or 13, we must fill up two arrays of size $p = 144$, one for s and one for S, that contain the parameters of the 144 policies. We have estimated the expected profit per day for each of those $p = 144$ policies, first using CRNs with the code of Figure 12, and then with IRNs. For the latter, it suffices to replace the statements `clrngMrg31k3pRewindStreams` in Figure 12 by `clrngMrg31k3pCreateOverStreams` statements, to create $2n_1$ new streams for each policy.

We took $m = 100$, $n = 2^{18}$, and $n_1 = 2^{12}$ work items. We verified that the code of Figure 13 gives the same results for CRNs. Clearly, the sample function computed by the simulation varies much more smoothly with CRNs than with IRNs. Also, the optimizer of the sample function (the point where it reaches its maximum) is a much better (less noisy) estimate of the true optimal policy (the point (s, S) where the expected profit has its maximum) with CRNs than with IRNs. The optimum appears to be at (or very near) $(s, S) = (55, 159)$.

	156	157	158	159	160	161	162	163	164	165	166	167
50	37.94537	37.94888	37.94736	37.95314	37.95718	37.97194	37.95955	37.95281	37.96711	37.95221	37.95325	37.92063
51	37.95740	37.96650	37.95732	37.97337	37.98137	37.94273	37.96965	37.97573	37.95425	37.96074	37.94185	37.93139
52	37.96725	37.96166	37.97192	37.99236	37.98856	37.98708	37.98266	37.94671	37.95961	37.97238	37.95982	37.94465
53	37.97356	37.96999	37.97977	37.97611	37.98929	37.99089	38.00219	37.97693	37.98191	37.97217	37.95713	37.95575
54	37.97593	37.98520	37.99233	38.00043	37.99056	37.97440	37.98008	37.98817	37.98168	37.97703	37.97145	37.96138
55	37.97865	37.99460	37.97297	37.98383	37.99527	38.00068	38.00826	37.99519	37.96897	37.96675	37.95770	37.95672
56	37.97871	37.98670	37.97672	37.97440	37.99550	37.97120	37.96967	37.99717	37.97736	37.97275	37.97968	37.96523
57	37.97414	37.97797	37.98816	37.99192	37.96780	37.98415	37.97774	37.97844	37.99203	37.96531	37.97226	37.93934
58	37.96869	37.97435	37.96250	37.96581	37.97331	37.95655	37.98382	37.97144	37.97409	37.96631	37.96764	37.94759
59	37.95772	37.94725	37.97110	37.97905	37.97504	37.96237	37.98182	37.97656	37.97212	37.96762	37.96429	37.93976
60	37.94434	37.95081	37.94275	37.95515	37.98134	37.95863	37.96581	37.95548	37.96573	37.93949	37.93839	37.92030
61	37.92200	37.93006	37.92656	37.93281	37.94999	37.95799	37.96368	37.94849	37.95400	37.92439	37.90535	37.93375

