A tool set for random number generation on GPUs in R

Ruoyong Xu
University of Toronto

Patrick Brown
University of Toronto

Pierre L'Ecuyer
University of Montréal

Abstract

We introduce the R package **clrng** that leverages the **gpuR** package, and can generate random numbers on GPU by utilizing the **clRNG** (OpenCL) library. Parallel processing by graphics processing unit (GPU) can be used to speed up computationally intensive tasks, which when combined with R, it can largely improve R's downsides in terms of slow speed, memory usage and computation mode. There is currently no R package that does random number generation on GPU, while random number generation is critical in simulation-based statistical inference and modeling. **clrng** enables reproducible research by setting random streams on GPU and can thus accelerate several types of simulation and modelling. This package is portable and flexible, developers can use its random number generation kernel for various other purposes and applications.

Keywords: GPU, **clrng** package, parallel computing, **clRNG** library.

1. Introduction

In recent years, parallel computing with R [R Core Team 2021] has become a very important topic and attracted lots of interest from researchers [see Eddelbuettel 2021, for a review]. Although R is one of the most popular statistical software with many advantages, it has drawbacks in memory usage and computation mode aspects [Zhao 2016]. To be more specific, (1) R requires all data to be loaded into the main memory (RAM) and thus can handle a very limited size of data; (2) R is a single-threaded program, it can not effectively use all the computing cores of multi-core processors. Parallel computing is the solution to these drawbacks, for an overview of current parallel computing approaches with R, see CRAN Task View by Eddelbuettel at `https://cran.r-project.org/web/views/HighPerformanceComputing.html`.

Graphics Processing Units (GPUs) have the potential to make important contribution to parallel computing with R. GPUs can perform thousands of tests simultaneously, which makes them powerful for doing massively parallel computing, and they are relatively cheap compared to multicore CPU's. Although there have been a number of R packages developed which provide some GPU capability, they inevitably come with some limitations. Packages such as **gputools**, **gpumatrix**, **cudaBayesreg**, **rpud** (available on github), are now no longer maintained, the popular **tensorflow** [Allaire, Eddelbuettel, Golding, and Tang 2016] package uses GPU via Python, which makes it difficult to include as a dependency for new R packages. All of these mentioned packages are restricted to R users with NVIDIA GPUs.

gpuR [Determan Jr. 2017] is the only R package with a convenient and flexible interface between R and GPUs, and it is compatible with many GPU devices. By utilizing the **ViennaCL** [Rupp, Tillet, Rudolf, Weinbub, Morhammer, Grasser, Jungel, and Selberherr 2016] library, it provides a bridge between R and non-proprietary GPUs through the OpenCL (Open Computing Language) backend, which when combined with **Rcpp** [Eddelbuettel and François 2011] gives a building block for other R packages.

Random number generation is critical in simulation-based statistical inference, machine learning and many other scientific fields. While most random number generators are sequential, the R packages **parallel**, **rlecuyer** [Sevcikova, Rossini, L'Ecuyer, and Sevcikova 2015] and **future** [Bengtsson 2021] are able to generate random numbers in parallel on multicore CPUs. More specifically, **parallel** writes an interface for the **RngStreams**, a C++ library by L'ecuyer, Simard, Chen, and Kelton [2002] which is based on a combined multiple-recursive generator (MRG). **future** also uses the combined MRG algorithm for generating random numbers. For up-to-date review papers on the generation of random numbers on parallel devices, and GPUs in particular, see: L'Ecuyer [2015]; L'Ecuyer, Munger, Oreshkin, and Simard [2017]; L'Ecuyer, Nadeau-Chamard, Chen, and Lebar [2021].

The **clrng** package described here is currently the only R package that is able to generate random numbers on GPUs. Accomplishing this is complicated to do because each process must produce an independent sequence of random numbers, and in order to ensure reproducibility it should be possible to save and restore the current state of each stream of numbers at any point. Here we introduce the R package **clrng** that leverages the **gpuR** package, and can generate random numbers on GPU by using the **clRNG** [L'Ecuyer, Munger, and Kemerchou 2015] library, and can thus accelerate several types of statistical simulation and modelling.

The remaining sections are organized as follows: In Section 2, we introduce streams and the use of streams in work-items on a GPU device for generating uniform random numbers, and the usage of `clrng::runif()`. In Section 3, we introduce two non-uniform RNGs in **clrng**, as examples for users to develop other RNGs of interest on GPUs in R. In Section 4, we apply GPU-generated uniform random numbers in Monte Carlo simulation for Fisher's exact test, we introduce how the random numbers are used and how the algorithm is parallelized and implemented on GPU. Then we provide two real data examples to demonstrate the function usage and its R performance. Section 5, we show an useful application of normal random numbers on GPU, using them to simulate batches of Gaussian random surfaces with Matérn covariance matrices simultaneously, the simulation also uses GPU-enabled functions from our another package **gpuBatchMatrix**. Finally, the paper concludes with a short summary and a discussion in Section 6.

2. Uniform random number generation

Uniform random number generators (RNGs) are the foundation for simulating random numbers from all types of probability distributions. L'Ecuyer [2012] summarized the usual two steps to generate a random variable in computational statistics: (1) generating independent and identically distributed (i.i.d.) uniform random variables on the interval $(0, 1)$, (2) applying transformations to these i.i.d. $U(0, 1)$ random variables to get samples from the desired distribution. L'Ecuyer [2012] and Robert and Casella [2004] present many general transforma-

tion methods for generating non-uniform random variables, for example, the most frequently used inverse transform method, the Box-Muller algorithm [Box 1958] for Gaussian random variable generation, and so on, which are all built on uniform random variables.

clRNG is an OpenCL library for uniform random number generation, it provides four different RNGs: the MRG31k3p, MRG32k3a, LFSR113, and Philox-4×32-10 generators. These four RNGs use different types of constructions. **clrng** package uses the MRG31k3p RNG, making it able to generate random numbers on GPUs. We choose MRG31k3p as the base generator for the following reasons: the original **RNGStreams** package [L'ecuyer *et al.* 2002] was built with MRG32k3a, which was designed to be implemented in double precision, and not with 32-bit integers. The MRG31k3p generator was designed later, specifically for 32-bit integer arithmetic, so it runs faster on the 32-bit GPUs. It is also faster than Philox-4×32-10 [L'Ecuyer and Touzin 2000]. MRG31k3p was also statistically tested extensively and successfully, [see L'Ecuyer and Simard 2007].

In what follows, we will illustrate how to create streams and how to use streams to generate uniform random numbers.

