Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

A tool set for random number generation on GPUs in R

Ruoyong Xu
University of Toronto

Patrick Brown
University of Toronto

Pierre LEcuyer
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 Rs 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 [**?**] has become a very important topic and attracted lots of interest from researchers [see **?**, 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 [**?**]. 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 multicore processors. Parallel computing is the solution to these drawbacks, for an overview of current parallel computing approaches with R, see CRAN Task View by **?** 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** [**?**] 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** [**?**] 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** [**?**] library, it provides a bridge between R and non-proprietary GPUs through the OpenCL (Open Computing Language) backend, which when combined with **Rcpp** [**?**] 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**, **future** [**?**] and **rlecuyer** [**?**] are able to generate random numbers in parallel on multicore CPUs. More specifically, **parallel** writes an interface for the **RngStreams**, a C++ library by **?** which is based on a combined multiple-recursive generator (MRG) MRG32k3a. **future** and **rlecuyer** 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: **???**.

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** [**?**] 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 Fishers 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. **?** 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. **?** and **?** present many general transformation methods for generating non-uniform random variables, for example, the most frequently used inverse transform method, the Box-Muller algorithm [**?**] 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-4Œ32-10 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 [**?**] 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-4Œ32-10 [**?**]. MRG31k3p was also statistically tested extensively and successfully, [see **?**].

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. A popular way of obtaining multiple streams is to take an RNG with a long period and cut the RNG's sequence into very long disjoint pieces of equal length Z, and use each piece as a separate stream. Creating a new stream amounts to computing its starting point. In general, a stream object contains three elements: 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). 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. Each of these streams can also be partitioned into substreams with equally-spaced starting points [**??**].

In clRNG library, the MRG31k3p RNG's entire period of length 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 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 $(12345, 12345, 12345, 12345, 12345, 12345)$.

clrng is able to create streams both on the host and on the GPU device. The `createStreamsCpu()` function does the former. The R output below creates 4 streams on the host.

```
##                [,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 move streams to the GPU by converting them to a '`vclMatrix`'.

`createStreamsGpu()` creates streams directly on the GPU and returns a '`vclMatrix`', which makes it slightly more efficient when the number of streams is large. However, any data is lost when the R session is terminated.

2.2. Using streams to generate uniform random numbers

The streams created sequentially are then used by the GPU's work-items to generate random numbers. In **clrng** each work-item takes one distinct stream to generate random numbers, so the number of streams should always be greater than 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 2.2. Users can set `verbose=2` in `runif()` to print out the kernels. 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. `Npad` is the internal number of columns of `out`, `NpadStreams` is the internal number of columns of `streams` (these variables are defined in macros not shown here). `index` represents the position of a work-item when mapping it to a one-dimensional data structure. Each stream's current state is copied to the private memory of each work-item by the function `streamsToPrivate()`, in which `g1` and `g2` point to the first three and second three elements of stream states, respectively, and `startvalue` tells the starting position of the particular stream to be copied to the work-item. The function `clrngMrg31k3pNextState()` generates an uniform random integer named `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 $1/2147483648$ depending on the precision type. If to generate random integers, `temp` is not scaled. At the end of generating random numbers, streams are transferred back to global memory through the function `streamsFromPrivate()`.

```
__kernel void mrg31k3pMatrix(
  __global int* streams,
  __global float* out,
```

```
  int Nrow, int Ncol, int Npad){

int Drow, Dcol;
uint g1[3], g2[3];
float temp, fact = mrg31k3p_NORM_cl;

streamsToPrivate(streams,g1,g2,startvalue);

for(Drow=get_global_id(0); Drow < Nrow;
    Drow += get_global_size(0)){

    for(Dcol=get_global_id(1); DcolBlock < Ncol;
        Dcol += get_global_size(1)){

        temp = fact * clrngMrg31k3pNextState(g1, g2);
        out[Drow * Npad + Dcol] = temp;

  }//Dcol
}//Drow

streamsFromPrivate(streams,g1,g2,startvalue);

}//kernel
```

Now we use the streams created in section 2.1 to generate a vector of double-precision i.i.d. $U(0,1)$ random numbers with `clrng::runif()`. 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.