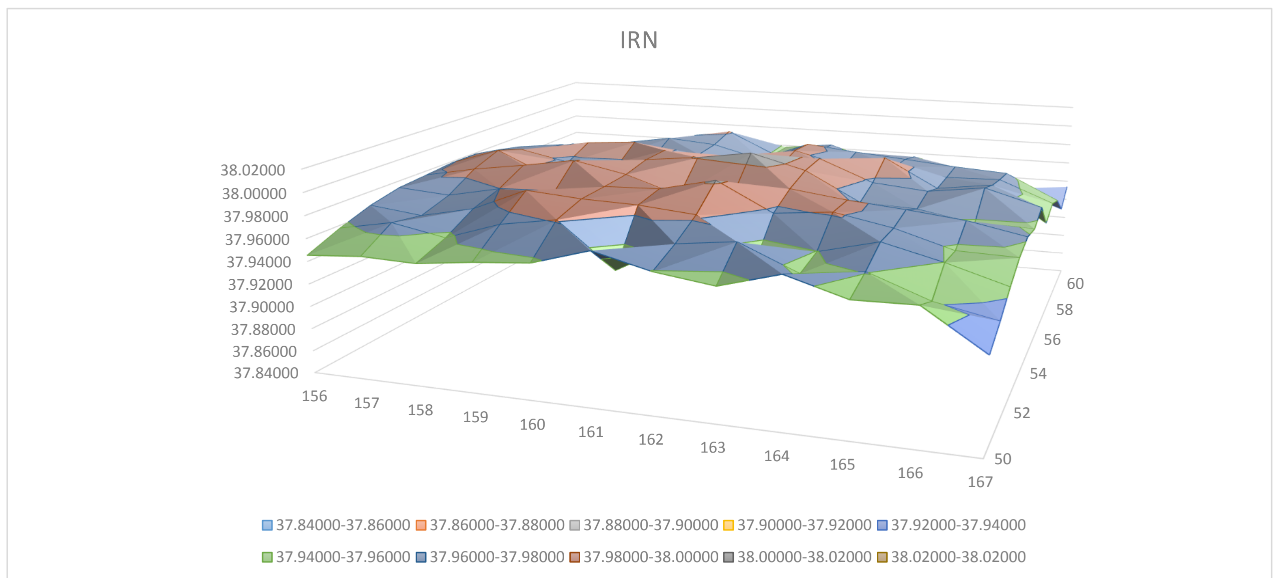

Figure 15: Comparing 144 policies with IRNs

Reusing the same streams and the convenience of multiple stream creators. As we said earlier, the codes of Figures 12 and 13 will produce exactly the same results if they use the same streams, starting from the same states. This will work if we run these codes in two different programs that execute independently, using the default stream creator in each case. Then the same streams will be created in the two programs.

In case we want to run the two codes sequentially in the same program, we must rewind and reuse the same streams for the second code, and not create new streams again, because then the new streams (and the results) will be different. In fact the code in Figure 12 already does the rewind at the end of the loop, so it would suffice to just reuse the streams already created.

In this example, the streams are created externally and passed as parameters to a function that simulates the system, for each policy. But for much larger and complicated simulation models, the number of streams can be very large and it is often more convenient to create them (as well as other objects used in the simulation) inside the simulation code (in a part that will run on the host). In that case, if we want to run the simulation several times with exactly the same streams (perhaps with different model parameters), then we need to reset the stream creator between those simulations so it can create the same streams over and over again. This can be achieved via the function `clrngRewindStreamCreator()`. An equivalent approach is to clone the default creator into multiple copies via the function `clrngCopyStreamCreator()`, and pass one of those copies to each simulation. This approach can be convenient when running the different simulations in parallel on different computers in a cluster, for example.

4 The API in a generic form

In this section, we discuss and define the API. The function and type names in this API all start with `clrng`. In each specific implementation, this prefix is expanded to a specific prefix; e.g., `clrngMrg31k3p` for the MRG31k3p generator.

In the standard case, streams and substreams are defined as in [5, 6, 9]. 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 `clrngSetBaseCreatorState()` 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 must be stored in the private memory of the work item that uses it in the kernel code to generate random numbers. In the current implementation, to avoid wasting (scarce) private memory, without the option `CLRNG_ENABLE_SUBSTREAMS` (see below), only the current state of the stream is stored explicitly in the work-item's private memory. The work item also keeps in private memory a pointer to the initial state of the stream, but this initial state is not copied into private memory, and the work item does not keep track of the initial state of the current substream. With the option `CLRNG_ENABLE_SUBSTREAMS` (see below), the initial state of the current substream is also stored into private memory. This permits one to rewind the current state to it or move forward to the next substream.