2.1. Creating streams

Random number generation is more complicated when the RNG algorithm is run in parallel processes in R, there is a risk that the generated sequences of random numbers have correlation. **clrng** uses multiple distinct streams that are used in work-items that executes in parallel on a GPU device. The popular way of obtaining multiple streams is to take an RNG with a long period and cut the RNG's sequence into very long pieces, these pieces are adjacent and of equal length Z, so each piece can be used as a separate stream. Creating a new stream amounts to computing its starting point (seed). In general, a stream object contains three states: the current state of the stream, the initial state of the stream (or seed), and the initial state of the current substream (by default it is equal to the seed). Each of these streams can also be partitioned into substreams with equally-spaced starting points [L'ecuyer *et al.* 2002; L'Ecuyer 2015], which are occasionally useful.

in clRNG library, for the MRG31k3p RNG, the entire period length of approximately 2^{185} is divided into approximately 2^{51} non-overlapping streams of length $Z = 2^{134}$. Each stream can be further partitioned into substreams of length $W = 2^{72}$, although this is not currently implemented in **clrng**. 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, while providing a sufficient period length for most applications. The initial state of the first stream (also called "initial seed for the package" in clRNG and "initial" in **clrng**) for the MRG31k3p is by default $(12345, 12345, 12345, 12345, 12345, 12345)$. Streams are created sequentially in the way that 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 function `clrngMrg31k3pCreateStreams()` from clRNG creates streams on the host using the MRG31k3p RNG. If we want to use the streams in work-items on a GPU device, the streams have to be copied from the host to the global memory of the GPU device, and then each work-item copies the current states only of the streams to its private memory. **clrng** is able to create streams both on the host and on the GPU device. `createStreamsCpu()` is an interface function of `clrngMrg31k3pCreateStreams()` that creates streams on host as R matrices, see the R output below, where we created 4 streams on

the host using `createStreamsCpu()`, after transposing `myStreamsCpu`, each column of the matrix is a stream, creating streams on host makes it easy for us to view and check the streams behavior. Substreams are not created in **clrng** because currently we use one stream per work-item.

```
# creating streams on CPU
library("gpuR")
library("clrng")
myStreamsCpu <- createStreamsCpu(n=4, initial=12345)
t(myStreamsCpu)
##                [,1]       [,2]       [,3]       [,4]
## current.g1.1 12345  336690377  502033783  739421137
## current.g1.2 12345  597094797 1322587635 1475938232
## current.g1.3 12345 1245771585 1964121530  730262207
## current.g2.1 12345   85196284 1949818481 1630192198
## current.g2.2 12345  523477687 1607232546  324551134
## current.g2.3 12345 2094976052 1462898381  795289868
## initial.g1.1 12345  336690377  502033783  739421137
## initial.g1.2 12345  597094797 1322587635 1475938232
## initial.g1.3 12345 1245771585 1964121530  730262207
## initial.g2.1 12345   85196284 1949818481 1630192198
## initial.g2.2 12345  523477687 1607232546  324551134
## initial.g2.3 12345 2094976052 1462898381  795289868
```

`createStreamsCpu()` has two arguments:

- `n`: number of streams to create.
- `initial`: vector of length 6, recycled if shorter.

We can have streams on GPU by either converting the `myStreamsCpu` to a '`vclMatrix`' or producing streams using `createStreamsGpu()`. `createStreamsGpu()` is adapted from the clrngMrg31k3pCreateStreams() and creates streams directly on a GPU device. Setting a seed same as the MRG31k3p default seed in `createStreamsGpu()`, we produce on device exactly the same streams as those created on host as follows. Notice that the code is just for showing `createStreamsGpu()` and `createStreamsCpu()` are R interface functions for `clrngMrg31k3pCreateStreams()` on GPU and CPU respectively, in practice it is not recommended to create a new stream that has the same initial seed as another existing stream in one simulation program.

```
# creating streams on GPU
myStreamsGpu1 = vclMatrix(myStreamsCpu)
myStreamsGpu2 = createStreamsGpu(n=4, initial=12345)
range(as.matrix(myStreamsGpu1 - myStreamsGpu2))
## [1] 0 0
```

`createStreamsGpu()` also has the two arguments `n` and `initial`. `createStreamsGpu()` is similar to `.Random.seed` in R. By setting the RNG status with the R function `set.seed()`,

we are able to repeat the sequence of random seeds, which is an important quality criteria of RNGs. Similarly in `createStreamsGpu()` and `createStreamsCpu()`, with the `initial` argument, we make the sequence of streams repeatable.

2.2. Using streams to generate uniform random numbers

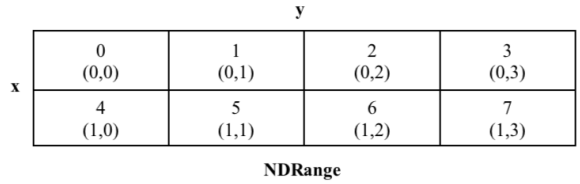
The streams created sequentially on a GPU are then used in work-items that executes in parallel on a GPU device. In **clrng** each work-item takes one distinct stream to generate random numbers, so the number of streams should always be equal to the number of total work-items in use. The main part of the kernel (OpenCL functions for execution on the device) for generating uniform random numbers is shown in Listing 1. Kernels are written in the OpenCL C language, in which `__kernel` declares a function as a kernel, pointer kernel arguments must be declared with an address space qualifier, for example `__global` and `__local`. Here the pointers to `streams` and to the output matrix `out` on the global memory are passed to the kernel as arguments. `Nrow` and `Ncol` represent row and column number of the matrix `out` respectively. `NpadCol` is the internal number of columns of `out`, `NpadStreams` is the internal number of columns of the `streams` (these variables are defined in macros not shown here). Each stream's current state is copied to the private memory of each work-item by the function `streamsToPrivate()`. `index` represents the position of the work-item when mapping it to a one-dimensional data structure. The function `clrngMrg31k3pNextState()` generates an uniform random integer called `temp` between 1 and 2147483647, the value of `temp` is then scaled to be in the interval $(0, 1)$ by multiplying it by a constant `mrg31k3p_NORM_cl`, which is defined to be 1/2147483648, if to generate double-precision random numbers. At the end of generating random numbers, streams are transferred back to global memory through the function `streamsFromPrivate()`.

Listing 1: Using streams to produce random U(0,1) numbers

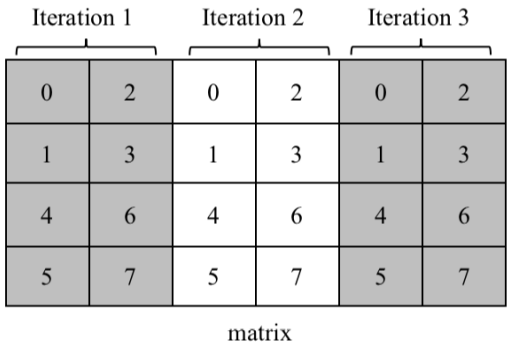
```
__kernel void mrg31k3pMatrix(
    __global int* streams,
    __global double* out){

const int
index = get_global_id(0)*get_global_size(1) + get_global_id(1),
DrowInc = 2*get_global_size(0),
DrowStartInc = DrowInc * NpadCol,
startvalue=index * NpadStreams;
int Drow, Dcol, DrowStart, Dentry;
uint temp;
uint g1[3], g2[3];
streamsToPrivate(streams,g1,g2,startvalue);

//filling in random numbers in the output matrix
for(Drow=2*get_global_id(0),
    DrowStart = (Drow + get_local_id(1))* NpadCol;
    Drow < Nrow; Drow += DrowInc, DrowStart += DrowStartInc) {

   for(Dcol=get_group_id(1), Dentry = DrowStart + Dcol;
```

```
        Dcol < Ncol;
        Dcol += get_num_groups(1), Dentry += get_num_groups(1)) {

        temp = clrngMrg31k3pNextState(g1, g2);
        out[Dentry] = mrg31k3p_NORM_cl * temp;

    }//Dcol
 }//Drow
streamsFromPrivate(streams,g1,g2,startvalue);
}
```

This kernel executes over a 2-dimensional NDRange. Figure 1 illustrates how this kernel executes in practice with a simple example. Suppose we have a 2×4 NDRange ((a) in Figure 1) and want to create a 4×6 matrix ((b) in Figure 1) of uniform random numbers. In each grid of (a) that represents a work-item, there are two rows of numbers, the number in the above row is the stream index (from 0 to 7), and a pair of numbers in the second row is the 2-dimensional global ID of the work-item. Each stream object is passed to each work-item. The number in a cell of the matrix corresponds to the stream index, it indicates which stream generates a random number for this matrix cell. For example, stream 0 which is allocated in work-item $(0, 0)$ generates a random number in matrix $(0, 0)$ in the first iteration, then moves on to matrix $(0, 2)$ and generates a random number there in the second iteration, in the third iteration it generates a random number in matrix $(0, 4)$ and so on until it goes out of the matrix range in the next iteration.