```
## [1] 0.735 0.842 0.614 0.216 0.110 0.870
```

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.
- `verbose`: print extra information if verbose > 1. Default is FALSE.

Reusing `myStreamsGpu` 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.

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

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 **??**, 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 is a very good choice for Gaussian transform on GPU compared to other transform methods [**?**], 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 3.1 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 2.2. 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 index space with 1×2 work-groups. `part[0]` and `part[1]` correspond to R and Θ in the formulas respectively. `PI_2` is defined to be $\pi/2$. 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.

```
__kernel void mrg31k3pMatrix(
  __global int* streams,
  __global double* out,
  int Nrow, int Ncol, int Npad, int NpadStreams){

int Drow, Dcol;
uint g1[3], g2[3];
local double part[2];
```

```
int startvalue = (get_global_id(0) * get_global_size(1) +
get_global_id(1)) * NpadStreams;

double sinOrCosPart1, addForSine = - get_local_id(1) * PI_2;
double temp, fact = mrg31k3p_NORM_cl;
if(get_local_id(1)){
  fact = TWOPI_mrg31k3p_NORM_cl;
}

streamsToPrivate(streams,g1,g2, startvalue);

for(Drow=get_global_id(0); Drow < Nrow;
    Drow += get_global_size(0)){

    for(Dcol=get_global_id(1); DcolBlock < Ncol;
        Dcol += get_global_size(1)){

      temp = fact * clrngMrg31k3pNextState(g1, g2);
      part[get_local_id(1)] = temp;

      if(!get_local_id(1)) {
        part[0] = sqrt(-2.0*log(part[0]));
      }
      barrier(CLK_LOCAL_MEM_FENCE);

      // sine for local[1], cos for local[0]
      sinOrCosPart1 = cos(part[1] + addForSine);
      out[Drow * Npad + Dcol] = part[0]*sinOrCosPart1;

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

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.

```
##    user  system elapsed
##   0.077   0.000   0.076

##    user  system elapsed
##   7.103   0.651   7.750
```

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 $F_X(\cdot)$ 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, except for the part that does the inverse transform.

Below is an example that produces a matrix of Exponential random numbers with expectation equal to 1.

```
##       [,1]  [,2]  [,3]  [,4]
## [1,] 0.343 0.364 0.542 0.054
## [2,] 1.113 3.642 0.200 2.770
```

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 $(64, 8)$.
- `type`: "double" or "float" format of generated random numbers.
- `verbose`: print extra information if verbose > 1. Default is FALSE.

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. Fishers 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, [**?**, 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 using the `rcont2()` algorithm, 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 **?**] 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 [**???**]. 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 contingency table, a "vclMatrix" object.
- `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 $(64, 16)$.

Users request N number of replicates, while the actual number of replicates to be executed on GPU is larger (`ceiling(N/prod(Nglobal))*prod(Nglobal)`). 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.

4.1. Comparing run-time: Month data example

Table 1: Monthly birth anomaly data

	Ane	Men	Cya	Her	Omp	Gas	Lim	Cle	Pal	Dow	Chr	Hyp
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

The 2-way contingency tables consist of selected data from the 2018 Natality public use data [**?**] from the Centers for Disease Control and Preventions National Center for Health Statistics (NCHS), the 2018 natality data file may be downloaded at `https://www.cdc.gov/nchs/data_access/VitalStatsOnline.htm`. Table 1 is a 12×12 table that shows frequencies for congenital anomalies of the newborn by birth month in 2018 within the United States. 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. The first 5 rows of the table are displayed below.

```
##    user  system elapsed
##   0.316   0.000   0.316
```

```
## [1] -47955
```

```
## [1] 1015808
```

```
## [1] 410825
```

```
## [1] 0.404
```

We got 410825 cases whose test statistics are below the observed threshold based on an actual number of 1015808 simulations, 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()`. `clrng::fisher.sim()` takes about 0.31 seconds, **clrng** accounts for 2.17% of the elapsed time on CPU.