To use the clRNG library from within a user-defined kernel, the user must include the clRNG 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:

> `CLRNG_SINGLE_PRECISION`: With this option, all the random numbers returned by `clrngRandomU01()` and `clrngRandomU01Array()`, and those generated by `clrngDeviceRandomU01Array()`, will be of type `cl_float` instead of `cl_double` (the default setting). This option affects all implemented RNGs. This option can be activated on the device and the host separately (i.e., on either one on or both), and affects all implemented RNGs.
>
> `CLRNG_ENABLE_SUBSTREAMS`: With this option, the current state of a stream can be reset on the device to the initial state of the current or next substream. This is made possible by storing in private memory the initial substream state. Without this option, by default, this is not possible and only the current state of the stream and a pointer to its initial state (left in global memory) are kept in private memory and is accessible, in a work item. This option applies only to the device; operations on substreams are always available on the host.

For example, to enable substreams support, generate single-precision floating point numbers on the device, and use the MRG31k3p generator, one would have:

```
#define  CLRNG_ENABLE_SUBSTREAMS
#define  CLRNG_SINGLE_PRECISION
#include <mrg31k3p.clh>
```

4.1 Host interface

The functions described here are all available on the host, in all implementations, unless specified otherwise. Only some of the functions are also available on the device in addition to the host; they are listed in Section 4.2. Some functions return an error code in `err`.

```
typedef struct { /* ... */ } clrngStreamState;
```

Contains the state of a random stream. The definition of a state depends on the type of generator.

```
typedef struct { /* ... */ } clrngStream;
```

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.

The device API offers a variant of this struct definition called `clrngHostStream` to receive stream objects from the host. Stream objects, as defined on the device, do not store as much information as stream objects on the host, but keep pointers to relevant information from the host stream object. The definition of the `clrngStream` type on the device also depends on whether substreams support is required by the user (with the `CLRNG_ENABLE_SUBSTREAMS` option).

```
typedef struct { /* ... */ } clrngStreamCreator;
```

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.

```
clrngStreamCreator* clrngCopyStreamCreator(const clrngStreamCreator* creator,
                                           clrngStatus* err);
```

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 `clrngCopyStreamCreator(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.

```
clrngStatus clrngDestroyStreamCreator(clrngStreamCreator* creator);
```

Release the resources associated to a stream creator object.

```
clrngStatus clrngRewindStreamCreator(clrngStreamCreator* creator);
```

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

```
clrngStatus clrngSetBaseCreatorState(clrngStreamCreator* creator,
                                     const clrngStreamState* baseState);
```

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.

```
clrngStatus clrngChangeStreamsSpacing(clrngStreamCreator* creator,
                                      cl_int e, cl_int c);
```

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 $Z = 2^e + c$ if $e > 0$, or to $Z = c$ if $e = 0$. One must have $e \geq 0$ 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.

```
clrngStream* clrngAllocStreams(size_t count, size_t* bufSize, clrngStatus* 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.

```
clrngStatus clrngDestroyStreams(clrngStream* streams);
```

Release the memory space taken by those stream objects.

```
clrngStream* clrngCreateStreams(clrngStreamCreator* creator, size_t count,
                                size_t* bufSize, clrngStatus* err);
```

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 `count = 1`.

```
clrngStatus clrngCreateOverStreams(clrngStreamCreator* creator,
                                   size_t count, clrngStream* streams);
```

This function is similar to `clrngCreateStreams()`, 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 `clrngAllocStreams()` or `clrngCreateStreams()`. It permits the client to reuse memory that was previously allocated for other streams.

```
clrngStream* clrngCopyStreams(size_t count, const clrngStream* streams,
                              clrngStatus* err);
```

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.

```
clrngStatus clrngCopyOverStreams(size_t count, clrngStream* destStreams,
                                 const clrngStream* srcStreams);
```

Copy (or restore) the stream object `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. Note: The device API offers variants of this function to convert stream objects across their host and device representations, while copying across different types of memory.