(a)



(b)

Figure 1: Using streams to produce a 4×6 matrix of random U(0,1) numbers.

Now we use the streams created in section 2.1 to generate two vectors of double-precision i.i.d. $U(0,1)$ random numbers with the function `runif()`. `runif()` can also be used to produce random integers in $[1, 2147483647]$. To view the generated random numbers, we need to convert them to R vectors or matrices, by doing this, the random numbers are moved from the global memory of the GPU device to the host.

```
as.vector(clrng::runif(n = 6, streams = myStreamsGpu1, Nglobal = c(1, nrow(myStreamsGpu1)),
    type = "double"))
## [1] 0.7353245 0.5180770 0.6142074 0.2319392 0.1100781 0.3619766
as.vector(clrng::runif(n = 10, streams = myStreamsGpu2, Nglobal = c(1,
    nrow(myStreamsGpu1)), type = "double"))[-(1:6)]
## [1] 0.6487742 0.1112075 0.3661944 0.5018562
```

The arguments of the `clrng::runif()` are described as follows:

- `n`: a vector of length 2 specifying the row and column number if to create a matrix, or a number specifying the length if to create a vector.
- `streams`: streams for random number generation.
- `Nglobal`: global index space. Default is set as $(64, 8)$.
- `type`: "double" or "float" or "integer" format of generated random numbers.

`runif()` returns the same sequence of random numbers because it uses the same streams `myStreamsGpu1` and `myStreamsGpu2` here. Reusing `myStreamsGpu1` will produce a vector different from `sim_1`. Because each time a random number is generated, the current state of the stream advances by one position.

Unlike the objects created by R, by default they remain in the memory after a restart. Streams created on GPU does not remain in the memory when a new R session begins. However, we can save the streams on the CPU in a file using the `saveRDS()` function, and later recall the streams with the `readRDS()` function, in this way we can reproduce the exact same sequence of random numbers in simulations. Below is the code that saves streams to a data file called `streams_from_GPU.rds` on CPU and then load it back and transfer it to GPU.

```
saveRDS(as.matrix(createStreamsGpu(n=4)), "myStreams.rds")
# Load the streams object as streams_saved
streams_saved <- vclMatrix(readRDS("myStreams.rds"))
```

2.3. Plot the uniform random numbers (now deleted?)

3. Some non-uniform random number generation

3.1. Normal random number generation

We apply the Box-Muller transformation to $U(0,1)$ random numbers to generate standard normal random numbers. As shown in Algorithm 1, Box-Muller algorithm takes two independent, standard uniform random variables U_1 and U_2 and produces two independent, standard

Gaussian random variables X and Y, where R and Θ are polar coordinate random variables. The Box-Muller algorithm turns out to be the best choice for Gaussian transform on GPU compared to other transform methods [Howes and Thomas 2007], because this algorithm has no branching or looping, which are the things GPU is not good at.

Algorithm 1: Box-Muller algorithm.

1, Generate U_1, U_2 i.i.d. from $U(0,1)$;
2, Define

$$R = \sqrt{-2 * \log U_1},$$
$$\Theta = 2\pi * U_2,$$
$$X = R * \cos(\Theta),$$
$$Y = R * \sin(\Theta);$$

3, Take X and Y as two independent draws from $N(0,1)$;

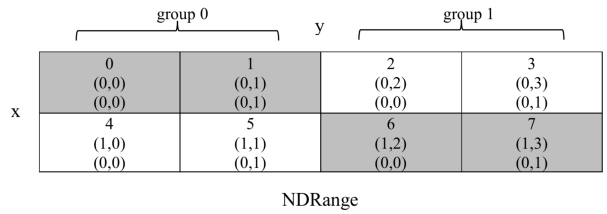
Listing 2 shows a fragment of the kernel that generates standard Gaussian random numbers. Aside from using local memory and the part that does the transformation, this kernel is the same with the kernel for generating uniform random numbers shown in Listing 1. If developers want to use other Gaussian transform methods, this kernel is easy to be adapted and incorporated in other R packages. The kernel executes over a 2-dimensional NDRange with 1×2 work-groups. `part[0]` and `part[1]` correspond to R and Θ in the formulas respectively. As the work-items in a work-group proceed at differing rates, `barrier(CLK_LOCAL_MEM_FENCE)` ensures correct ordering of memory operations to local memory, so that Gaussian random numbers $(X_1, Y_1), \ldots, (X_n, Y_n)$ are generated in pairs correctly, errors such as $(X_n, Y_{(n-1)})$ or $(X_{(n-1)}, Y_{(n)})$ are avoided.

Listing 2: Using streams to produce standard normal random numbers

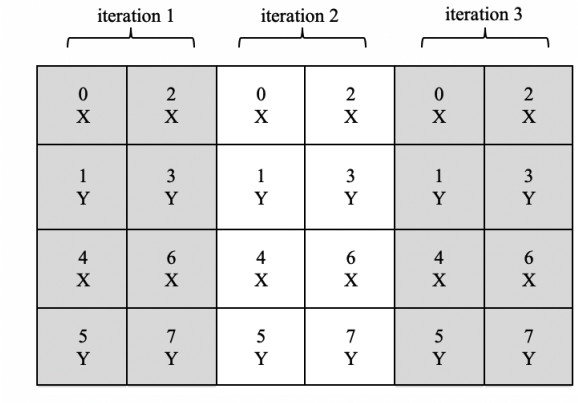
```
const int
index = get_global_id(0)*get_global_size(1) + get_global_id(1);
const int
DrowInc = 2*get_global_size(0), DrowStartInc = DrowInc * NpadCol;
int Drow, Dcol, DrowStart, Dentry;
uint temp;
uint g1[3], g2[3];
const int startvalue=index * NpadStreams;
local double part[2], cosPart1, sinPart1;

streamsToPrivate(streams,g1,g2,startvalue);

// Iterate for the rows
for(Drow=2*get_global_id(0),
    DrowStart = (Drow + get_local_id(1))* NpadCol;
    Drow < Nrow;
    Drow += DrowInc, DrowStart += DrowStartInc) {
  // Iterate for the columns
  for(Dcol=get_group_id(1), Dentry = DrowStart + Dcol;
```

```
      Dcol < Ncol;
      Dcol += get_num_groups(1), Dentry += get_num_groups(1)) {

    // Generates an uniform random integer
    temp = clrngMrg31k3pNextState(g1, g2);

    if(get_local_id(1)) {
      // Cache data to local memory
      part[1] = TWOPI_mrg31k3p_NORM_cl * temp;
      cosPart1 = cos(part[1]);
      sinPart1 = sin(part[1]);
    } else {
      part[0] = sqrt(-2.0*log(temp * mrg31k3p_NORM_cl));
    }

    // Wait until all work items have read the data and
    // it becomes visible
    barrier(CLK_LOCAL_MEM_FENCE);

    // Perform the operation and output the data
    if(get_local_id(1)) {
      out[Dentry] = part[0]*sinPart1;
    } else {
      out[Dentry] = part[0]*cosPart1;
    }
    barrier(CLK_LOCAL_MEM_FENCE);

  }//Dcol
}//Drow

streamsFromPrivate(streams,g1,g2,startvalue);
```