Table 2: Day-of-week birth anomaly data

	Ane	Men	Cya	Her	Omp	Gas	Lim	Cle	Pal	Dow	Chr	Hyp
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

```
##    user  system elapsed
##   0.137   0.000   0.137
```

```
## [1] 0.4
```

4.2. Comparing run-time: Week data example

Table 2 is a 7×12 table shows frequencies for congenital anomalies of the newborn by birth day of week in 2018 within the United States.

```
##    user  system elapsed
##   3.854   0.011   3.864
```

```
## [1] -54990
```

```
## [1] 1e+07
```

```
## [1] 1193
```

```
## [1] 0.000119
```

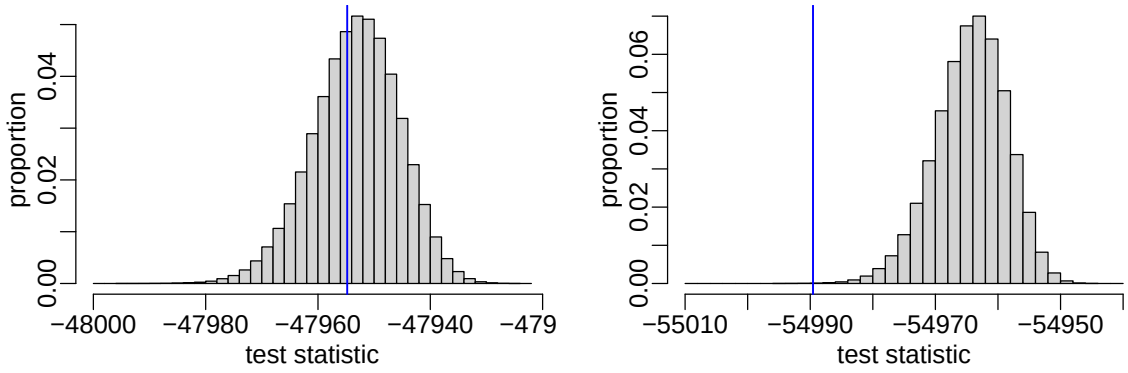
The "week" table has a much smaller p-value: around 0.0001249672, which should require a larger number of simulations to get a more accurate p-value. With more than ten million simulations, we get 1205 cases and a p-value around 0.000120472 by `fisher.sim()`. `stats::fisher.test()` takes 91.6 seconds,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

```
##    user  system elapsed
##   0.009   0.000   0.009
```

```
## [1] 0.000999
```

Table 3: 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



(a) Month data

(b) week data

Figure 1: Approximate sampling distributions of the test statistics from the two examples Month and Week. The test statistic values of the observed tables are marked with a blue line on each plot.

4.3. A summary of the results

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

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 of U and covariance function Σ completely determine the distribution of U, where

$$U = [U(x_1), \ldots, U(x_n)]^\top \sim \text{MVN}(0, \Sigma)$$

$$\Sigma_{ij} = \text{COV}[U(x_i), U(x_j)] = \sigma^2 \frac{2^{\kappa-1}}{\Gamma(\kappa)} \left(\sqrt{8\kappa}\frac{|x_i = x_j|}{\phi}\right)^\kappa K_\kappa \left(\sqrt{8\kappa}\frac{|x_i = x_j|}{\phi}\right).$$

The common choice for the covariance function Σ is the Matérn covariance function [**?**]. The Matérn covariance between $U(x_i)$ and $U(x_j)$ takes the above form. 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 distance 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 **?**]. In **clrng**, we use the above form for computing Matérn covariance.

? 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** [**?**], does not work for large number of locations; the **RandomFields** [**??**] package can use different methods for simulation of Gaussian fields, among which the (modified) circulant embedding method [**?**] 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 Algorithm **??**,

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 $\mathcal{O}(n^2)$ [**?**]. Our another R package **gpuBatchMatrix** is able to compute batches of Matérn covariance matrices in parallel, and do Cholesky decomposition and matrix-matrix multiplication in batches in parallel on GPU. The following shows an example, in which we simulate

Algorithm 2: Gaussian random fields simulation using covariance matrix decomposition method.

1, Calculate the covariance matrix Σ between locations;

2, Compute the Cholesky decomposition of $\Sigma = L \cdot D \cdot L^{\top}$, L is the lower unit triangular matrix, D is a diagonal matrix;

3, Generate on GPU a random matrix $Z = (Z_1, Z_2, \ldots, Z_n) \sim \mathrm{MVN}(0, I_n)$, where I_n is a $n \times n$ identity matrix;

4, Compute the random samples from U in batches using $U = L \cdot D^{\frac{1}{2}} \cdot Z$;

on GPU 10 Gaussian random fields of matérn covariance at one time with 5 sets of parameters, by taking advantage of the GPU-capabilities provided by **clrng** and **gpuBatchMatrix** together.