```
cl_double clrngRandomU01(clrngStream* stream);
```

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 `cl_double`. But if the option `CLRNG_SINGLE_PRECISION` is defined on the host, the returned value will be of type `cl_float`. Setting this option changes the type of the returned value on the host for all RNGs and all functions that use `clrngRandomU01()`.

```
cl_int clrngRandomInteger(clrngStream* stream, cl_int i, cl_int j);
```

Generate and return a (pseudo)random integer from the discrete uniform distribution over the integers $\{i, \ldots, j\}$, using `stream`, by calling `clrngRandomU01()` once and transforming the output by inversion. That is, it returns `i + (cl_int)((j-i+1) * clrngRandomU01(stream))`.

```
clrngStatus clrngRandomU01Array(clrngStream* stream, size_t count,
                                cl_double* buffer);
```

Fill preallocated `buffer` with `count` successive (pseudo)random numbers. Equivalent to calling `clrngRandomU01(stream)` `count` times to fill the buffer.

In case `CLRNG_SINGLE_PRECISION` is defined, the buffer will be filled by `count` values of type `cl_float` instead.

```
clrngStatus clrngRandomIntegerArray(clrngStream* stream, cl_int i, cl_int j,
                                    size_t count, cl_int* buffer);
```

Same as `clrngRandomU01Array()`, but for integer values in $\{i, \ldots, j\}$. Equivalent to calling `clrngRandomInteger()` `count` times to fill the buffer.

```
clrngStatus clrngRewindStreams(size_t count, clrngStream* streams);
```

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

```
clrngStatus clrngRewindSubstreams(size_t count, clrngStream* streams);
```

Reinitialize all the streams in `streams` to the initial states of their current substream.

```
clrngStatus clrngForwardToNextSubstreams(size_t count, clrngStream* streams);
```

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.

```
clrngStream* clrngMakeSubstreams(clrngStream* stream, size_t count,
                                 size_t* bufSize, clrngStatus* err);
```

Make and return an array of `count` copies of `stream`, whose initial and current states are the initial states of the next `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 put `count = 1`. When this function is invoked, the substream state and initial state of `stream` are advanced by `count` substreams.

```
clrngStatus clrngMakeOverSubstreams(clrngStream* stream,
                                    size_t count, clrngStream* substreams);
```

This function is similar to `clrngMakeStreams()`, 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 `clrngAllocStreams()`, `clrngMakeSubstreams()` or `clrngCreateStreams()`. It permits the client to reuse memory that was previously allocated for other streams.

```
clrngStatus clrngAdvanceStreams(size_t count, clrngStream* streams,
                                cl_int e, cl_int c);
```

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 $k = 2^e + c$; 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.

```
clrngStatus clrngDeviceRandomU01Array(size_t streamCount, cl_mem streams,
               size_t numberCount, cl_mem outBuffer, cl_uint numQueuesAndEvents,
               cl_command_queue* commQueues, cl_uint numWaitEvents,
               const cl_event* waitEvents, cl_event* outEvents);
```

Fill the buffer pointed to by `outBuffer` with `numberCount` uniform random numbers of type `cl_double` (or `cl_float` in case `CLRNG_SINGLE_PRECISION` is defined), using `streamCount` work items. In the current implementation, `numberCount` must be a multiple of `streamCount`. See `clEnqueueNDRangeKernel()` from the OpenCL API documentation for a description of the `numWaitEvents` and `waitEvents` arguments. This function requires access to the `clRNG` device header files (like `mrg31k3p.clh`) and assumes that the environment variable `CLRNG_ROOT` points to the installation path of the `clRNG` package, where lies the `cl/include` subdirectory that contains these files. Means of setting an environment variable depend on the operating system used.