We illustrate in Figure 2 how this kernel executes in practice with the previous toy example: to create a 4×6 matrix of Gaussian random numbers using a 2×4 NDRange. Work-items are organized to be in 1 by 2 work-groups as the random numbers are generated in pairs using the Box-Muller method. The shading on the grids in the NDRange distinguishes work-groups. The numbers in the third row in each gird of NDRange represents the local index of each work-item. Each work-group generates a pair of Gaussian random numbers `(X,Y)` in each iteration, where work-item of '`get_local_id(1)`= 0' generates the `X` of a pair, and work-item of '`get_local_id(1)`= 1' generates the `Y` of a pair. Here with the loop iterating over columns, the cells of matrix are filled from left to right, that is, in the first iteration, the 8 work-items generate random numbers in the leftmost two columns (8 cells) of the matrix simultaneously, the second iteration fills random numbers in the middle two columns of the matrix, and the third iteration fills the rest two columns of the matrix simultaneously. The matrix filling process is the same as that shown in Figure 1.



(a)



(b)

Figure 2: Using streams to produce a 4×6 matrix of normal random numbers.

We generate a large-size matrix of 100 million double-precision Gaussian random numbers, and compare the run-time between using `stats::rnorm()` and `clrng::rnorm()`. The best performance we have seen using `clrng::rnorm()` is around 120 times faster with 512×128 work-items than using the `stats::rnorm()`. The difference in elapsed time becomes larger when matrix size goes larger.

```
streams <- createStreamsGpu(n = 512 * 128)
system.time(clrng::rnorm(c(10000,10000), streams=streams,
                         Nglobal=c(512,128), type="double"))
##    user  system elapsed
##   0.093   0.008   0.391


system.time(matrix(stats::rnorm(10000^2),10000,10000))
##    user  system elapsed
##   4.629   0.348   4.971
```

3.2. Exponential random number generation

The Exponential random variates are produced by applying the inverse transform method on i.i.d. $U(0,1)$ random numbers. The random variable $X \sim \text{Exponential}(\lambda)$ has cumulative distribution function $F_X(x) = 1 - e^{-\lambda x}$ for $x \geq 0$ and $\lambda > 0$. The inverse of F_X^{-1} is $F_x^{-1}(y) = -(1/\lambda)\log(1-y)$, for $0 \leq y < 1$. The kernel is not shown because it is mostly same as

the kernel for uniform and normal random numbers, basically just by applying the inverse transform formula to the `clrngMrg31k3pNextState(g1, g2)` in the kernel. Developers can make use of our kernel for generating other types of random variables on GPU in R.

Below is the code that we produce a 2 by 2 matrix of exponential random number with expectation 1.

```
r_matrix <- clrng::rexp(c(2,2), rate=1, Nglobal=c(1,4), type="double")
as.matrix(r_matrix)
##           [,1]      [,2]
## [1,] 1.2243377 0.9593675
## [2,] 0.7679958 0.5158689
```

The arguments of the `clrng::rexp()` are described as follows:

- `n`: a vector of length 2 specifying the row and column number if to create a matrix, or a number specifying the length if to create a vector.
- `streams`: streams. If missing, a stream with random initial state is supplied.
- `Nglobal`: global index space. Default is set as $(64, 8)$.
- `type`: "double" or "float" format of generated random numbers.

4. An application of uniform RNG: Fisher's simulation

The GPU-generated random numbers can be applied in suitable statistical simulations to accelerate the performance. One application of GPU-generated random numbers in **clrng** is Monte Carlo simulation for Fisher's exact test. Fisher's exact test is applied for analyzing usually 2×2 contingency tables when one of the expected values in table is less than 5. Different from methods which rely on approximation, Fisher's exact test computes directly the probability of obtaining each possible combination of the data for the same row and column totals (marginal totals) as the observed table, and get the exact p-value by adding together all the probabilities of tables as extreme or more extreme than the one observed. However, when the observed table gets large in terms of sample size and table dimensions, the number of combinations of cell frequencies with the same marginal totals gets very large, [Mehta and Patel 2011, p. 23] shows a 5×6 observed table that has 1.6 billion possible tables. Calculating the exact P-values may lead to very long run-time and can sometimes exceed the memory limits of your computer. Hence, the option `simulate.p.value = TRUE` in `stats::fisher.test()` is provided, which enables computing p-values by Monte Carlo simulation for tables larger than 2×2.

The test statistic calculated for each random table is $-\sum_{i,j} \log(n_{ij}!), i = (1, \ldots, I), j = (1, \ldots, J)$, (i.e., minus log-factorial of table), where I and J are the row and column number of the observed table. This test statistic can also be independently calculated for a table by `clrng::logfactSum()`. Given an observed table and a number of replicates B, the Monte Carlo simulation for Fisher's exact test does the following steps:

1. Calculate the test statistic for the observed table.

2. In each iteration, simulate a random table with the same dimensions and marginal totals as the observed table, compute and sometimes save the test statistic from the random table.
3. Count the number of iterations (*Counts*) that have test statistics less or equal to the one from the observed table.
4. Estimate p-value using $\frac{1+Counts}{B+1}$.

Step 1 and step 2-3 are done on a GPU with two kernels enqueued sequentially. For step 2, `clrng::fisher.sim()` adapted the function `rcont2()` used by `stats::fisher.test()` for constructing random two-way tables with given marginal totals. The algorithm [see Patefield 1981] samples the entries row by row, one at a time, conditional on the values of the entries already sampled. The conditional probabilities for the possible values of the next entry are updated dynamically each time a new entry is sampled. Then this entry is sampled by standard inversion of the cumulative distribution function, using one $U(0,1)$ random number. For an $I \times J$ table, this requires $(I-1)(J-1)$ random numbers (the last column and last row do not need to be sampled). Finally, one can compute the test statistic for this newly sampled table. On a GPU, this step can be replicated say n times in parallel by creating n distinct random streams and launching n separate work-items. Each work-item takes a random stream as input, performs all of Step 2 and 3, and returns the value of the test statistic on the GPU. Computing the p-value on the CPU (Step 4) is then a trivial operation. Step 2 of saving test statistics from the random tables is made optional, which can reduce the run-time. By the way, there are a lot of other more recent methods for sampling (larger) contingency tables, many of them use Markov Chain Monte Carlo, the use of streams and GPU would be quite different in that case. See for examples [Miller and Harrison 2013; Kayibi, Pirzada, and Chishti 2018; Dyer, Kannan, and Mount 1997]. Doing the `fisher.sim()` function on GPU opens up many possibilities for future work, the specific implementation we've done for the tables isn't necessarily the optimal one.

The arguments of the `clrng::fisher.sim()` are described as follows:

- `x`: a "vclMatrix".
- `N`: number of simulation runs.
- `streams`: streams.
- `type`: "double" or "float" of returned test statistics.
- `returnStatistics`: if TRUE, return all test statistics.
- `Nglobal`: global index space. Default is set as $(64, 16)$.

We show the advantage of `clrng::fisher.sim()` by computing the p-values for two real data examples: one with a relatively big p-value and another with a very small p-value, and compare the run-time with using `stats::fisher.test()` for each of the data sets on two computers: one with a very good CPU and an ordinary GPU, the other is equipped with an excellent GPU and an ordinary CPU. The R outputs for testing on computer 2 is shown in the following.

The 2-way contingency tables consist of selected data from the 2018 Natality public use file [for Health Statistics] from the Centers for Disease Control and Prevention's National Center

for Health Statistics, the 2018 natality data file may be downloaded at https://www.cdc.gov/nchs/data_access/VitalStatsOnline.htm.

4.1. Comparing run-time: Month data example

```
almost.1 <- 1 + 64 * .Machine$double.eps
observed_m= -logfactSum(month, c(16,16))/almost.1

month_gpu<-vclMatrix(month,type="integer")
result_month <- fisher.sim(month_gpu, 1e6, streams=streams,
                    type="double", returnStatistics=TRUE,  Nglobal = c(256,64))
result_month$simNum
## [1] 1015808
result_month$counts
## [1] 410825
result_month$p.value
## [1] 0.4044323


#using GPU
system.time(fisher.sim(month_gpu, 1e6, streams=streams,
            type="double", returnStatistics=FALSE, Nglobal = c(256,64)))
##    user  system elapsed
##   0.619   0.009   2.158
```

We requested one million simulations on GPU while the actual number of simulation is 1015808, and got 409524 cases whose test statitsic is below the observed threshold, so the p-value from `clrng::fisher.sim()` for this table is about 0.40443, which is close enough to the p-value from `stats::fisher.test()`: 0.40448. `clrng::fisher.sim()` takes about 0.69 seconds, **clrng** accounts for 4.64% of the elapsed time on CPU.

```
#using CPU
stats::fisher.test(month,simulate.p.value = TRUE,B=1015808)$p.value
## [1] 0.4039588
system.time(stats::fisher.test(month,simulate.p.value = TRUE,B=1015808))
##    user  system elapsed
##  10.184   0.004  10.178
```

4.2. Comparing run-time: Week data example

```
observed_w= -clrng::logfactSum(week, c(16,16))/almost.1
week_GPU<-gpuR::vclMatrix(week,type="integer")
result_weekgpu<-clrng::fisher.sim(week_GPU, 1e7, streams=streams,
                type="double",returnStatistics=TRUE,Nglobal = c(256,64))
result_weekgpu$simNum
```

```
## [1] 10010624
result_weekgpu$counts
## [1] 1205
result_weekgpu$p.value
## [1] 0.000120472
```

```
#using GPU
system.time(clrng::fisher.sim(week_GPU, 1e7, streams=streams,
            type="double", returnStatistics=FALSE,Nglobal = c(256,64)))
##    user  system elapsed
##   0.514   0.010  10.215
```

The "week" table has a much smaller p-value: around 0.0001249672, which should require a larger number of simulations to get a more accurate p-value. With more than ten million simulations, we get 1205 cases and a p-value around 0.000120472 by `fisher.sim()`. `stats::fisher.test()` takes 91.6 seconds, while `fisher.sim()` takes about 4.282 seconds, the elapsed time is decreased to about 4.67% of the time taken on CPU.

```
#using CPU
fisher.test(week,simulate.p.value = TRUE,B=10010624)$p.value
## [1] 0.0001252669
system.time(fisher.test(week,simulate.p.value = TRUE,B=10010624))
##    user  system elapsed
##  62.503   0.052  62.503
```

4.3. A summary of the results

We summarized the comparison results in the table 1 and plot the test statistics in Figure 3.

Table 1: Summary of comparions of Fisher's test simulation on different devices. Computer 1 is equipped with CPU Intel Xenon W-2145 3.7Ghz and AMD Radeon VII. Computer 2 is equipped with VCPU Intel Xenon Skylake 2.5Ghz and VGPU Nvidia Tesla V100.

	Computer 1		Computer 2		
B	Intel 2.5ghz	AMD Radeon	Intel 3.7ghz	NVIDIA V100	Data
P-value					
1m	0.403804	0.403507	0.4035606	0.403507	month
10m	0.0001251	0.0001274	0.0001202	0.0001274	week
Run-time					
1m	10.74	2.28	15	0.72	month
10m	63.24	10.82	91.58	4.18	week

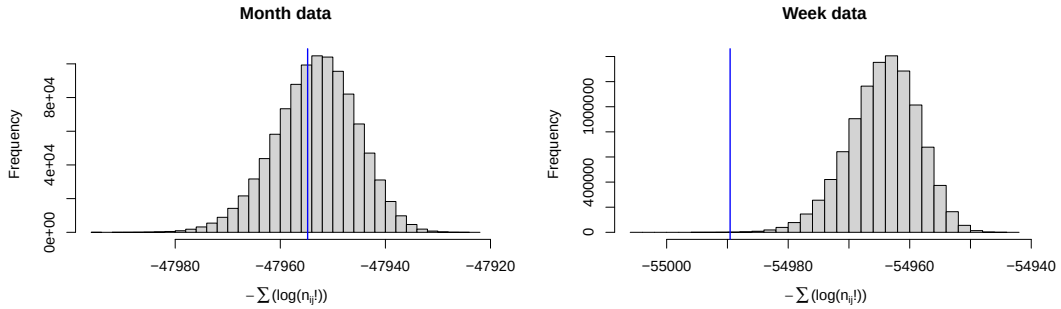


Figure 3: Approximate sampling distributions of the test statistics from the two examples "Month" and "Week". The test statistic values of the observed tables are calculated by `clrng::logfactSum()` and are marked with a blue line on each plot.

5. An application of normal RNG: Gaussian surface simulation

Given a parameter space X, a random field U on X is a collection of random variables $\{U_x : x \in X\}$. A Gaussian random field is a random field with the property that for any positive integer n and any set of locations $x_1, \ldots, x_n \in X$, the joint distribution of $U = (U_{x_1}, \ldots, U_{x_n})$ is multivariate Gaussian. The expectation $\mu(x)$ and covariance function Σ completely determine the distribution of U, where

$$\mu(x) = \mathrm{E}(U(x)),$$
$$\Sigma_{ij} = \mathrm{COV}(U(x_i), U(x_j)).$$

The common choice for the covariance function Σ is the Matérn covariance function [Matérn 1960]. The Matérn covariance between $U(x)$ and $U(x')$ takes the form

$$\Sigma(d; \sigma^2, \phi, \kappa) = \sigma^2 * \frac{2^{\kappa-1}}{\Gamma(\kappa)} (\sqrt{8\kappa}\frac{d}{\phi})^{\kappa} K_{\kappa}(\sqrt{8\kappa}\frac{d}{\phi}), \quad \text{where} \quad \phi \geq 0, \kappa \geq 0,$$

$d = \|x - x'\|$ is the Euclidean distance between two spatial points, σ^2 the variance of the random field U. $\Gamma(\cdot)$ is the standard gamma function, ϕ is the range parameter or scale parameter with the dimension of distance, it controls the rate of decay of the correlation as d increases. $K_{\kappa}(\cdot)$ is the modified Bessel function of the second kind with order κ, κ is the shape parameter which determines the smoothness of $U(x)$, specifically, $U(x)$ is m times mean-square differentiable if and only if $\kappa > m$. There are several alternative parameterisations of Matérn covariance functions in literatures [see Haskard 2007]. In **clrng**, we use the above form for computing Matérn covariance.

Liu, Li, Sun, and Yu [2019] gives a comprehensive review on 7 popular methods for Gaussian random field generation, all of these methods except matrix decomposition method, are approximations and has specific requirements on the type of grid or covariance functions. The matrix decomposition method is exact, it works for all covariance functions and can generate random field on all types of grids. Other R packages that offer simulation of Gaussian random fields like **geoR** [Ribeiro Jr. and Diggle 2001], does not work for large number of

locations; the **RandomFields** [Schlather, Malinowski, Menck, Oesting, and Strokorb 2015; Schlather, Malinowski, Oesting, Boecker, Strokorb, Engelke, Martini, Ballani, Moreva, Auel, Menck, Gross, Ober, Ribeiro, Ripley, Singleton, Pfaff, and R Core Team 2020] package can use different methods for simulation of Gaussian fields, among which the (modified) circulant embedding method [Dietrich and Newsam 1997] for covariance matrix decomposition is also an exact method (need to confirm???), however, it works only for isotropic Gaussian fields and on rectangular grids. **clrng** does exact simulation of exact Gaussian fields on the GPU as it relies on the matrix decomposition method. The implementation of this method is as follows.

Suppose $U = (U_1, U_2, \ldots, U_n)$ is a Gaussian random field with mean μ and covariance matrix Σ, without loss of generality, we assume mean value zero ($\mu = 0$). Since covariance matrix Σ is symmetric and positive-definite, we can take Cholesky decomposition of Σ, then we can simulate random samples from U using $U = L * D^{1/2} * Z$, where

- L is the lower unit triangular matrix in Cholesky decomposition of $\Sigma = L * D * L^\top$,
- D is a diagonal matrix.
- $Z = (Z_1, Z_2, \ldots, Z_n) \sim \text{MVN}(0, I_n)$, where I_n is a $n \times n$ identity matrix.

The matrix decomposition method is straightforward to implement, however, when a large number of locations (n) is involved, the cost of Cholesky decomposition of the covariance matrix is $\mathcal{O}(n^3)$, and the cost of matrix-vector multiplication $L * Z$ to generate the random field U is $\mathcal{O}(n^2)$ [Liu *et al.* 2019]. **clrng** package not only does the matrices decomposition (Cholesky decomposition and matrix-vector multiplication) in parallel, but also computes the covariance matrices in parallel on GPU. The following shows an example, in which we simulate 10 Gaussian random fields of matérn covariance with 5 sets of parameters at one time.

5.1. Simulating Gaussian random fields with Matérn covariances

```
library("gpuR")
library("clrng")
library("gpuBatchMatrix")
library('geostatsp')
```

Step 1, set up a 60×80 raster and coordinates on the raster, convert the matrix of coordinates `coordsSp@coords` to a "vclMatrix" object for later use.

```
Ngrid = c(60, 80)
NlocalCache = 1000
Nglobal = c(128, 64, 2)
Nlocal = c(4, 2, 2)
theType = "double"

myRaster = raster::raster( raster::extent(0,Ngrid[1]/Ngrid[2],5,6),Ngrid[1], Ngrid[2])
coordsSp = sp::SpatialPoints(raster::xyFromCell(myRaster, 1:raster::ncell(myRaster)))
```

```
coordsGpu = vclMatrix(coordsSp@coords,
                      nrow(coordsSp@coords),
                      ncol(coordsSp@coords), type=theType)
```

Step 2, create 5 parameter sets as a small example, in practical studies, there can be hundreds or thousands of parameter sets. Then, convert the matrix of parameters to a "vclMatrix" for later use by `gpuBatchMatrix::maternGpuParam()`.

```
myParamsBatch
##      shape range variance nugget anisoRatio anisoAngleRadians
## [1,]  1.25  0.50      1.5      0          1         0.0000000
## [2,]  2.15  0.25      2.0      0          4         0.4487990
## [3,]  0.55  1.50      2.0      0          4         0.4487990
## [4,]  2.15  0.50      2.0      0          4        -0.4487990
## [5,]  2.15  0.50      2.0      0          2         0.7853982
paramsGpu = gpuBatchMatrix::maternGpuParam(myParamsBatch, type=theType)
```

Step 3, compute Matérn covariance matrices using `gpuBatchMatrix::maternBatch()`, the returned Matérn covariance matrices are each of size 4800×4800 and are stacked by row in the output `maternCov`.

```
maternCov = vclMatrix(0, nrow(paramsGpu)*nrow(coordsGpu),
                         nrow(coordsGpu),type=theType)
dim(maternCov)
## [1] 24000  4800
gpuBatchMatrix::maternBatch(maternCov, coordsGpu, paramsGpu,
                        Nglobal=c(128,64), Nlocal=c(16,4))
```

Step 4, the first argument `maternCov` in `gpuBatchMatrix::cholBatch()` specifies the matrix object to take Cholesky decomposition. Unit lower triangular matrices L_i's are returned and stacked by row in `maternCov`, `diagMat` stores the diagonal values of each D_i in a row, for example, if each batch Σ_i is of size $n \times n$, then each batch L_i is $n \times n$, and each batch D_i is $1 \times n$.

```
diagMat = vclMatrix(0, nrow(paramsGpu), ncol(maternCov), type = theType)
gpuBatchMatrix::cholBatch(maternCov, diagMat, numbatchD=nrow(myParamsBatch),
                      Nglobal= c(128, 8),
                      Nlocal= c(32, 8),
                      NlocalCache=1000)
```

Step 5, generate 2 standard Gaussian random vectors `zmatGpu`$= (Z_1, Z_2)$ using `clrng::rnorm()`, in which `c(nrow(materCov),2)` specifies the number of rows and columns of `zmatGpu`.

```
streamsGpu <- createStreamsGpu(n=128*64)
```

```
zmatGpu = rnorm(c(nrow(maternCov),2), streams=streamsGpu, Nglobal=c(128,64),
                type = theType)
```

Step 6, use `gpuBatchMatrix::multiplyLowerDiagonalBatch()` to compute $U = L{*}D^{(1/2)}{*}Z$ in batches, see the following illustration. `simMat` is the output matrix for U, `maternCov`, `diagMat`, and `zmatGpu` correspond to the matrices of L, D and $Z = (Z_1, Z_2)$ respectively.