5.1. Simulating Gaussian random fields with Matérn covariances

```
R> library("gpuR")
R> library("clrng")
R> library("gpuBatchMatrix")
R> 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.

```
R> data("swissRain", package = "geostatsp")
R> myRaster = geostatsp::squareRaster(swissBorder, 60)
R> myRaster
```

```
## class      : RasterLayer
## dimensions : 38, 60, 2280  (nrow, ncol, ncell)
## resolution : 5811, 5811  (x, y)
## extent     : 2485351, 2833996, 1075013, 1295822  (xmin, xmax, ymin, ymax)
## crs        : +proj=somerc +lat_0=46.9524055555556 +lon_0=7.43958333333333 +k_0=1 +x_0=2
```

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()`.

```
##      shape range variance nugget anisoRatio
## [1,]  1.25 50000      1.5      0          1
## [2,]  2.15 60000      2.0      0          4
## [3,]  0.60 30000      2.0      0          2
## [4,]  3.00 30000      2.0      0          2
##      anisoAngleRadians
## [1,]             0.000
```

```
## [2,]            0.449
## [3,]            0.449
## [4,]            0.449
```

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`.

```
## [1] 9120 2280
```

Step 4, the first argument `A` in `gpuBatchMatrix::cholBatch()` specifies the object to take Cholesky decomposition. Computed unit lower triangular matrices L_i's are stacked by row and stored in `maternCov`. Each row of `diagMat` stores the diagonal values of each D_i, 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 $1 \times n$.

$$\begin{bmatrix} \Sigma_1 \\ \Sigma_2 \\ \Sigma_3 \\ \vdots \end{bmatrix} \rightarrow \begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ \vdots \end{bmatrix} \text{ and } \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \end{bmatrix} \tag{1}$$

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`.

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.