```
clrngStatus clrngWriteStreamInfo(const clrngStream* stream, FILE *file);
```

Format and output information about a stream object to a file. The `file` can be set to `stdout` for standard output and to `stderr` for the error file.

4.2 Device interface

The functions that can be called on the device are the following.

4.2.1 Functions that are always available

```
clrngStatus clrngCopyOverStreams(size_t count, clrngStream* destStreams,
                                 const clrngStream* srcStreams);
```

```
clrngStatus clrngCopyOverStreamsFromGlobal(size_t count,
           clrngStream* destStreams, __global const clrngHostStream* srcStreams);
```

Copy the host stream objects `srcStreams` from global memory as device stream objects into the buffer `destStreams` in private memory.

```
clrngStatus clrngCopyOverStreamsToGlobal(size_t count,
           __global clrngHostStream* destStreams, const clrngStream* srcStreams);
```

Copy the device stream objects `srcStreams` from private memory as host stream objects into the buffer `destStreams` in global memory.

```
cl_double clrngRandomU01(clrngStream* stream);
```

By default, the returned value is of type `cl_double`. But if the option `CLRNG_SINGLE_PRECISION` is defined on the device, the returned value will be of type `cl_float`, for all RNGs.

```
cl_int clrngRandomInteger(clrngStream* stream, cl_int i, cl_int j);
```

```
clrngStatus clrngRandomU01Array(clrngStream* stream, size_t count,
                                cl_double* buffer);
```

```
clrngStatus clrngRandomIntegerArray(clrngStream* stream, cl_int i, cl_int j,
                                    size_t count, cl_int* buffer);
```

```
clrngStatus clrngRewindStreams(size_t count, clrngStream* streams);
```

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

4.2.2 Functions that are available only if `CLRNG_ENABLE_SUBSTREAMS` has been set

```
clrngStatus clrngRewindSubstreams(size_t count, clrngStream* streams);
```

```
clrngStatus clrngForwardToNextSubstreams(size_t count, clrngStream* streams);
```

```
clrngStatus clrngMakeOverSubstreams(clrngStream* stream,
                                    size_t count, clrngStream* substreams);
```

5 Implementation for the MRG31k3p generator

The MRG31k3p generator is defined in [10]. In its specific implementation, the function and type names start with `clrngMrg31k3p`. For this RNG, a *state* is a vector of six 31-bit integers, represented internally as `cl_uint`. 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.

6 Implementation for the MRG32k3a generator

MRG32k3a is a combined multiple recursive generator (MRG) proposed by L'Ecuyer [2], 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 [9]. The state of a stream at any given step is a six-dimensional vector of 32-bit integers, but those integers are stored as `cl_ulong` (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 [2, 9], 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.

7 Implementation for the LFSR113 generator

The LFSR113 generator is defined in [3]. In its implementation, the function and type names start with `clrngLfsr113`. For this RNG, a *state* is a vector of four 31-bit integers, represented internally as `cl_uint`. 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 `clrngLfsr113ChangeStreamsSpacing` and `clrngLfsr113AdvancedStreams` are not implemented in the current version.

8 Implementation for 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 [8] (the other three generators also have). In its implementation, the function and type names start with `clrngPhilox432`. 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 `cl_uint` values and the index, as a single `cl_uint` value. In the current clRNG version, the key is the same for all streams, with all bits set to zero as in the `Engine` module of [11], 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 (zero), initial counter value 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 `clrngPhilox432ChangeStreamsSpacing` supports only values of c that are multiples of 4, with either $e = 0$ or $e \geq 2$.

9 Conclusion and future plans

In the future, we plan to provide implementations of this API with other types of base generators, provide types of streams that can create children streams independently of each other, provide special types of streams that produce the successive coordinates of quasi-random points, build a generic interface that can be used for all those different types of streams, and provide facilities to transform uniform random numbers to non-uniform ones from various distributions.

References

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

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

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

[4] 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.

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

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

[7] P. L'Ecuyer, D. Munger, 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`.

[8] 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.

[9] 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.

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

[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.

[12] A. Shapiro, D. Dentcheva, and A. Ruszczyński, editors. *Lecture Notes on Stochastic Programming: Modeling and Theory.* SIAM, Philadelphia, 2009.