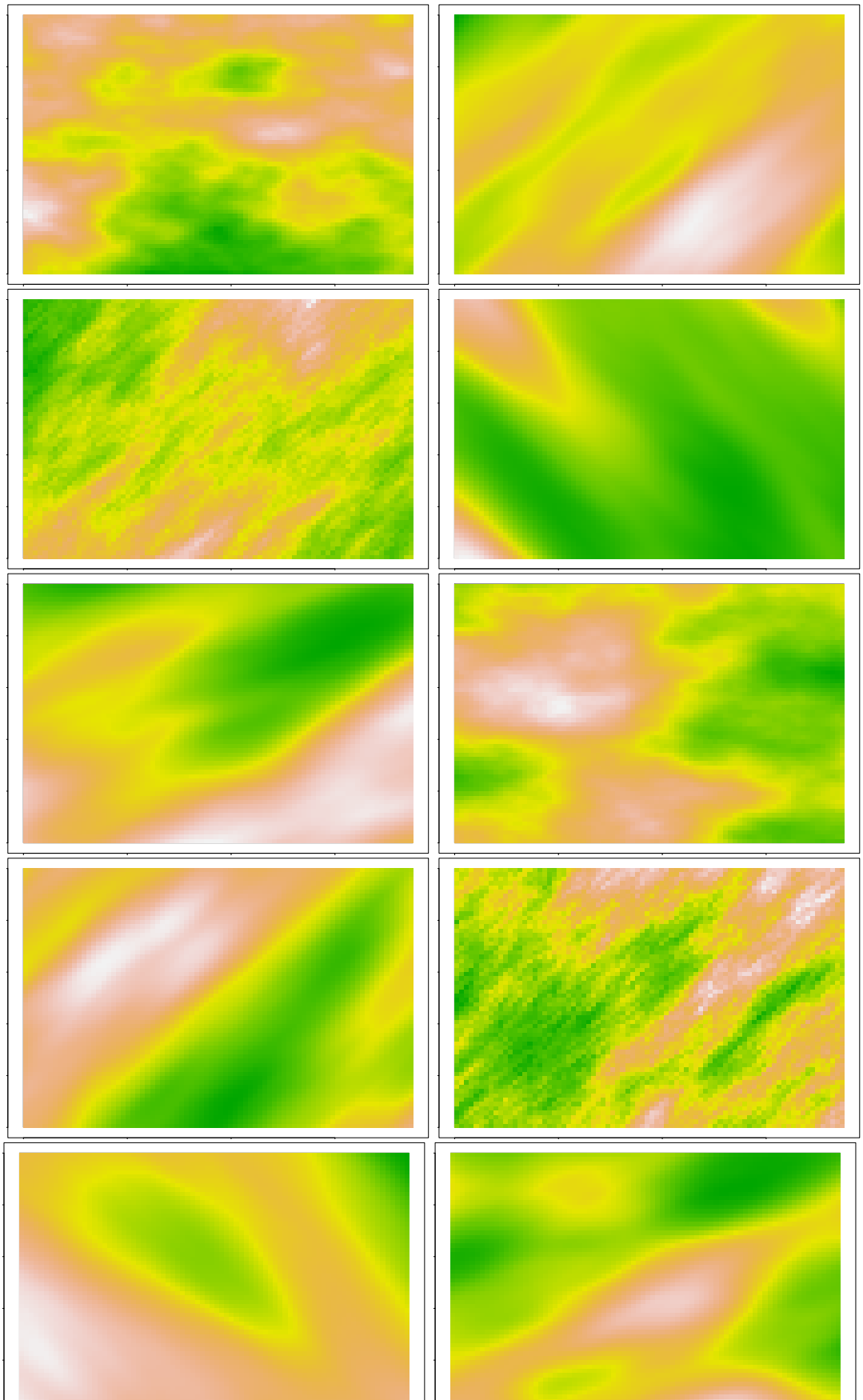
```
simMat = vclMatrix(0, nrow(zmatGpu), ncol(zmatGpu),
                   type = theType)

gpuBatchMatrix::multiplyLowerDiagonalBatch(
  simMat, maternCov, diagMat, zmatGpu,
  diagIsOne = 1L, # diagonal of L is one
  transformD = "sqrt", # take the square root of each element of D
  Nglobal,
  Nlocal,
  NlocalCache)
```

$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ \vdots \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \end{bmatrix} * \begin{bmatrix} Z_{11} & Z_{12} \end{bmatrix} = \begin{bmatrix} L_1 D_1 Z_{11} & L_1 D_1 Z_{12} \\ L_2 D_2 Z_{11} & L_2 D_2 Z_{12} \\ L_3 D_3 Z_{11} & L_3 D_3 Z_{12} \end{bmatrix} \tag{1}$$

Step 7, Finally, plot the 10 simulated Gaussian random surfaces.

```
library(raster)
simRaster = brick(myRaster, nl = ncol(simMat)*nrow(paramsGpu))
values(simRaster) = as.vector(as.matrix(simMat))
names(simRaster) = apply(expand.grid('par',1:nrow(paramsGpu),
                                     'sim', 1:ncol(simMat)), 1, paste, collapse='')
```

```
par(mar=rep(0.1, 4))
for(D in names(simRaster)) {
  plot(extent(simRaster))
  plot(simRaster[[D]], legend=FALSE, add=TRUE)
}
```



6. Discussion

The package **clrng** has been created to make GPU-generated uniform and normal random numbers accessible for R users, it enables reproducible research in simulations by setting seeds in streams on GPU. We further applied the GPU-generated random numbers in suitable statistical simulations, such as the Monte Carlo simulation for Fisher's exact test, and (exact) Gaussian spatial surfaces simulation, for which we are able to calculate quantiles for the normal distribution, compute matérn covariance matrices, do Cholesky decomposition and matrix multiplication in batches on GPU. Most of these functions uses local memory on GPU. Comparison of performance between using **clrng** and using classic R on CPU for some real data examples has demonstrated significant improvement in execution time.

By leveraging the **gpuR** package, **clrng** provides a user-friendly interface that bridges R and OpenCL, users can use the facilities in our package without the need to know the complex OpenCL or even the C++ code. **clrng** is portable as its backend OpenCL supports multiple types of processors, and it is also flexible as its kernels can be incorporated or reconstructed in other R packages for other applications.

clrng is limited by the number and type of OpenCL RNGs used and the features of the **gpuR** package, as it depends upon the **gpuR** package, if developers wants to use our package to do something that's not supported in **gpuR**, for example "sparse" class objects, they would have to write OpenCL code to implement it.

Future work on **clrng** package could be: (1) Explore other RNGs, for example the MRG32k3a, LFSR113, and Philox-4×32-10 generators in clRNG library, and compare the R performance between using different GPU RNGs. (2) Now that the package can do Cholesky decomposition and matrix multiplication in batches on GPU, which is a motivation for us to work on parallel likelihood evaluations on GPU for Gaussian spatial models in the next step. (3) Create a "sparse" matrix class on GPU and use it for simulating Gaussian random fields with sparse correlation structures such as the Gaussian Markov random fields.

References

Allaire J, Eddelbuettel D, Golding N, Tang Y (2016). *tensorflow: R Interface to TensorFlow*. URL https://github.com/rstudio/tensorflow.

Bengtsson H (2021). "A Unifying Framework for Parallel and Distributed Processing in R using Futures." 10.32614/RJ-2021-048, URL https://journal.r-project.org/archive/2021/RJ-2021-048/index.html.

Box GE (1958). "A note on the generation of random normal deviates." *Ann. Math. Statist.*, **29**, 610–611.

Determan Jr C (2017). *gpuR: GPU Functions for R Objects*. R package version 2.0.0, URL http://github.com/cdeterman/gpuR.

Dietrich C, Newsam G (1997). "Fast and Exact Simulation of Stationary Gaussian Processes through Circulant Embedding of the Covariance Matrix." *SIAM J. Sci. Comput.*, **18**, 1088–1107.

Dyer M, Kannan R, Mount J (1997). "Sampling contingency tables." *Random Structures & Algorithms*, **10**(4), 487–506.

Eddelbuettel D (????). "CRAN Task View: High-Performance and Parallel Computing with R." https://cran.r-project.org/web/views/HighPerformanceComputing.html.

Eddelbuettel D (2021). "Parallel computing with R: A brief review." *Wiley Interdisciplinary Reviews: Computational Statistics*, **13**(2), e1515.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08. URL https://www.jstatsoft.org/v40/i08/.

for Health Statistics NC (????). "Natality 2018. Public use file." Annual internet product. 2019. Available at http://www.cdc.gov/nchs/data_access/VitalStatsOnline.htm.

Haskard KA (2007). *An anisotropic Matern spatial covariance model: REML estimation and properties.* Ph.D. thesis.

Howes L, Thomas D (2007). "Efficient random number generation and application using CUDA." *GPU gems*, **3**, 805–830.

Kayibi KK, Pirzada S, Chishti T (2018). "Sampling contingency tables." *AKCE International Journal of Graphs and Combinatorics*, **15**(3), 298–306. ISSN 0972-8600. doi:https://doi.org/10.1016/j.akcej.2017.10.001. URL https://www.sciencedirect.com/science/article/pii/S0972860017302396.

L'Ecuyer P (2015). "Random Number Generation with Multiple Streams for Sequential and Parallel Computers." In L Yilmaz, WKV Chan, I Moon, TMK Roeder, C Macal, MD Rossetti (eds.), *Proceedings of the 2015 Winter Simulation Conference*, pp. 31–44. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer P, Munger D, Oreshkin B, Simard R (2017). "Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on GPUs." *Mathematics and Computers in Simulation*, **135**, 3–17. Open access at http://dx.doi.org/10.1016/j.matcom.2016.05.005.

L'Ecuyer P, Nadeau-Chamard O, Chen YF, Lebar J (2021). "Multiple Streams with Recurrence-Based, Counter-Based, and Splittable Random Number Generators." In *Proceedings of the 2021 Winter Simulation Conference*. To appear, see https://www-labs.iro.umontreal.ca/~lecuyer/myftp/papers/wsc21rng.pdf.

L'Ecuyer P, Simard R (2007). "TestU01: A C Library for Empirical Testing of Random Number Generators." *ACM Transactions on Mathematical Software*, **33**(4), Article 22.

L'ecuyer P, Simard R, Chen EJ, Kelton WD (2002). "An object-oriented random-number package with many long streams and substreams." *Operations research*, **50**(6), 1073–1075.

L'Ecuyer P, Touzin R (2000). "Fast Combined Multiple Recursive Generators with Multipliers of the Form $a = \pm 2^q \pm 2^r$." In JA Joines, RR Barton, K Kang, PA Fishwick (eds.), *Proceedings of the 2000 Winter Simulation Conference*, pp. 683–689. IEEE Press.

Liu Y, Li J, Sun S, Yu B (2019). "Advances in Gaussian random field generation: a review." *Computational Geosciences*, **23**(5), 1011–1047.

L'Ecuyer P (2012). "Random number generation." In *Handbook of computational statistics*, pp. 35–71. Springer.

L'Ecuyer P, Munger D, Kemerchou N (2015). "clRNG: a random number API with multiple streams for OpenCL." Report, URL http://shorturl.at/lmxF7.

Matérn B (1960). "Spatial variation, Technical Report." *Statens Skogsforsningsinstitut, Stockholm*.

Mehta CR, Patel NR (2011). "IBM SPSS exact tests." *Armonk, NY: IBM Corporation*, pp. 23–24.

Miller JW, Harrison MT (2013). "Exact sampling and counting for fixed-margin matrices." *The Annals of Statistics*, **41**(3), 1569 – 1592. doi:10.1214/13-AOS1131. URL https://doi.org/10.1214/13-AOS1131.

Patefield W (1981). "Algorithm AS 159: An Efficient Method of Generating Random R × C Tables with Given Row and Column Totals." *Applied Statistics*, **30**(1), 91–7.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org.

Ribeiro Jr P, Diggle P (2001). "geoR: a package for geostatistical analysis." *R-NEWS*, **1**(2), 15–18. ISSN 1609-3631. URL http://cran.R-project.org/doc/Rnews.

Robert CP, Casella G (2004). "Random variable generation." In *Monte Carlo Statistical Methods*, pp. 35–77. Springer.

Rupp K, Tillet P, Rudolf F, Weinbub J, Morhammer A, Grasser T, Jungel A, Selberherr S (2016). "ViennaCL—linear algebra library for multi-and many-core architectures." *SIAM Journal on Scientific Computing*, **38**(5), S412–S439.

Schlather M, Malinowski A, Menck PJ, Oesting M, Strokorb K (2015). "Analysis, Simulation and Prediction of Multivariate Random Fields with Package RandomFields." *Journal of Statistical Software*, **63**(8), 1–25. URL http://www.jstatsoft.org/v63/i08/.

Schlather M, Malinowski A, Oesting M, Boecker D, Strokorb K, Engelke S, Martini J, Ballani F, Moreva O, Auel J, Menck PJ, Gross S, Ober U, Ribeiro P, Ripley BD, Singleton R, Pfaff B, R Core Team (2020). *RandomFields: Simulation and Analysis of Random Fields*. R package version 3.3.8, URL https://cran.r-project.org/package=RandomFields.

Sevcikova H, Rossini T, L'Ecuyer P, Sevcikova MH (2015). "Package 'rlecuyer'."

Zhao P (2016). "R with Parallel Computing from User Perspectives." URL https://parallelr.com/2016/09/10/r-with-parallel-computing/.

The "month" and "week" tables for testing `fisher.sim()` in section 4.1 and 4.2 are shown below, where The row variable of the "month" table represents the twelve months: from January to December, and for the "week" table it represents the seven weekdays: Monday to Sunday. The column variables of these two tables represent the twelve categories of congenital anomalies of the newborn: 1) Anencephaly; 2) Meningomyelocele/Spina bifida; 3) Cyanotic congenital heart disease; 4) Congenital diaphragmatic hernia; 5) Omphalocele; 6) Gastrochisis; 7) Limb reduction defect; 8) Cleft lip with or without cleft palate; 9) Cleft palate alone; 10) Down syndrome; 11) Suspected chromosomal disorder; and 12) Hypospadias.

Table 2: Month

	Ane	Men	Cya	Her	Omp	Gas	Lim	Cle	Pal	Down	Chro	Hypo
Jan	29	55	172	46	39	73	48	183	77	103	102	174
Feb	25	45	175	35	31	55	34	142	81	115	100	180
Mar	31	48	182	41	47	72	40	200	86	90	96	180
Apr	34	45	186	36	32	75	42	173	56	87	90	193
May	33	40	187	46	24	80	35	180	75	91	100	197
Jun	34	48	189	35	33	75	45	154	74	102	100	182
Jul	26	43	198	34	21	74	36	179	79	86	92	193
Aug	24	41	189	44	43	62	48	183	88	109	94	194
Sept	34	44	147	40	37	66	36	158	73	112	103	196
Oct	25	43	207	45	31	65	49	181	77	108	115	220
Nov	36	55	188	39	39	62	43	144	68	98	79	173
Dec	23	48	196	31	31	71	31	177	86	86	73	156

Table 3: Week

	Ane	Men	Cya	Her	Omp	Gas	Lim	Cle	Pal	Down	Chro	Hypo
Mon	30	34	173	37	23	80	49	191	83	122	109	216
Tue	60	121	383	80	83	131	71	349	146	164	168	352
Wed	51	106	417	92	73	145	72	333	136	179	196	351
Thu	60	86	362	69	74	120	85	326	132	220	187	359
Fri	52	94	347	87	59	123	68	323	145	170	166	345
Sat	52	63	323	67	64	135	73	316	170	189	188	357
Sun	49	51	211	40	32	96	69	216	108	143	130	258

Affiliation:

Pierre L'Ecuyer
Department of Computer Science and Operations Research
University of Montréal
Pavillon André-Aisenstadt 2920, chemin de la Tour Montréal QCH3T 1J4
E-mail: lecuyer@iro.umontreal.ca