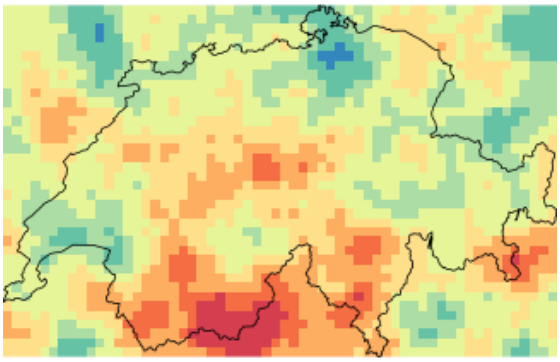
$$\begin{bmatrix} L_1 \\ L_2 \\ L_3 \\ \vdots \end{bmatrix} * \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \end{bmatrix} * \begin{bmatrix} Z_{11} & Z_{12} \end{bmatrix} = \begin{bmatrix} L_1 D_1 Z_{11} & L_1 D_1 Z_{12} \\ L_2 D_2 Z_{11} & L_2 D_2 Z_{12} \\ L_3 D_3 Z_{11} & L_3 D_3 Z_{12} \\ \vdots & \vdots \end{bmatrix} \tag{2}$$

Step 7, Finally, plot the 10 realizations.

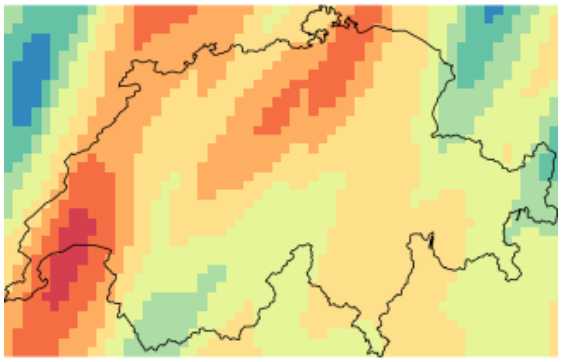
```
R> simRaster = raster::brick(myRaster, nl = ncol(simMat) * nrow(params))
R> values(simRaster) = as.vector(as.matrix(simMat))
```

6. Discussion

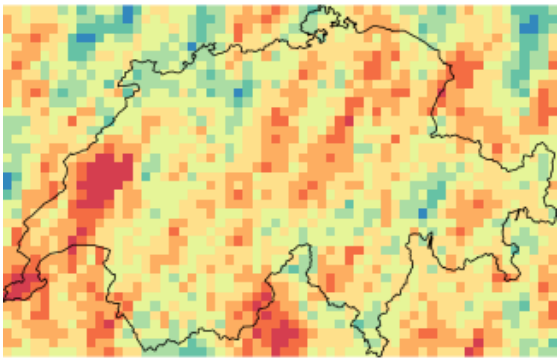
The package **clrng** has been created to make GPU-generated uniform and normal random numbers accessible for R users, it enables reproducible research in simulations by setting seeds in streams on GPU. We further applied the GPU-generated random numbers in suitable statistical simulations, such as the Monte Carlo simulation for Fishers exact test, and (exact) Gaussian spatial surfaces simulation, for which we are able to calculate quantiles for the



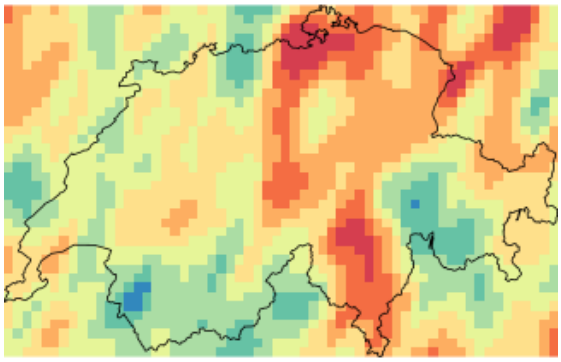
(a) parameter 1, simuation 1



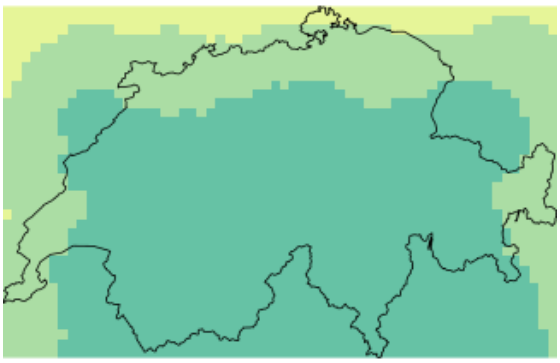
(b) parameter 2, simuation 1



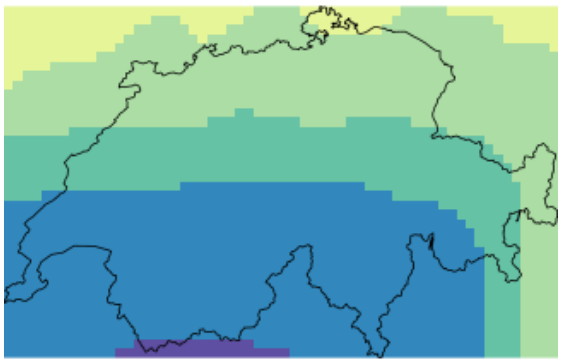
(c) parameter 3, simuation 1



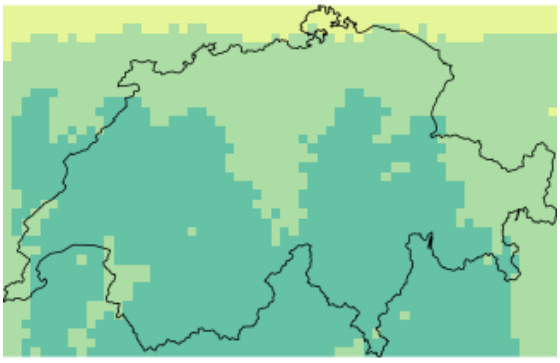
(d) parameter 4, simuation 1



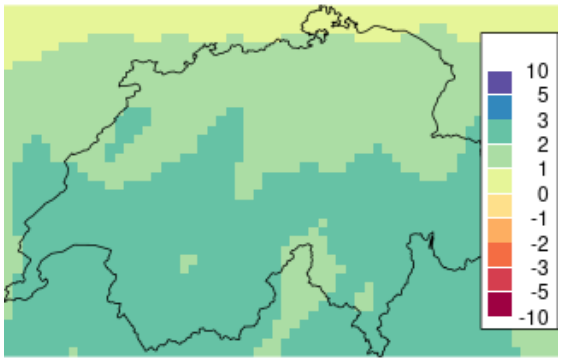
(e) parameter 1, simuation 2



(f) parameter 2, simuation 2



(g) parameter 3, simuation 2



(h) parameter 4, simuation 2

Figure 2: Simulated Gaussian random fields

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-4Œ32-10 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.

Affiliation:

Pierre LEcuyer
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

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd